UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA DELL'INFORMAZIONE

Valutazione automatica della coerenza tra codice sorgente e relativa documentazione

Relatore

Prof. Antonio Giunta

Laureando

Samuele Mega

ANNO ACCADEMICO 2023 – 2024

Data di laurea 16 novembre 2023

Indice

Capitolo 1

Finalità

1.1 Introduzione

Il linguaggio C. Nato negli anni '70 nei laboratori *Bell*, il *C* ha decretato una rivoluzione nel panorama dell'Ingegneria Informatica. In un contesto dominato da linguaggi di programmazione dipendenti dalla piattaforma hardware, come *Assembly*, e fortemente orientati a specifici ambiti applicativi, quali *Fortran* e *COBOL*, il linguaggio *C* si è distinto per portabilità e flessibilità. Unendo una sintassi semplice a strumenti di controllo della memoria potenti e granulari, il *C* permette la scrittura di codice espressivo ed altamente efficiente. Queste, ed altre caratteristiche, ne hanno determinato la fortuna, giustificando la pervasività che oggi mostra nei più disparati settori della programmazione.

Nonostante sia trascorso mezzo secolo dalla nascita del linguaggio *C*, esso tuttora risulta un importante attore nell'industria del software. La sua conoscenza è spesso esplicitamente richiesta, e quandanche così non fosse, è certamente consigliata perché propedeutica allo studio dei linguaggi moderni che da esso, sovente, derivano.

Le buone norme di programmazione. Le logiche dello sviluppo software a livello industriale pongono sfide organizzative e strategiche. È di norma previsto che allo stesso progetto contribuiscano diversi professionisti, con competenze e mansioni varie e complementari. È inoltre richiesto che il sorgente venga mantenuto e aggiornato nel tempo da team di programmatori, spesso, in continua evoluzione. Questo scenario

rende imprescindibile richiedere ai tecnici la scrittura di *codice di qualità*, che possa essere facilmente letto e compreso dai collaboratori. A tal fine è quindi ragionevole valutare un insieme di regole di buon senso, scelte in modo da garantire chiarezza laddove i costrutti del linguaggio potrebbero risultare sibillini. Tra queste buone norme di programmazione, particolare importanza riveste la scrittura di documentazione.

La documentazione funzionale. Posizione cardine nel ciclo di vita di un progetto è occupata dalla scrittura della documentazione. L'obiettivo di tale compendio al software è quello di delinearne i requisiti, le funzionalità e le interazioni tra i componenti, al fine di coadiuvare le fasi di sviluppo, testing, manutenzione ed evoluzione.

L'insegnamento *Laboratorio di Ingegneria Informatica*, erogato dal *Dipartimento di Ingegneria dell'Informazione*, pone come obiettivo l'approfondimento del linguaggio *C*, nella sua revisione del 1990[1], enfatizzando l'importanza della leggibilità e della manutenibilità del sorgente. Durante le lezioni vengono pertanto incoraggiate diverse buone norme di programmazione, e nel particolare la scrittura di documentazione funzionale. La stesura di quest'ultima si intende quale contestuale alla definizione di ogni funzione, e destinata a descriverne i parametri, in termini di utilità e destinazione d'uso.

La documentazione funzionale risulterà il principale oggetto di studio di tale tesi. Se ne darà una definizione formale, e si proporrà uno strumento di analisi statica capace di valutarne la correttezza all'interno di un codice sorgente *ANSI C*.

[1] Il linguaggio *C90*, o *ANSI C*, formalmente noto come *ISO/IEC 9899:1990*, rappresenta la prima standardizzazione del linguaggio di programmazione C, adottata dall'*International Organization for Standardization* (*ISO*) nel 1990. Questa versione ha codificato molti aspetti del linguaggio che erano precedentemente definiti in modo informale.

1.2 Implementazione

Analisi statica del codice sorgente. Con analisi statica si intende un insieme di tecniche e metodologie finalizzate all'ispezione di un codice sorgente senza che questo venga eseguito. Tale tipo di analisi viene comunemente condotta attraverso l'utilizzo di strumenti automatizzati.

Gli strumenti di analisi statica esaminano il codice sorgente ricercando pattern non desiderati, codice mal strutturato, bug potenziali e deviazioni rispetto alle norme stilistiche. L'adozione di queste tecniche di controllo permette allo sviluppatore di rilevare precocemente problematiche che potrebbero influenzare la qualità del codice sorgente, o causare errori in fase di esecuzione.

Le tecniche in questione possono risultare inoltre utili a garantire la conformità, e l'aderenza, del codice sorgente rispetto ad una classe di *buone norme di programmazione*, promuovendone leggibilità, manutenibilità e robustezza. L'applicativo proposto in questa tesi può pertanto essere classificato quale semplice strumento di analisi statica, destinato alla valutazione automatica della coerenza tra il codice sorgente, e la relativa documentazione funzionale.

Abstract syntax tree. L'*Abstract syntax tree* (*AST*) è una rappresentazione strutturata e gerarchica del codice sorgente di un software, dove ogni nodo rappresenta un costrutto del linguaggio di programmazione. Tale albero riveste un ruolo fondamentale nella progettazione di uno strumento di *analisi sintattica*, costituendo infatti un'astrazione rispetto alla sintassi lineare del codice sorgente e permettendone una valutazione efficace e sistematica.

Essendo l'*AST* un *albero*, e quindi un caso particolare di *grafo*, le analisi su di esso si riducono all'oculato utilizzo di un ristretto numero di *algoritmi di esplorazione*, conosciuti e ampiamente trattati in letteratura. L'applicativo trae infatti vantaggio dall'uso degli algoritmi di *esplorazione in profondità* (*depth-first search*, o *DFS*) e di *esplorazione in ampiezza* (*breadth-first search*, o *BFS*).

Lexer e Parser. La costruzione dell'*AST* si articola in due fasi distinte ma interconnesse: l'*analisi lessicale*, eseguita da un *Lexer*, e l'*analisi sintattica*, eseguita da un *Parser*.

Il *Lexer* ha il compito di suddividere il codice sorgente in una serie di *token*, ovvero ne riconosce ogni entità atomica e la classifica rispetto alle prescrizioni del linguaggio in uso[2]. Il *Lexer* elimina inoltre le parti di codice non utili alla costruzione dell'*AST*, generalmente commenti e caratteri di formattazione. Durante lo stadio dell'*analisi lessicale* è possibile individuare errori nel codice sorgente, quali l'utilizzo di caratteri non supportati e la presenza di *token* non riconosciuti.

Il *Parser*, successivamente, è responsabile della costruzione dell'*AST*. Identificando le relazioni strutturali tra i *token* secondo la grammatica del linguaggio scelto, il *Parser* è in grado di riconoscere i costrutti utilizzati, e di organizzarli gerarchicamente. Durante questo stadio è possibile verificare la correttezza sintattica del codice sorgente, e individuare la posizione degli eventuali *token* errati.

Lo strumento presentato nella tesi è proposto come supporto allo sviluppo con linguaggio *C90*. La progettazione del *Lexer*, e del *Parser*, verrà trattata approfonditamente, ma è importante sottolineare sia stata svolta seguendo pedissequamente le indicazioni del documento *ISO/IEC 9899:1990*, pubblicato dalla commissione tecnica *ISO/IEC JTC 1/SC 22*[3].

[2] Il *Lexer* proposto segue le regole lessicali del linguaggio *C90*. Alcuni *token* riconosciuti appartengono pertanto alle seguenti classi: *keyword*, *identifier*, *integer-constant*, *floating-constant*, *literal-constant*.

[3] La commissione *ISO/IEC JTC 1/SC 22*, dell'*International Organization for Standardization*, si occupa di linguaggi di programmazione, ambienti e interfacce di sistema.

Capitolo 2

Limitazioni

2.1 Caratteri utilizzabili

Aderendo rigorosamente allo standard *C90*, i caratteri utilizzabili nel codice sorgente dato in input all'applicativo, sono quelli contenuti nel *basic source character set*, che indicheremo con la lettera ∑. Introduciamo inoltre alcuni altri insiemi, che risulteranno utili durante la trattazione del *Lexer*.

```
∑L = { A B C D E F G H I J K L M N O P Q R S T U V W X Y Z }
∑l = { a b c d e f g h i j k l m n o p q r s t u v w x y z }
∑9 = { 0 1 2 3 4 5 6 7 8 9 }
∑★ = { ! " # % & ' ( ) * + , - . / : ; < = > ? [ \ ] ^ _ { | } ~ }
∑△ = { space \t \v \f \n \r }
∑ = ∑L ∪ ∑l ∪ ∑9 ∪ ∑★ ∪ ∑△
```

Si tenga presente che lo standard *C90* definisce ∑ quale insieme dei caratteri che devono necessariamente poter essere utilizzati nel codice sorgente, ma non vieta che anche altri simboli siano riconosciuti. Non deve quindi stupire che i compilatori moderni supportino ben oltre i 97 caratteri sopra citati, arrivando in generale a considerare per default l'insieme *UTF-8*.

2.2 Tipi di dato non nativi

L'applicativo richiede che i nomi dei tipi di dato non nativi, definiti tramite **`struct`** e **`union`**, utilizzino il prefisso "**`type_`**". È stata operata questa scelta al fine di semplificare il processo di *analisi sintattica* e viene trattata nel paragrafo *5.1.2*.

Si noti che tale limitazione non permetterebbe di utilizzare i tipi di dato non nativi definiti nelle librerie standard. È il caso, ad esempio, di **`FILE`**. In questo frangente, è però sufficiente e risolutivo rinominare, all'interno di un file di *header*, il tipo di dato non nativo in modo che rispetti la limitazione.

```
/* source.h */

typedef FILE type_FILE;

typedef struct user {
  char *name;
  char *surname;
} type_user;
```

2.3 Funzioni ricorsive

L'applicativo opera un controllo per verificare che il sorgente non contenga *funzioni ricorsive*, o *catene di funzioni ricorsive*. Tale limitazione è motivata dalla volontà di semplificare l'*analisi statica* del codice. Per una trattazione approfondita dell'argomento, si invita alla lettura del paragrafo *6.1.1.4*.

Capitolo 3

Librerie

3.1 Introduzione

Librerie generiche. Durante lo sviluppo dell'applicativo, si è data particolare attenzione alla scrittura di librerie di *strutture di dati*, e *algoritmi*, di taglio generico. Queste sono destinate a prestarsi quali blocchi fondamentali per lo sviluppo di *Lexer*, *Parser* e *Analyzer*.

Tale approccio è l'incarnazione dei principi di modularità e di riutilizzo del codice, paradigmi importanti nella progettazione di architetture software robuste ed estendibili. Tra le librerie, in ordine progressivo di complessità, si annoverano: *Buffer*, *List*, *HashMap*, *Graph*, *Deterministic Finite Automaton*.

Buffer. La libreria in questione implementa una *struttura di dati*, il *buffer*, che offre un meccanismo agile e sicuro per la manipolazione di stringhe di lunghezza variabile. Progettato per adattarsi automaticamente al volume di dati che deve contenere, tale *buffer* elimina la necessità di definire a priori la dimensione della stringa, scongiurando così il rischio di *overflow*. La capacità di scalare in modo trasparente rende il *buffer* uno strumento adatto alla gestione di input di dimensioni imprevedibili, garantendo al contempo un'occupazione ottimale della memoria.

List. Questa libreria offre uno strumento utile alla gestione delle collezioni di elementi. L'implementazione della struttura dati, chiamata *list*, prevede le operazioni

di inserimento, ricerca e cancellazione, senza vincoli dimensionali, e con complessità temporale al più lineare. Tali caratteristiche rendono talvolta l'utilizzo di *list* preferibile rispetto agli *array*, laddove non di possa conoscere a priori la cardinalità dell'insieme degli elementi trattati.

HashMap. La *struttura di dati HashMap* è stata progettata per offrire un'efficace soluzione alla necessità di accedere rapidamente a dati associativi. Attraverso l'utilizzo di una *funzione di hashing*, garantisce prestazioni significativamente migliori rispetto ad una ricerca lineare.

Graph. La libreria *Graph* fornisce una *struttura di dati* capace di rappresentare, in modo astratto e versatile, nodi e connessioni tra essi. Tale rappresentazione permette di modellare relazioni complesse e di eseguire algoritmi di esplorazione, di classificazione di nodi interni ed esterni, di analisi delle componenti connesse, di ricerca di percorsi ciclici. L'implementazione della libreria *Graph* enfatizza i concetti di generalità e di riutilizzabilità, e viene infatti utilizzata dall'applicativo nei più disparati contesti. Tratteremo in modo approfondito come *Lexer*, *Parser* e *Analyzer* traggano vantaggio da tale *struttura di dati*.

Deterministic Finite Automaton. La libreria *Deterministic Finite Automaton* (*DFA*) è responsabile dell'esecuzione delle operazioni di *pattern matching*. Tale *struttura di dati*, implementa un *automa a stati finiti*, *deterministico*, e con supporto per le *ε-transizioni*. Riveste importanza centrale durante la fase di *analisi lessicale*, dove garantisce il riconoscimento dei *token* in tempo lineare.

3.2 Strutture dati e algoritmi

Entriamo ora nel merito delle librerie presentate, illustrando le interfacce delle strutture dati relative, e citando le più importanti funzioni che ne implementano le logiche.

3.2.1 Libreria *Buffer*

```
typedef struct Buffer {
  char *value;
  unsigned long size;
  unsigned long length;
} Buffer;

Buffer  *buffer_create(void);
void    buffer_append_character(Buffer *buffer, char character);
void    buffer_append_integer(Buffer *buffer, int integer);
void    buffer_append_string(Buffer *buffer, const char *string);
void    buffer_free(Buffer *buffer);
void    buffer_reset(Buffer *buffer);

void    buffer_scale(Buffer *buffer, unsigned long length);
```

La libreria *Buffer* offre un'astrazione flessibile e dinamica per la manipolazione di stringhe di testo. La struttura *Buffer* è definita come un tipo di dato che contiene un puntatore ad una stringa (**`value`**), la dimensione allocata (**`size`**), e la lunghezza effettiva del contenuto (**`length`**). Le funzioni associate permettono di creare un nuovo *Buffer*, aggiungervi caratteri, interi e stringhe, resettarlo e liberare la memoria allocata. Di seguito una breve presentazione delle funzionalità:

- **`buffer_create`**: inizializza un nuovo *Buffer*;
- **`buffer_append_character`**: aggiunge un singolo carattere al *Buffer*;
- **`buffer_append_integer`**: aggiunge un numero intero al *Buffer*;
- **`buffer_append_string`**: aggiunge una stringa al *Buffer*;
- **`buffer_free`**: libera la memoria allocata dal *Buffer*;
- **`buffer_reset`**: inizializza il contenuto del *Buffer*.

La libreria fa inoltre uso, internamente, della seguente funzione:

- **`buffer_scale`**: quando necessario, raddoppia la dimensione del *Buffer*.

Per una comprensione completa della libreria, si consiglia la lettura del codice, che può essere trovato nei file **`lib_buffer.h`** e **`lib_buffer.c`**.

3.2.2 Libreria *List*

```
typedef struct ListNode {
  void* value;
  struct ListNode* next;
  struct ListNode* prev;
} ListNode;

typedef struct List {
  struct ListNode* first;
  struct ListNode* last;
  unsigned int length;
} List;

List*      list_copy(const List *list);
List*      list_create(int length, ...);
List*      list_create_empty();
ListNode*  list_get(const List *list, unsigned int index);
ListNode*  list_get_iterator(const List *list);
ListNode*  list_get_iterator_reverse(const List *list);
ListNode*  list_iterate(const ListNode *iterator);
ListNode*  list_iterate_reverse(const ListNode *iterator);
ListNode*  list_push_back(List *list, void *value);
ListNode*  list_push_front(List *list, void *value);
bool       list_is_empty(const List *list);
void*      list_get_last(const List *list);
void*      list_pop_back(List *list);
void*      list_pop_front(List *list);
void       list_free(List *list);
void       list_remove(List *list, ListNode *node);
```

La libreria *List* offre una struttura di dati per la gestione di collezioni dinamiche di elementi, con cardinalità non nota a priori. La struttura *ListNode* contiene un puntatore al valore immagazzinato (**`value`**), ed i puntatori al nodo precedente (**`prev`**) e successivo (**`next`**). La struttura *List* tiene invece traccia del primo

(**`first`**) e dell'ultimo (**`last`**) nodo della lista, nonché della lunghezza totale (**`length`**). Si fornisce di seguito una panoramica delle funzionalità offerte dalla libreria:

- **`list_copy`**: crea una copia di una *List* esistente;
- **`list_create`**: utilizza la funzionalità *varargs*, per inizializzare una *List* con un numero specificato di elementi iniziali;
- **`list_create_empty`**: inizializza una *List* vuota;
- **`list_get`**: recupera *ListNode* con un determinato indice;
- **`list_get_iterator`**: ottiene un iteratore che permetta di scorrere la *List* in avanti;
- **`list_get_iterator_reverse`**: ottiene un iteratore che permetta di scorrere la *List* indietro;
- **`list_iterate`**: muove l'iteratore al *ListNode* successivo;
- **`list_iterate_reverse`**: muove l'iteratore al *ListNode* precedente;
- **`list_push_back`**: aggiunge un elemento alla fine della *List*;
- **`list_push_front`**: aggiunge un elemento all'inizio della *List*;
- **`list_is_empty`**: verifica se la *List* è vuota;
- **`list_get_last`**: restituisce l'ultimo elemento della *List*;
- **`list_pop_back`**: rimuove e restituisce l'ultimo elemento della *List*;
- **`list_pop_front`**: rimuove e restituisce il primo elemento della *List*;
- **`list_free`**: libera la memoria associata alla *List*, ma non quella allocata dagli elementi all'interno di essa collezionati;
- **`list_remove`**: rimuove uno specifico *ListNode* dalla *List*.

Per una comprensione completa della libreria, si consiglia la lettura del codice, che può essere trovato nei file **`lib_list.h`** e **`lib_list.c`**.

3.2.3 Libreria *Hashmap*

```
typedef struct HashMap {
  List **lists;
  int size;
  int length;
} HashMap;

typedef struct HashMapNode {
  char *key;
  void *value;
} HashMapNode;

typedef struct HashMapIterator {
  char *key;
  void *value;
  const HashMap *hashmap;
  ListNode *list_node;
  int list_index;
} HashMapIterator;

HashMap*          hashmap_copy(const HashMap *hashmap);
HashMap*          hashmap_create();
HashMapIterator*  hashmap_get_iterator(const HashMap *hashmap);
HashMapIterator*  hashmap_iterate(HashMapIterator *iterator);
void  hashmap_free(HashMap *hashmap);
void* hashmap_get(const HashMap *hashmap, const char *key);
void* hashmap_set(HashMap *hashmap, const char *key, void *value);
```

La libreria *HashMap* predispone alcune *strutture di dati* utili a gestire coppie chiave-valore. La struttura *HashMap* contiene un puntatore ad un array di *List* (**`lists`**), un intero che indica la dimensione di tale array (**`size`**), ed un intero che rappresenta la cardinalità dell'insieme degli elementi contenuti (**`length`**). Ruolo molto importante riveste la struttura *HashMapIterator*, che permette di mantenere alcuni

utili riferimenti durante lo scorrimento della *HashMap*. Elenchiamo di seguito una lista delle logiche offerte dalla libreria:

- **`hashmap_copy`**: duplica una *HashMap*, mantenendo le associazioni chiave-valore;
- **`hashmap_create`**: inizializza una nuova *HashMap* vuota;
- **`hashmap_get_iterator`**: restituisce un *HashMapIterator* per navigare la *Hashmap*;
- **`hashmap_iterate`**: avanza l'*HashMapIterator* al successivo nodo, e ne libera la memoria associata ad iterazioni terminate;
- **`hashmap_free`**: rilascia la memoria impiegata dalla *HashMap*;
- **`hashmap_get`**: recupera il valore associato a una chiave;
- **`hashmap_set`**: assegna o aggiorna il valore relativo a una chiave.

La libreria fa inoltre uso, internamente, delle seguenti funzioni:

- **`hashmap_compute_key`**: funzione di *hashing* che, data una chiave testuale, vi associa l'indice di una delle *List* dell'*HashMap*;
- **`hashmap_get_next_list_index`**: dato un elemento dell'*HashMap*, restituisce l'indice della *List* contenente quello successivo;
- **`hashmap_get_previous_list_index`**: dato un elemento dell'*HashMap*, restituisce l'indice della *List* contenente quello precedente.

Per una comprensione completa della libreria, si consiglia la lettura del codice, che può essere trovato nei file **`lib_hashmap.h`** e **`lib_hashmap.c`**.

3.2.4 Libreria *Graph*

```
typedef struct GraphNode {
  char *identifier;
  HashMap *fields;
  List *children_edges;
  List *parent_edges;
} GraphNode;

typedef struct GraphEdge {
  GraphNode *child_node;
  GraphNode *parent_node;
  HashMap *fields;
} GraphEdge;

typedef struct Graph {
  HashMap *nodes;
} Graph;

Graph* graph_create();

GraphEdge* graph_add_edge(
  GraphNode *parent_node,
  GraphNode *child_node
);

GraphNode* graph_add_node(Graph *graph, const char *label);

void graph_copy_edges(GraphNode *node_a, const GraphNode *node_b);

void graph_remove_edge(GraphEdge *edge);

void graph_remove_edges(GraphNode *node);

void* graph_edge_get_field(
  const GraphEdge *edge,
  const char *key
);

void* graph_edge_set_field(
  GraphEdge *edge,
  const char *key,
  void *value
```

```
);

void* graph_node_get_field(
  const GraphNode *node,
  const char *key
);

void* graph_node_set_field(
  GraphNode *node,
  const char *key,
  void *value
);

GraphNode* graph_dfs_preorder_find_first(
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
);

List* graph_bfs(
  const Graph *graph,
  const List *roots
);

List* graph_dfs_postorder(
  const Graph *graph,
  const List *roots
);

List* graph_dfs_postorder_find_all(
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
);

List* graph_dfs_preorder_find(
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
);
```

```
List* graph_get_scc(const Graph *graph);

List* graph_get_self_loops(const Graph *graph);

void graph_free(Graph *graph);
```

La libreria *Graph* mette a disposizione uno strumento robusto per la modellazione, la manipolazione e l'analisi di grafi diretti. La struttura *GraphNode* rappresenta un nodo con identificatore univoco (**`identifier`**), una mappa di campi associativi (**`fields`**) e liste di archi entranti (**`parent_edges`**) e uscenti (**`children_edges`**). *GraphEdge* modella gli archi tra i nodi, e anch'esso contiene una mappa di campi associativi (**`fields`**). La struttura *Graph*, infine, contiene l'*HashMap* (**`nodes`**) destinata a immagazzinare ogni *GraphNode*, usando come chiave il relativo identificatore univoco.

Si noti che i *GraphEdge* di un *GraphNode* sono immagazzinati all'interno delle *List* **`children_edges`** e **`parent_edges`**. Tale caratteristica è importante, perché fa sì che tra questi venga mantenuto un ordinamento. Nello specifico, data l'implementazione della funzione **`graph_add_edge`**, l'ultimo *GraphEdge* aggiunto ad un *GraphNode* risulta essere l'ultimo elemento delle *List*. Questa proprietà è di fondamentale importanza, in particolare nell'ambito della costruzione dell'*AST*, in quanto questo risulta essere un *albero ordinato*.

Diamo ora una panoramica delle funzionalità della libreria:

- **`graph_create`**: inizializza un nuovo *Graph*;
- **`graph_add_edge`**: aggiunge un *GraphEdge* tra due *GraphNode*;
- **`graph_add_node`**: aggiuge un *GraphNode* al *Graph*;
- **`graph_copy_edges`**: copia i *GraphEdge* di un *GraphNode* su di un altro;
- **`graph_remove_edge`**: elimina un *GraphEdge*;
- **`graph_remove_edges`**: elimina ogni *GraphEdge* di un *GraphNode*;

- **`graph_edge_get_field`**: ricerca un valore all'interno della mappa associativa di un *GraphEdge*;
- **`graph_edge_set_field`**: imposta un valore all'interno della mappa associativa di un *GraphEdge*;
- **`graph_node_get_field`**: ricerca un valore all'interno della mappa associativa di un *GraphNode*;
- **`graph_node_set_field`**: imposta un valore all'interno della mappa associativa di un *GraphNode*;
- **`graph_dfs_preorder_find_first`**: funzione per esplorare il *Graph* con l'algoritmo *DFS pre-order*, restituendo il primo *GraphNode* che rispetti una condizione di equivalenza su di un **`field`** della mappa associativa;
- **`graph_bfs`**: funzione per esplorare il *Graph* con l'algoritmo *BFS*;
- **`graph_dfs_postorder`**: funzione per esplorare il *Graph* con l'algoritmo *DFS post-order*;
- **`graph_dfs_postorder_find_all`**: funzione per esplorare il *Graph* con l'algoritmo *DFS post-order*, restituendo ogni *GraphNode* che rispetti una condizione di equivalenza su di un **`field`** della mappa associativa;
- **`graph_dfs_preorder_find`**: funzione per esplorare il *Graph* con l'algoritmo *DFS pre-order*, restituendo ogni *GraphNode* che rispetti una condizione di equivalenza su di un **`field`** della mappa associativa, ed ignorando i relativi *GraphNode* figli;
- **`graph_get_scc`**: implementazione dell'algoritmo di *Kosaraju* per ottenere la lista delle *componenti fortemente connesse* (*strongly connected components*, o *SCC*) del *Graph*;

- **`graph_get_self_loops`**: funzione per ottenere la lista di *GraphNode* tali da essere figli di se stessi;
- **`graph_free`**: funzione per liberare la memoria associata a *Graph*.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`lib_graph.h`** e **`lib_graph.c`**. Si vuole tuttavia ribadire l'importanza delle funzioni di esplorazione e ricerca, approfondendo la logica degli algoritmi su cui esse si basano: *BFS*, *DFS pre-order* e *DFS post-order*. Si vuole inoltre brevemente trattare la definizione di *SCC*, e l'algoritmo di *Kosaraju*.

3.2.4.1 Algoritmo *BFS*

L'algoritmo di *esplorazione in ampiezza* (*breadth-first search*, o *BFS*) esplora sistematicamente un *grafo* partendo da un nodo radice e visitando tutti i suoi vicini, prima di procedere al livello successivo. Tale algoritmo ha quindi la caratteristica di visitare in successione tutti i nodi dello stesso livello, ovvero tutti i nodi con uguale distanza dalla radice. L'algoritmo *BFS* utilizza una *coda* per tenere traccia dei nodi da visitare, e ha complessità lineare rispetto al numero di nodi e archi del grafo.

Vengono di seguito proposti due esempi di funzionamento dell'algoritmo *BFS*, mostrando l'ordine di esplorazione dei nodi di un generico *grafo*, e di un *albero ordinato*.

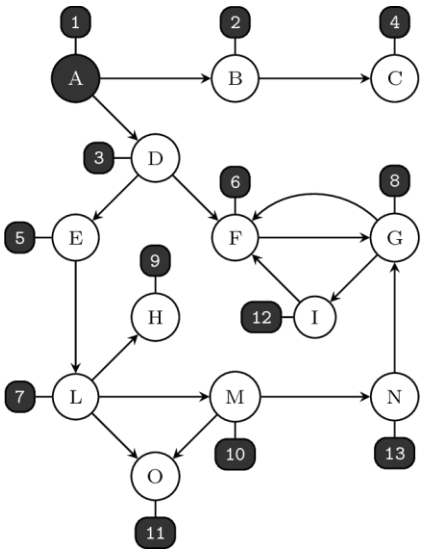

Figura 1, esempio di esecuzione dell'algoritmo *BFS* su di un *grafo*.

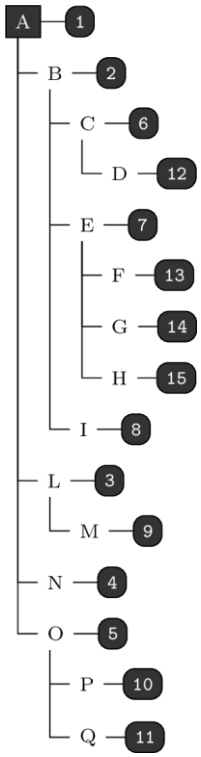

Figura 2, esempio di esecuzione dell'algoritmo *BFS* su di un *albero ordinato*.

3.2.4.2 Algoritmo *DFS pre-order*

L'algoritmo di *esplorazione in profondità in pre-ordine* (*depth-first search pre-order*, o *DFS pre-order*) esplora sistematicamente un *grafo* partendo da un nodo radice, per poi proseguire di nodo adiacente in nodo adiacente, fino al raggiungimento di un nodo foglia, per poi retrocedere ed esplorare le altre adiacenze. Tale algoritmo ha, quindi, la caratteristica di visitare in successione ogni nodo di uno stesso ramo prima di procedere al successivo. L'algoritmo *DFS pre-order* utilizza una *pila* per tenere traccia dei nodi da visitare, e ha complessità lineare rispetto al numero di nodi e archi del *grafo*.

L'algoritmo *DFS pre-order* trova applicazione nei seguenti contesti:

- *Clonazione di strutture ad albero* - durante la clonazione di una struttura ad *albero*, si procede prima alla copia di un nodo radice, e successivamente a quella dei nodi adiacenti;
- *Esportazione di strutture ad albero* - durante la serializzazione di un *albero* in forma lineare, come nell'esportazione in formati come *XML* o *JSON*, dove si inizia dall'elemento radice;
- *Ricerca di percorsi* - nell'ambito della ricerca di percorsi all'interno di un *grafo*, l'algoritmo *DFS pre-order* risulta utile perché traccia il percorso dalla radice fino al nodo corrente;
- *Elaborazione di alberi sintattici* - nell'*analisi sintattica*, risulta utile quando è necessario processare un nodo prima dei relativi figli.

Vengono di seguito proposti due esempi di funzionamento dell'algoritmo *DFS pre-order*, mostrando l'ordine di esplorazione dei nodi di un generico *grafo*, e di un *albero ordinato*.

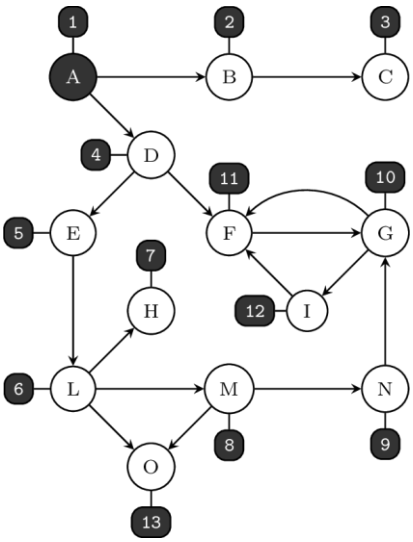

Figura 3, esempio di esecuzione dell'algoritmo *DFS pre-order* su di un *grafo*.

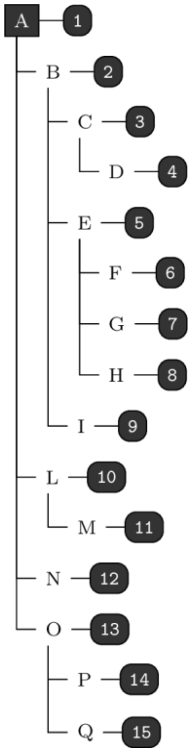

Figura 4, esempio di esecuzione dell'algoritmo *DFS pre-order* su di un *albero ordinato*.

3.2.4.3 Algoritmo *DFS post-order*

L'algoritmo di *esplorazione in profondità in post-ordine* (*depth-first search post-order*, o *DFS post-order*) esplora sistematicamente un *grafo*, iniziando dalle foglie e procedendo verso il nodo radice, assicurandosi di visitare tutti i nodi di un ramo prima di risalire al nodo genitore. Tale algoritmo ha quindi la caratteristica di elaborare ogni nodo solo dopo che ne sono stati visitati tutti i discendenti. L'algoritmo *DFS post-order* utilizza una pila per tenere traccia dei nodi da visitare, e ha complessità lineare rispetto al numero di nodi e archi del *grafo*.

L'algoritmo *DFS post-order* trova applicazione nei seguenti contesti:

- *Calcolo di proprietà cumulative* - l'algoritmo *DFS post-order* risulta utile quando è necessario considerare il valore dei nodi figli per la computazione dei nodi genitori;
- *Liberazione in memoria di strutture ad albero* - l'algoritmo assicura che la liberazione della memoria associata ai nodi figli avvenga prima di quella del nodo genitore;
- *Ordinamento topologico* - l'algoritmo permette di realizzare un ordinamento degli elementi del *grafo* per cui ogni nodo preceda tutti quelli ai quali punta direttamente;
- *Elaborazione di alberi sintattici* - nell'*analisi sintattica*, risulta utile quando è necessario processare prima i nodi figli e successivamente il genitore, come nel caso della valutazione dell'ordine di esecuzione delle istruzioni in un *AST*.

Vengono di seguito proposti due esempi di funzionamento dell'algoritmo *DFS post-order*, mostrando l'ordine di esplorazione dei nodi di un generico *grafo*, e di un *albero ordinato*.

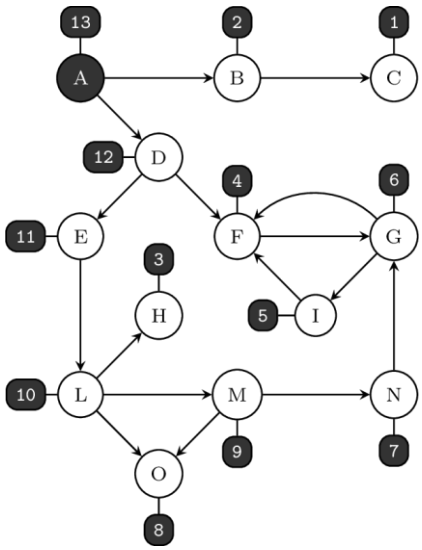

Figura 5, esempio di esecuzione dell'algoritmo *DFS post-order* su di un *grafo*.

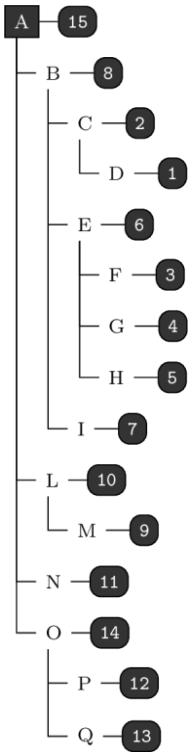

Figura 6, esempio di esecuzione dell'algoritmo *DFS post-order* su di un *albero ordinato*.

3.2.4.4 *SCC* e algoritmo di *Kosaraju*

Dato un *grafo diretto*, definiamo *componente strettamente connessa* (*strongly connected component*, o *SCC*) un sottoinsieme di nodi tali che ognuno di questi sia raggiungibile da ogni altro nodo all'interno della *SCC* stessa.

L'algoritmo di *Kosaraju* è un metodo efficiente per trovare tutte le componenti fortemente connesse in un *grafo diretto*. Si articola in due fasi principali:

- *DFS pre-order* - esplorazione del grafo mediante l'algoritmo *DFS pre-order*;
- *Reverse DFS pre-order* - esplorazione del *grafo trasposto*[4] mediante l'algoritmo *DFS pre-order*, utilizzando come radici i nodi del grafo disposti secondo l'ordine calcolato durante la prima fase.

Tale processo permette di identificare ogni *componente fortemente connessa*, ed opera in tempo lineare rispetto al numero di nodi e archi del *grafo*.

Viene di seguito proposto un esempio di funzionamento dell'algoritmo di *Kosaraju*; vengono nello specifico evidenziate le *componenti connesse* del *grafo*, che l'algoritmo è in grado di individuare.

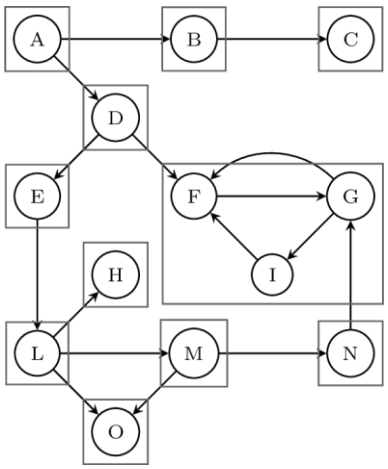

Figura 7, esempio di esecuzione dell'algoritmo di *Kosaraju* su di un *grafo*.

[4] Con *trasposizione* di un *grafo diretto*, si intende l'operazione di inversione di ogni arco.

3.2.5 Libreria *DFA*

```
typedef struct DFA {
  Graph *graph;
  GraphNode *state;
  FILE *source;
  int character;
  int line;
  int column;
} DFA;

typedef struct DFAMatch {
  int type;
  char *value;
  int line;
  int column;
} DFAMatch;

typedef GraphNode DFAState;
typedef List DFAStateGroup;

DFA *dfa_create(FILE *source);

DFAMatch *dfa_get_next_match(DFA *dfa);

DFAState *dfa_add_root(const DFA *dfa, const char *label);

DFAState *dfa_add_state(const DFA *dfa, const char *label);

DFAState *dfa_add_state_epsilon(
  const DFA *dfa,
  GraphNode *state,
  const char *prefix
);

DFAState *dfa_add_state_group_epsilon(
  const DFA *dfa,
  DFAStateGroup *state,
  const char *prefix
);

DFAStateGroup *dfa_add_state_group(
  const DFA *dfa,
```

```
  unsigned int count,
  const char *prefix
);

int *dfa_string_to_transitions(const char *string);

void dfa_epsilon_transition_state_group_to_state(
  List *states_a,
  GraphNode *state_b
);

void dfa_epsilon_transition_state_to_state(
  GraphNode *state a,
  GraphNode *state_b
);

void dfa_free(DFA *dfa);

void dfa_match_free(DFAMatch *match);

void dfa_set_leaf(const GraphNode *state, int token);

void dfa_set_leaves(const List *states, int token);

void dfa_transition_state_group_to_state(
  const DFAStateGroup *state_a,
  DFAState *state_b,
  int transition_value
);

void dfa_transition_state_group_to_state_group(
  const DFAStateGroup *state_a,
  DFAStateGroup *state_b,
  int transitions_count,
  int *transition_values
);

void dfa_transition_state_to_state(
  DFAState *state_a,
  DFAState *state_b,
  int transition_value
);

void dfa_transition_state_to_state_group(
```

```
  DFAState *state_a,
  const DFAStateGroup *states_b,
  int transitions_count,
  int *transition_values
);
```

La libreria *DFA* implementa un *automa a stati finiti* (*deterministic finite automaton*, o *DFA*), con supporto per le *ε-transizioni*, ovvero un modello computazione capace di riconoscere pattern o configurazioni di simboli. A differenza di un *DFA standard*, che richiede che per ogni stato siano definite transizioni per ogni possibile simbolo dell'alfabeto utilizzato, il *DFA* con supporto per le *ε-transizioni* permette di modellare transizioni che non consumino input. Tale caratteristica facilita la costruzione e l'ottimizzazione degli automi.

La struttura *DFA* rappresenta l'*automa a stati finiti*, incorporando un *Graph* (**`graph`**) che modella gli stati e le transizioni tra essi. Questa struttura tiene inoltre traccia dell'attuale stato dell'*automa* (**`state`**), di un *FILE* di input associato e di valori utili all'esecuzione dell'algoritmo di *pattern-matching* (**`character`**, **`line`**, **`column`**). La struttura *DFAMatch*, invece, è utilizzata per restituire le informazioni relative ad un *pattern* riconosciuto, quali il tipo di *token* (**`type`**), il contenuto (**`value`**) e la posizione (**`line`**, **`column`**).

Rivestono un importante ruolo, inoltre, *DFAState* e *DFAStateGroup*, che non sono altro che *alias* rispettivamente di *GraphNode* e *List*. *DFAStateGroup*, nello specifico, è utile per rappresentare e manipolare gruppi di stati con caratteristiche simili.

Presentiamo brevemente le funzionalità della libreria:

- **`dfa_create`**: inizializza il *DFA*;
- **`dfa_get_next_match`**: restituisce il *DFAMatch* successivo trovato nell'input;
- **`dfa_add_root`**: aggiunge uno stato al *DFA*, e lo etichetta quale radice;
- **`dfa_add_state`**: aggiunge uno stato al *DFA*;

- **`dfa_add_state_epsilon`**: aggiunge uno stato al *DFA*, e lo collega tramite *ε-transition* ad un altro stato dato;
- **`dfa_add_state_group_epsilon`**: aggiunge uno stato al *DFA*, e lo collega tramite *ε-transition* ad un gruppo di stati;
- **`dfa_add_state_group`**: aggiunge un gruppo di stati al *DFA*;
- **`dfa_string_to_transitions`**: funzione di supporto, utile durante la creazione di *automi a stati finiti*, che data una stringa, restituisce un *array* di interi contenente il codice *ASCII* di ogni carattere;
- **`dfa_epsilon_transition_state_group_to_state`**: aggiunge una *ε-transition* tra un gruppo di stati ed uno stato;
- **`dfa_epsilon_transition_state_to_state`**: aggiunge una *ε-transition* tra due stati;
- **`dfa_free`**: libera la memoria associata al *DFA*;
- **`dfa_match_free`**: libera la memoria associata al *DFAMatch*;
- **`dfa_set_leaf`**: etichetta uno stato quale foglia, specificandone il relativo **`type`**;
- **`dfa_set_leaves`**: etichetta un gruppo di stati quale foglia, specificandone il relativo **`type`**;
- **`dfa_transition_state_group_to_state`**: aggiunge una transizione tra un gruppo di stati ed uno stato;
- **`dfa_transition_state_group_to_state_group`**: aggiunge una transizione tra due gruppi di stati;
- **`dfa_transition_state_to_state`**: aggiunge una transizione tra due stati;

- **`dfa_transition_state_to_state_group`**: aggiunge una transizione tra uno stato ed un gruppo di stati.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`lib_dfa.h`** e **`lib_dfa.c`**.

3.2.5.1 Algoritmo di *pattern-matching*

L'algoritmo di *pattern-matching* è di cardinale importanza teorica per comprendere il funzionamento della libreria. È implementato dalla funzione **`dfa_get_next_match`**, della quale andremo ora ad analizzare le logiche:

- viene inizializzato un *Buffer*, destinato ad accumulare i caratteri di che comporranno il contenuto di un potenziale *DFAMatch*;
- vengono letti i caratteri dal flusso di input, uno alla volta, fintantoché non viene raggiunto il termine del file (**`EOF`**) ed un *DFAState* etichettato quale radice;
- qualora il *DFAState* attuale sia etichettato quale radice, viene resettato il contenuto del *Buffer*;
- utilizzando il carattere ottenuto dal flusso di input, viene cercata una corrispondente transizione tra quelle del *DFAState* attuale; se trovata viene aggiunto il carattere al *Buffer*, viene aggiornato il *DFAState*, e si procede al successivo;
- se non viene trovata alcuna valida transizione, viene cercata una *ε-transition* tra le transizioni del *DFAState* attuale; se trovata non viene aggiunto alcun carattere al *Buffer*, viene aggiornato il *DFAState*, e si procede al successivo;
- se non viene trovata alcuna valida *ε-transition*, viene emesso un avviso e consumato il carattere senza che questo venga aggiunto al *Buffer*;

- viene ora controllato se il *DFAState* raggiunto rappresenti un *match*, in tal caso viene creato e restituito un *DFAMatch*;
- se non viene trovato alcun *DFAMatch*, pur avendo raggiunto la fine del *FILE* ed un *DFAState* etichettato quale radice, allora viene restituito `NULL`, indicando l'assenza di qualsivoglia altro match.

3.2.5.2 Radici multiple

Si noti che la libreria *DFA* permette la creazione di più *DFAState* etichettati quali radici. Nell'ambito degli *automi a stati finiti deterministici* tale caratteristica non è tipica, poiché contraddice la definizione di unico stato iniziale. In un'implementazione estesa, tuttavia, questa peculiarità permette di supportare scenari complessi, rendendo possibile la commutazione tra differenti logiche di *pattern matching*, in risposta a specifici input o condizioni.

Il *Lexer* deve gestire, oltre alle logiche di *pattern-matching* dei *token* caratterizzanti del *C90*, le logiche di *pattern-matching* dei *token* definiti per redigere la documentazione funzionale. A tal fine, viene implementato un automa a *stati finiti deterministico* che arriva a utilizzare quattro radici.

Capitolo 4

Lexer

4.1 Introduzione

Il *Lexer* è il modulo dell'applicativo capace di eseguire l'*analisi lessicale* di sorgenti *C90*. Facendo uso della libreria *DFA*, il *Lexer* costruisce un *automa a stati finiti deterministico*, modellato secondo le specifiche del documento *ISO/IEC 9899:1990*. La libreria è organizzata modularmente, in modo da costruire il *DFA* per progressiva integrazione di sotto-moduli specializzati: *keyword*, *identifier*, *symbol*, *numerical*, *literal*, *type_identifier*, *skip* e *comment*.

Il *Lexer* si occupa inoltre del *pattern-matching* dei *token* definiti per supportare la scrittura della documentazione funzionale, mediante i sotto-moduli: *comment*, *v_doc*, *v_doc_rule*. A tal fine, si è beneficiato della capacità della libreria *DFA* di gestione di più *DFAState* etichettati quali radice.

Verrà analizzato ogni sotto-modulo, presentando i *token* che questo permette di riconoscere, ed una rappresentazione in forma di *grafo* contenente la porzione di *automa a stati finiti* che questo integra. Per ogni *token*, verrà allegata la definizione formale specificata dal documento *ISO*.

La lettura del sorgente dei sotto-moduli è utile ma non necessaria, trattandosi infatti di una pedissequa traduzione in termini di codice delle rappresentazioni allegate.

Forma Backus-Naur. Il documento *ISO/IEC 9899:1990* specifica formalmente ogni token utilizzando la notazione *Backus-Naur* (*Backus-Naur form*, o *BNF*). La notazione *Backus-Naur* permette di descrivere la sintassi di linguaggi formali. Consiste in *regole di produzione* che vedono un lato sinistro, ed un lato destro. Il lato sinistro rappresenta un simbolo non terminale. Il lato destro, invece, descrive quale forma possano assumere le stringhe generate da tale simbolo, in termini di altri simboli terminali e non terminali. I simboli terminali corrispondono agli elementi costitutivi del linguaggio, che possono effettivamente essere riconosciuti tramite *pattern-matching*.

La notazione *BNF* viene ampiamente utilizzata nella documentazione, e progettazione, di linguaggi di programmazione e protocolli di comunicazione. Il *Parser* stesso ne farà largo uso per implementare le logiche di costruzione dell'*AST*.

Simboli terminali. La specifica *ISO/IEC 9899:1990*, nel contesto della definizione dei *token* relativi a identificatori, costanti numeriche e letterali, utilizza quali simboli terminali i singoli caratteri alfanumerici. Il *Lexer* permette, invece, di eseguire il *pattern-matching* dell'intero identificatore, o costante, operando così una ridefinizione dei simboli terminali. Tale approccio semplifica l'analisi successiva eseguita dal *Parser*, riducendo il numero di elementi da processare, e fornendo unità di significato più grandi, che meglio corrispondano alle costruzioni del linguaggio di programmazione. In ultima analisi, questo permette di generare un *AST* più leggibile e più facilmente correlabile al codice sorgente originale.

Rappresentazioni grafiche. Al fine di interpretare correttamente le rappresentazioni grafiche dell'*automa a stati finiti*, è importante familiarizzare con la simbologia adottata. I *DFAState* sono indicati da nodi circolari, che simboleggiano gli stati attraverso i quali l'automa transita. I *DFAStateGroup*, invece, sono indicati da nodi rettangolari, e rappresentano insiemi di stati che condividono determinate proprietà o transizioni. Gli stati etichettati quali radici, o quali match, sono distinti da uno sfondo scuro, a indicare il loro ruolo chiave nell'iniziare o concludere il processo di *pattern-matching*.

Nelle rappresentazioni sono talvolta inoltre presenti archi entranti o uscenti dai nodi, e distinti da particolari simboli. Vengono in questo modo indicate delle connessioni a zone dell'*automa a stati finiti* non facilmente raggiungibili graficamente con un arco, ed eventualmente presenti in altre rappresentazioni.

Al fine di agevolare la costruzione delle rappresentazioni grafiche, andiamo a definire i seguenti insiemi di caratteri:

```
Σ7     = { 0 1 2 3 4 5 6 7 }
Σ15    = { 0 1 2 3 4 5 6 7 8 9 a b c d e f A B C D E F }
Δ      = { A B C D E F G H I J K M N O P Q R S T U V W X Y Z
           h j k m n o p q x y z _ }
```

Nomenclatura dei *token*. I nomi dei *token* sono stati ispirati dalla nomenclatura proposta dallo sviluppatore Jeff Lee, in un documento di configurazione di *Lex*, un *lexical analyzer generator*. Tale documento di configurazione, pubblicato nel 1985, è stato sviluppato secondo le direttive contenute in una bozza della specifica del linguaggio *ANSI C*.

4.2 *Pattern-matching* dei *token* del codice sorgente

4.2.1 Sotto-modulo *keyword*

```
<keyword> ::= `auto`     | `double` | `int`      | `struct`
            | `break`    | `else`   | `long`     | `switch`
            | `case`     | `enum`   | `register` | `typedef`
            | `char`     | `extern` | `return`   | `union`
            | `const`    | `float`  | `short`    | `unsigned`
            | `continue` | `for`    | `signed`   | `void`
            | `default`  | `goto`   | `sizeof`   | `volatile`
```

```
            | `do`       | `if`     | `static`   | `while`
            ;
```

***Token*.** Il sotto-modulo *keyword*, del *Lexer*, implementa il *pattern matching* dei *token* **AUTO**, **DOUBLE**, **INT**, **STRUCT**, **BREAK**, **ELSE**, **LONG**, **SWITCH**, **CASE**, **ENUM**, **REGISTER**, **TYPEDEF**, **CHAR**, **EXTERN**, **RETURN**, **UNION**, **CONST**, **FLOAT**, **SHORT**, **UNSIGNED**, **CONTINUE**, **FOR**, **SIGNED**, **VOID**, **DEFAULT**, **GOTO**, **SIZEOF**, **VOLATILE**, **DO**, **IF**, **STATIC**, **WHILE**.

Collegamenti esterni. Dalla rappresentazione grafica si può notare che ogni *DFAState* presenti un collegamento esterno uscente che punta al sotto-modulo *Identifier*. Unica eccezione viene fatta per il *DFAState* 113, che presenta un collegamento esterno uscente verso il sotto-modulo *type-identifier*.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 14**, ed il relativo codice sorgente può essere consultato nei file **`mod_lexer_dfa_c90_keyword.h`** e **`mod_lexer_dfa_c90_keyword.c`**.

4.2.2 Sotto-modulo *identifier*

```
<identifier> ::= <nondigit>
               | <identifier> <nondigit>
               | <identifier> <digit>
               ;
<nondigit> ::= `a` | `b` | `c` | `d` | `e` | `f` | `g`
             | `h` | `i` | `j` | `k` | `l` | `m`
             | `n` | `o` | `p` | `q` | `r` | `s` | `t`
             | `u` | `v` | `w` | `x` | `y` | `z`
             | `A` | `B` | `C` | `D` | `E` | `F` | `G`
             | `H` | `I` | `J` | `K` | `L` | `M`
             | `N` | `O` | `P` | `Q` | `R` | `S` | `T`
             | `U` | `V` | `W` | `X` | `Y` | `Z`
             | `_`
             ;
<digit> ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
```

Token. Il sotto-modulo *identifier*, del *Lexer*, è responsabile del *pattern-matching* del *token* **IDENTIFIER**.

Collegamenti esterni. Dalla rappresentazione grafica si può notare la presenza di collegamenti esterni entranti, dai sotto-moduli *keyword* e *numerical*.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 8**, ed il relativo codice sorgente può essere consultato nei file **`mod_lexer_dfa_c90_identifier.h`** e **`mod_lexer_dfa_c90_identifier.c`**.

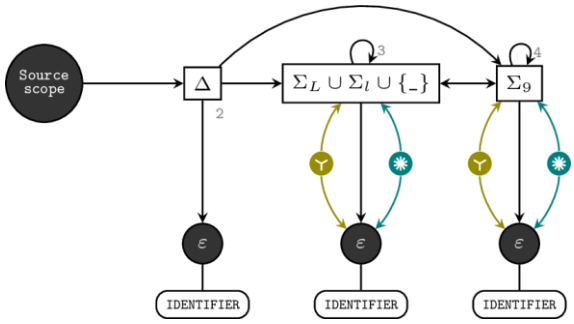

Figura 8, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *identifier*.

4.2.3 Sotto-modulo *symbol*

```
<operator> ::= `[`   | `]`  | `(`  | `)`  | `.`  | `->`
             | `++`  | `--` | `&`  | `*`  | `+`  | `-`  | `~`   | `!`  | `sizeof`
             | `/`   | `%`  | `<<` | `>>` | `<`  | `>`  | `<=`  | `>=`
             | `==`  | `!=` | `^`  | `|`  | `&&` | `||` | `?`   | `:`
             | `=`   | `*=` | `/=` | `%=` | `+=` | `-=` | `<<=`
             | `>>=` | `&=` | `^=` | `|=` | `,`
             ;

<punctuator> ::= `[` | `]` | `(` | `)` | `{` | `}`
               | `*` | `,` | `:` | `=` | `;` | `...`
               ;
```

Token. Il sotto-modulo *symbol*, del *Lexer*, è responsabile del *pattern-matching* del *token* **`SO_BRACKET`**, **`SC_BRACKET`**, **`RO_BRACKET`**, **`RC_BRACKET`**, **`DEC_OP`**, **`PTR_OP`**, **`SUB_ASSIGN`**, **`INC_OP`**, **`ADD_ASSIGN`**, **`AND_OP`**, **`AND_ASSIGN`**, **`MUL_ASSIGN`**, **`BITWISE_NOT_OP`**, **`NE_OP`**, **`COLON`**, **`TERNARY_OP`**, **`CO_BRACKET`**, **`CC_BRACKET`**, **`SEMICOLON`**, **`DIV_ASSIGN`**, **`OPEN_COMMENT`**, **`MOD_ASSIGN`**, **`LEFT_ASSIGN`**, **`LE_OP`**, **`RIGHT_ASSIGN`**, **`GE_OP`**, **`EQ_OP`**, **`XOR_ASSIGN`**, **`OR_OP`**, **`OR_ASSIGN`**, **`COMMA`**, **`ELLIPSIS`**,

SUB_OP, **ADD_OP**, **AMPERSAND**, **ASTERISK**, **LOGICAL_NOT_OP**, **DIV_OP**, **MOD_OP**, **L_OP**, **LEFT_OP**, **G_OP**, **RIGHT_OP**, **EQUAL**, **BITWISE_XOR_OP**, **BITWISE_OR_OP**, **DOT_OP**.

Commutazioni. Dalla rappresentazione è possibile evincere come il *Lexer* tragga effettivamente vantaggio dalla capacità della libreria *DFA* di supportare più radici. Tale porzione di *automa a stati finiti* permette infatti la commutazione dalla radice **Source scope**, verso la radice **Comment scope**.

Collegamenti esterni. Dalla rappresentazione grafica, si può notare la presenza di collegamenti esterni uscenti verso il sotto-modulo *numerical*.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 15**, ed il relativo codice sorgente può essere consultato nei file **mod_lexer_dfa_c90_symbol.h** e **mod_lexer_dfa_c90_symbol.c**.

4.2.4 Sotto-modulo *numerical*

```
<floating-constant> ::= <fractional-constant>
                      | <fractional-constant> <exponent-part>
                      | <fractional-constant> <floating-suffix>
                      | <fractional-constant> <exponent-part> <floating-suffix>
                      ;
<fractional-contant> ::= `.` <digit_sequence>
                       | <digit_sequence> `.` <digit_sequence>
                       | <digit_sequence>
                       ;
<exponent_part> ::= `e` <digit_sequence>
                  | `e` <sign> <digit_sequence>
                  | `E` <digit_sequence>
                  | `E` <sign> <digit_sequence>
                  ;
<sign> ::= `+`| `-`
<digit_sequence> ::= <digit> | <digit_sequence> <digit>
<floating_suffix> ::= `f` | `l` | `F` | `L`

<integer-constant> ::= <decimal-constant>
                     | <decimal-constant> <integer-suffix>
                     | <octal-constant>
```

```
                    | <octal-constant> <integer-suffix>
                    | <hexadecimal-constant>
                    | <hexadecimal-constant> <integer-suffix>
                    ;
<decimal-constant> ::= <nonzero-digit>
                    | <decimal-constant> <digit>
                    ;
<octal-constant> ::= `0`
                   | <octal-constant> <octal-digit>
                   ;
<hexadecimal-contant> ::= `0x` <hexadecimal-digit>
                        | `0X` <hexadecimal-digit>
                        | <hexadecimal-constant> <hexadecimal-digit>
                        ;
<nonzero-digit> ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
<octal-digit> ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
<hexadecimal-digit> ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
                      | `a` | `b` | `c` | `d` | `e` | `f`
                      | `A` | `B` | `C` | `D` | `E` | `F`
                      ;
<integer-suffix> ::= <unsigned-suffix>
                   | <long-suffix>
                   | <unsigned-suffix> <long-suffix>
                   | <long-suffix> <unsigned-suffix>
                   ;
<unsigned-suffix> ::= `u`| `U`
<long-suffix> ::= `l` | `L`
```

Token. Il sotto-modulo *numerical*, del *Lexer*, è responsabile del *pattern-matching* delle costanti numeriche, rappresentate dal *token* **`IDENTIFIER`**.

Collegamenti esterni. Dalla rappresentazione grafica, si può notare la presenza di collegamenti esterni entranti, dal sotto-modulo *symbol*.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 9**, ed il relativo codice sorgente può essere consultato nei file **`mod_lexer_dfa_c90_numerical.h`** e **`mod_lexer_dfa_c90_numerical.c`**.

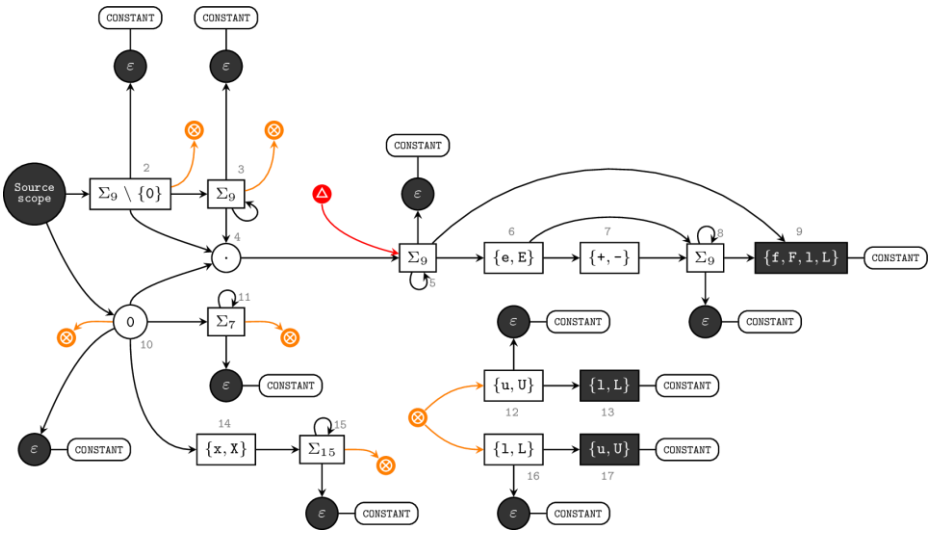

Figura 9, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *numerical*.

4.2.5 Sotto-modulo *literal*

```
<character-constant> ::= `'` <c-char-sequence> `'`
                       | `L` `'` <c-char-sequence> `'`
                       ;
<c-char-sequence> ::= <c-char>
                    | <c-char-sequence> <c-char>
                    ;
<c-char> ::= `A` | `B` | `C` | `D` | `E` | `F` | `G`
           | `H` | `I` | `J` | `K` | `L` | `M`
           | `N` | `O` | `P` | `Q` | `R` | `S` | `T`
           | `U` | `V` | `W` | `X` | `Y` | `Z`
           | `a` | `b` | `c` | `d` | `e` | `f` | `g`
           | `h` | `i` | `j` | `k` | `l` | `m`
           | `n` | `o` | `p` | `q` | `r` | `s` | `t`
           | `u` | `v` | `w` | `x` | `y` | `z`
           | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
           | `!` | `"` | `#` | `%` | `&` | `(` | `)` | `*` | `+`
           | `,` | `-` | `.` | `/` | `:` | `;` | `<` | `=` | `>`
           | `?` | `[` | `]` | `^` | `_` | `{` | `|` | `}` | `~`
           | `space` | `\t` | `\v` | `\f`
           | <escape-sequence>
           ;
<escape-sequence> ::= <simple-escape-sequence>
                    | <octal-escape-sequence>
```

```
                    | <hexadecimal-escape-sequence>
                    ;
<simple-escape-sequence> ::= `\` `'` | `\` `"` | `\` `?` | `\` `\`
                           | `\` `a` | `\` `b` | `\` `f` | `\` `n`
                           | `\` `r` | `\` `t` | `\` `v`
                           ;
<octal-escape-sequence> ::= `\` <octal-digit>
                          | `\` <octal-digit> <octal-digit>
                          | `\` <octal-digit> <octal-digit> <octal-digit>
                          ;
<hexadecimal-escape-sequence> ::= `\x` <hexadecimal-digit>
                                | <hexadecimal-escape-sequence> <hexadecimal-digit>
                                ;

<string-literal> ::= `"` <s-char-sequence> `"`
                   | `L` `"` <s-char-sequence> `"`
                   ;
<s-char-sequence> ::= <s-char>
                    | <s-char-sequence> <s-char>
                    ;
<s-char> ::= `A` | `B` | `C` | `D` | `E` | `F` | `G`
           | `H` | `I` | `J` | `K` | `L` | `M`
           | `N` | `O` | `P` | `Q` | `R` | `S` | `T`
           | `U` | `V` | `W` | `X` | `Y` | `Z`
           | `a` | `b` | `c` | `d` | `e` | `f` | `g`
           | `h` | `i` | `j` | `k` | `l` | `m`
           | `n` | `o` | `p` | `q` | `r` | `s` | `t`
           | `u` | `v` | `w` | `x` | `y` | `z`
           | `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
           | `!` | `'` | `#` | `%` | `&` | `(` | `)` | `*` | `+`
           | `,` | `-` | `.` | `/` | `:` | `;` | `<` | `=` | `>`
           | `?` | `[` | `]` | `^` | `_` | `{` | `|` | `}` | `~`
           | `space` | `\t` | `\v` | `\f`
           | <escape-sequence>
           ;
```

Token. Il sotto-modulo *literal*, del *Lexer*, è responsabile del *pattern-matching* dei caratteri, rappresentati dal *token* **`IDENTIFIER`**, e delle stringhe, rappresentate dal *token* **`STRING_LITERAL`**.

Collegamenti esterni. Dalla rappresentazione grafica, si può notare la presenza di un collegamento esterno uscente, verso il sotto-modulo *identifier*.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 10**, ed il relativo codice sorgente può essere consultato nei file `mod_lexer_dfa_c90_literal.h` e `mod_lexer_dfa_c90_literal.c`.

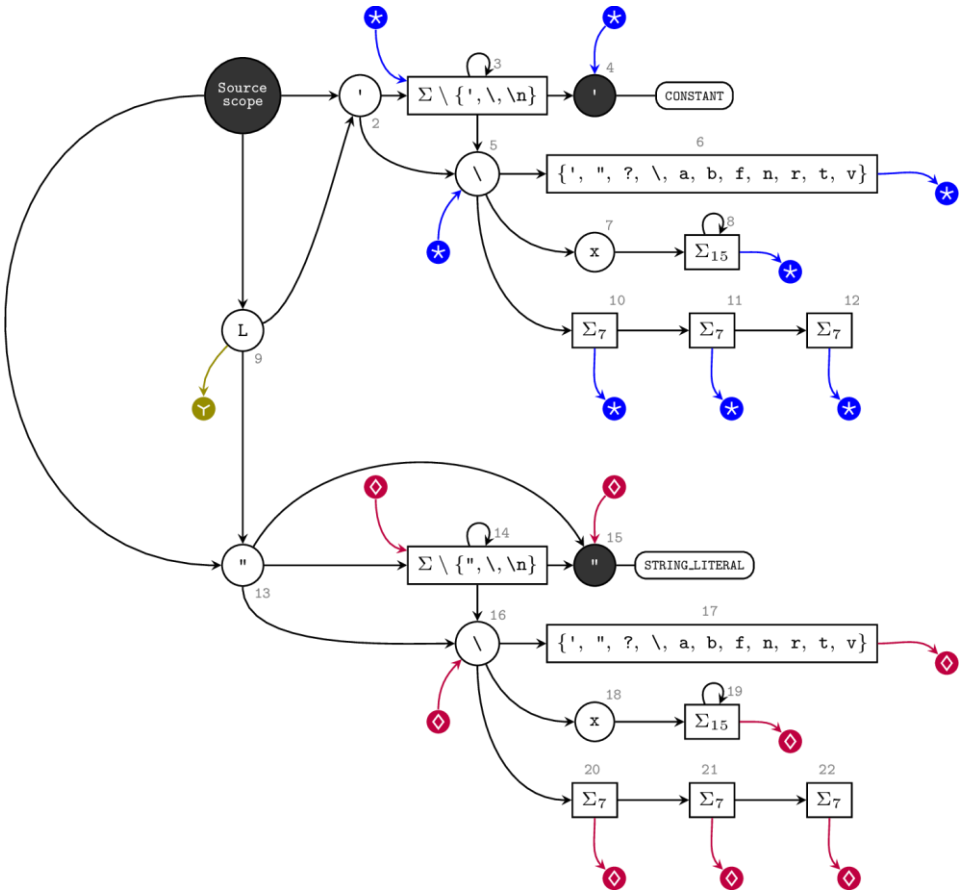

Figura 10, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *literal*.

4.2.6 Sotto-modulo *type_identifier*

```
<type_identifier> ::= `type_` <identifier>
```

Token. Il sotto-modulo *type_identifier*, del *Lexer*, è responsabile del *pattern-matching* dei caratteri, rappresentati dal *token* `TYPE_IDENTIFIER`. Tale *token* non risulta specificato nel documento *ISO/IEC 9899:1990*, è stato introdotto per semplificare il processo di *analisi sintattica*.

Collegamenti esterni. Dalla rappresentazione grafica, si può notare la presenza di un collegamento esterno entrante dal sotto-modulo *keyword*.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 11**, ed il relativo codice sorgente può essere consultato nei file `mod_lexer_dfa_c90_type_identifier.h` e `mod_lexer_dfa_c90_type_identifier.c`.

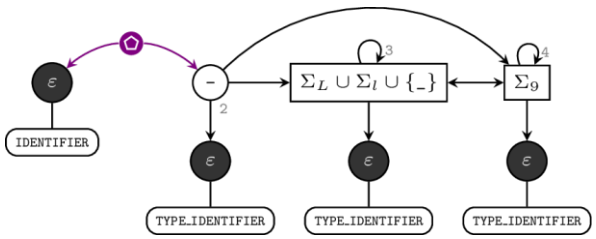

Figura 11, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *type_identifier*.

4.2.7 Sotto-modulo *skip*

Il sotto-modulo *skip* permette al Lexer di ignorare porzioni del codice sorgente non utili al *Parser*, nello specifico simboli di formattazione, quali gli spazi, e istruzioni per il preprocessore.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 12**, ed il relativo codice sorgente può essere consultato nei file `mod_lexer_dfa_c90_skip.h` e `mod_lexer_dfa_c90_skip.c`.

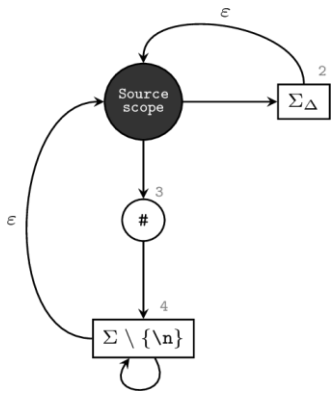

Figura 12, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *skip*.

4.2.8 Sotto-modulo *comment*

Il sotto-modulo *comment* permette al *Lexer* di riconoscere le porzioni di documentazione funzionale. Questa porzione di automa a stati finiti consente infatti al *Lexer* di cambiare logica di *pattern-matching*, effettuando una commutazione dalla radice **`Comment scope`**, verso la radice **`Docs scope`**.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 13**, ed il relativo codice sorgente può essere consultato nei file **`mod_lexer_dfa_c90_comment.h`** e **`mod_lexer_dfa_c90_comment.c`**.

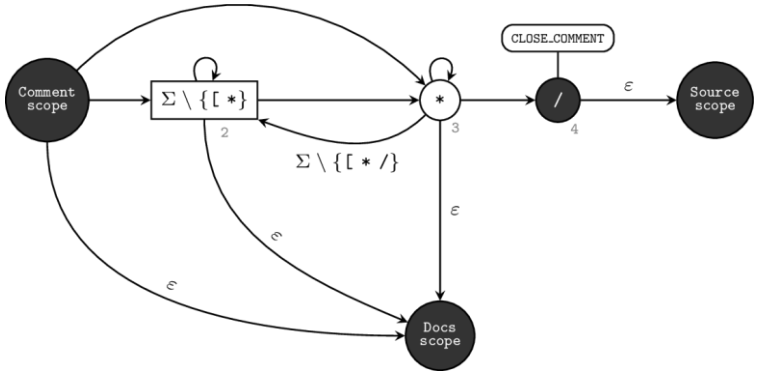

Figura 13, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *comment*.

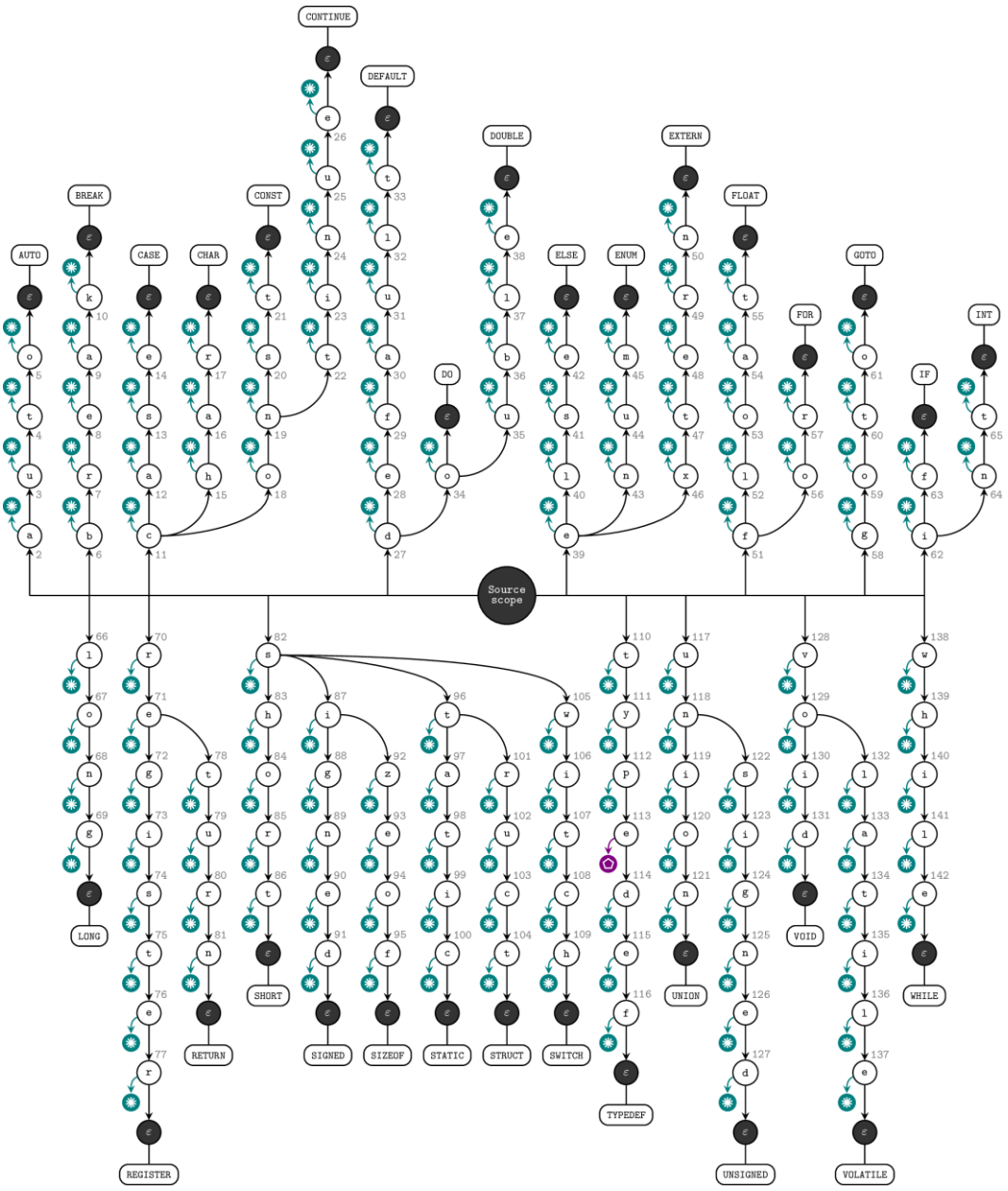

Figura 14, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *keyword*.

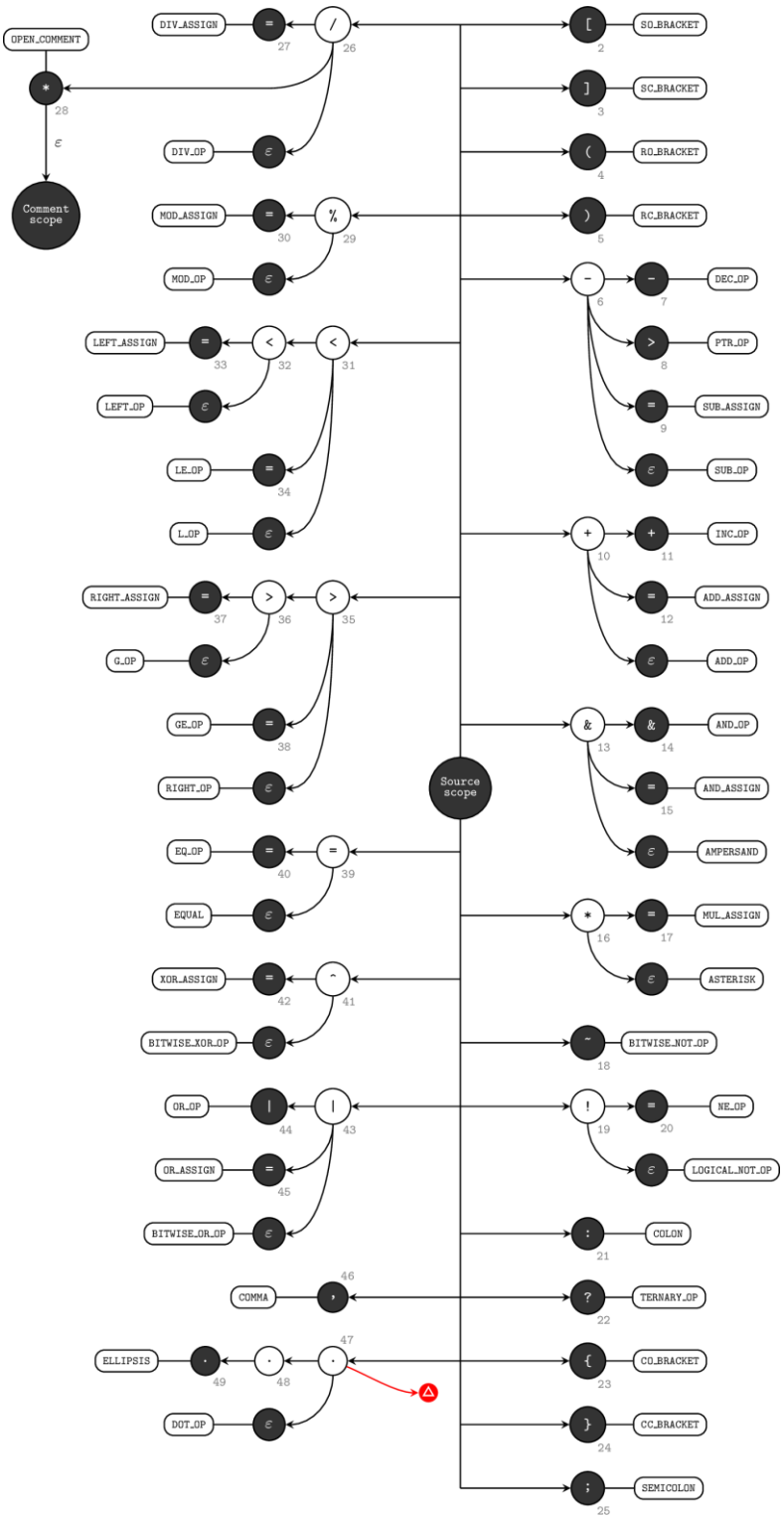

Figura 15, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *symbol*.

4.3 *Pattern-matching* dei *token* della documentazione funzionale

4.3.1 Sotto-modulo *v_doc*

Token. Il sotto-modulo *v_doc*, del *Lexer*, è responsabile del *pattern-matching* dei token `V_PARAM`, `V_DESCRIPTION` e `V_RECURSIVE`, necessari per supportare la scrittura della documentazione funzionale.

Commutazioni. Tale porzione di *automa a stati finiti* trae vantaggio dalla capacità della libreria *DFA* di gestire molteplici radici, permette infatti la commutazione dalla radice `Docs scope`, verso le radici `Docs rule scope`, e `Comment scope`.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 16**, ed il relativo codice sorgente può essere consultato nei file `mod_lexer_dfa_v_doc.h` e `mod_lexer_dfa_v_doc.c`.

4.3.2 Sotto-modulo *v_doc_rule*

Token. Il sotto-modulo *v_doc_rule*, del *Lexer*, è responsabile del *pattern-matching* dei token `V_KEYWORD`, `V_COMMA`, `V_PLUS`, `V_STRING` e `V_IDENTIFIER`, necessari per supportare la scrittura della documentazione funzionale. Viene inoltre gestito il pattern matching del token `CLOSE_COMMENT`.

Commutazioni. Tale porzione di *automa a stati finiti* gestisce la commutazione dalla radice `Doc rule scope`, verso le radici `Source scope`, e `Comment scope`.

Viene data una rappresentazione grafica della porzione di *automa a stati finiti* nella **Figura 17**, ed il relativo codice sorgente può essere consultato nei file `mod_lexer_dfa_v_doc_rule.h` e `mod_lexer_dfa_v_doc_rule.c`.

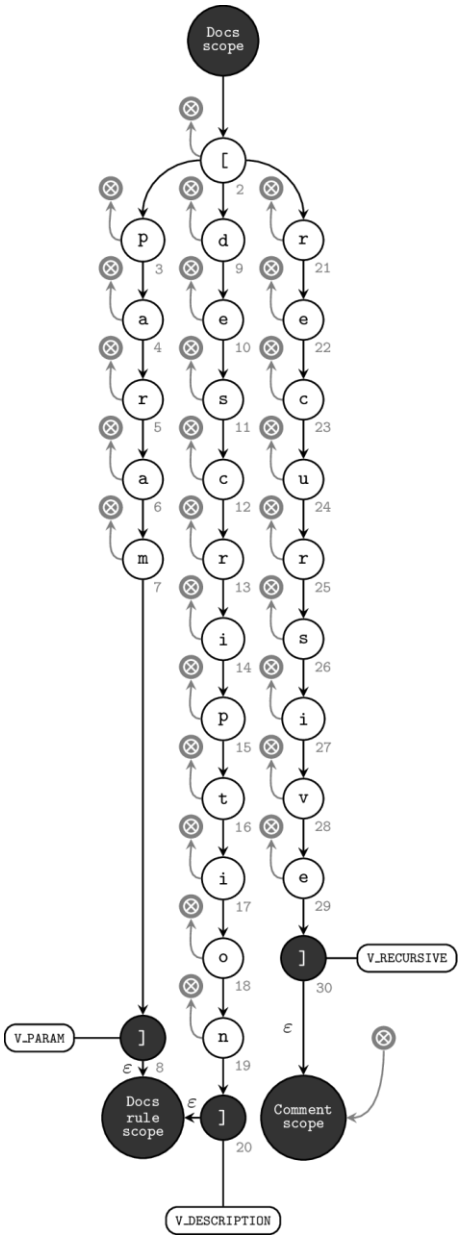

Figura 16, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *v_doc*.

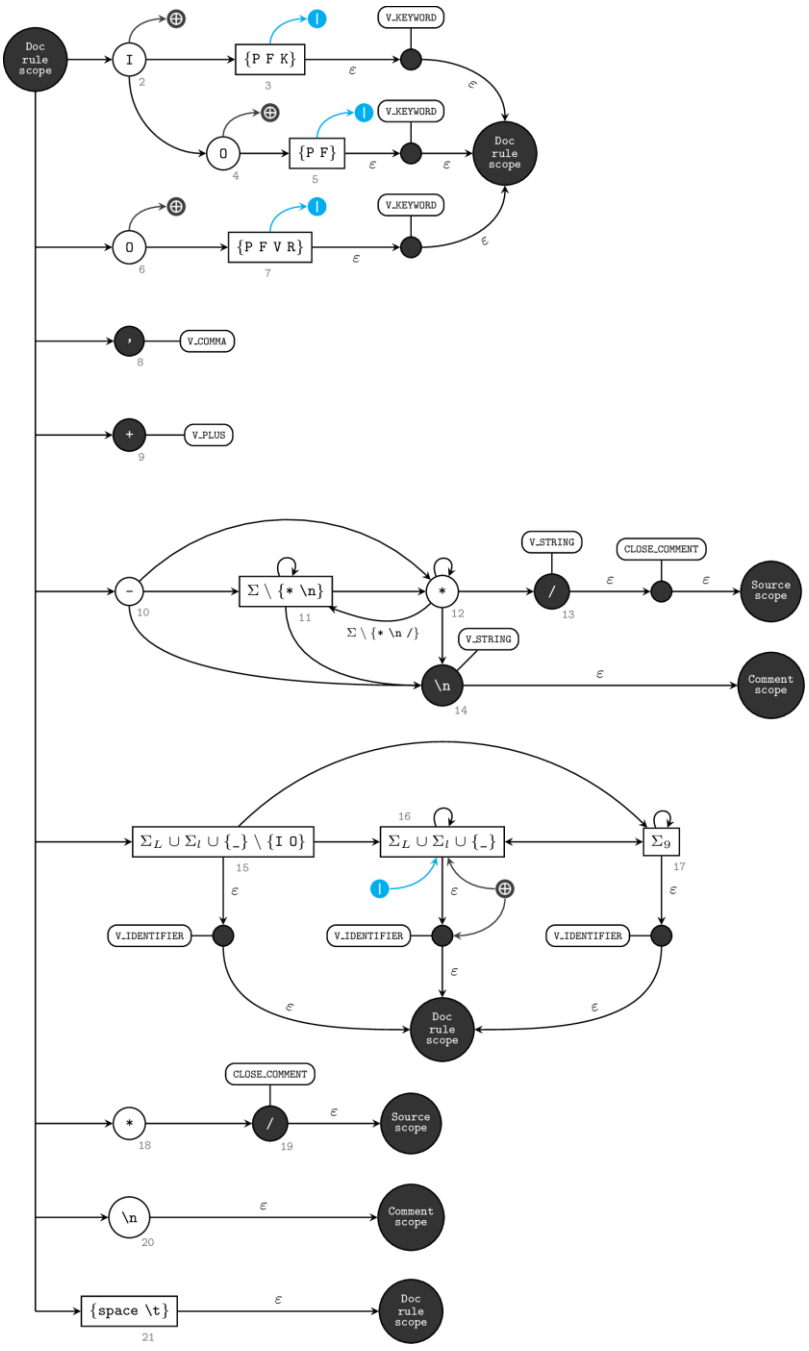

Figura 17, rappresentazione della porzione di *DFA* implementata dal sotto-modulo *v_doc_rule*.

Capitolo 5

Parser

5.1 Introduzione

Il *Parser* è il modulo dell'applicativo responsabile dell'*analisi sintattica* dei sorgenti *C90*. Il *Parser* opera analizzando la sequenza di *token* restituita dal *Lexer*, identificando le relazioni strutturali tra di essi, riconoscendo i costrutti del linguaggio di programmazione, e costruendo una rappresentazione del sorgente strutturata in forma di *albero*, nota come *Abstract Syntax Tree* (*AST*). Il processo di *parsing* valuta la sintassi del sorgente basandosi sulle regole grammaticali del linguaggio di programmazione.

Tecniche di parsing. Esistono vari paradigmi per l'implementazione di un *parser*, ma possono genericamente essere categorizzati in due classi: *parsing top-down*, e *parsing bottom-up*. Il *parsing top-down* inizia l'analisi dalla radice dell'*AST*, e procede verso le foglie seguendo le *regole di produzione* della grammatica. Il *parsing bottom-up*, invece, permette di costruire l'*AST* procedendo dalle foglie verso la radice, aggregando progressivamente i *token* in unità sintattiche superiori, fino ad un'intera ricostruzione del sorgente.

Tra le tecniche di *parsing top-down*, citiamo il *parsing LL(1)* (*left-to-right, leftmost derivation, with 1 look-ahead token*). Tale tecnica viene reputata particolarmente semplice e di facile implementazione. Dimostreremo, però, perché non si riveli adatta al *parsing* del linguaggio *C90*.

Ambiguità. Dipendentemente dalla tecnica di *parsing* adottata, e dalla grammatica del linguaggio, è possibile che una stessa sequenza di *token* abbia più di un'unica interpretazione. In tal caso la grammatica viene definita *ambigua*. Tali ambiguità possono talvolta essere risolte con una modifica della grammatica, o istruendo il *Parser* all'utilizzo di regole di precedenza e associatività.

La tecnica di *parsing LL(1)*, ad esempio, è sensibile alla presenza di *regole di produzione* che mostrino *left-recursion*[5]. La grammatica del linguaggio *C90*, dettagliata nel documento *ISO/IEC 9899:1990*, definisce molte *regole di produzione* caratterizzate da ricorsione a sinistra. Per questo motivo, è stato scelto di non utilizzare il *parsing LL(1)*, preferendo, piuttosto, una tecnica più complessa ma potente.

5.1.1 *Bison*

Bison è uno strumento di generazione di parser distribuito dalla *Free Software Foundation* come parte del progetto *GNU*. Viene utilizzato per la costruzione di *parser LALR(1)* (*look-ahead*, *left-to-right*, *rightmost derivation*, *with 1 look-ahead token*), strategia di analisi sintattica *bottom-up*.

Tra i progetti che fanno uso di *Bison*, annoveriamo nomi noti quali *Bash*, *CMake*, *MySQL*, *PHP* e *PostgreSQL*. In particolare, *Bison* è stato utilizzato dalla *Free Software Foundation* per lo sviluppo del compilatore *GCC*, fino alla versione *4.1* del 2004. Per tali motivi, l'utilizzo di *Bison* è stato reputato una scelta ottimale per la progettazione del modulo *Parser*. È stato infatti possibile implementare il *Parser*

[5] Una *regola di produzione* presenta *left-recursion* quando il simbolo non terminale a sinistra, si espande in una forma che comincia con il simbolo non terminale stesso. Viene dato un esempio di *regola di produzione* che presenta tale caratteristica.

```
<digit-sequence> ::= <digit-sequence> <digit>
```

senza effettuare pressoché alcuna modifica alla grammatica rispetto alla definizione contenuta nel documento *ISO/IEC 9899:1990*.

Integrazione con il Lexer. *Bison*, generatore di *analizzatori sintattici*, viene comunemente utilizzato in combinazione con *Flex*, generatore di *analizzatori lessicali*. Disponendo però l'applicativo del modulo *Lexer*, è stato necessario effettuare delle modifiche alle logiche standard di *Bison*, in modo che utilizzasse il sistema di *pattern-matching* interno. Tale modifica è stata operata ridefinendo la funzione **`yylex`**, contenuta nel file **`mod_parser.y`**.

Costruzione dell'AST. *Bison* permette di associare ad ogni *regola di produzione* una funzione, denominata *azione semantica*. Ogni qualvolta una *regola di produzione* venga riconosciuta, verrà eseguita la relativa *azione semantica*. All'interno di queste funzioni, è possibile definire le logiche di creazione e collegamento dei nodi dell'*AST*.

Diamo ora un esempio di *regola di produzione*, e di relativa *azione semantica*, di un ipotetico *costrutto somma*. Il simbolo **`$$`** associa un valore alla regola di produzione **`sum`**, nel caso specifico il *GraphNode* restituito dalla funzione **`graph_add_node`**. I simboli **`$1`** e **`$3`**, invece, contengono i valori associati rispettivamente al primo e al terzo termine della parte destra, ovvero il valore calcolato dall'*azione semantica* della *regola di produzione* **`term`**. In tal modo è possibile creare il *GraphNode* associato al *costrutto somma*, e collegare a questo i *GraphNode* associati ai costrutti dei singoli termini.

```
sum
  : term PLUS term {
    counter += 1;

    $$ = graph_add_node(ast, int_to_string(counter));
    graph_node_set_field($$, "type", "sum");

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;
```

5.1.2 Ambiguità

Ambiguità *dandling-else*. L'ambiguità *dandling-else* è un comune problema delle grammatiche di linguaggi di programmazione che utilizzano istruzioni condizionali *if-else* senza richiedere, obbligatoriamente, parentesi graffe che delimitino i blocchi di istruzioni. Nello specifico, quando si incontra un *else* innestato senza una chiara delimitazione sintattica, non è immediatamente chiaro se questo debba essere associato all'*if* interno o esterno.

Tale ambiguità è l'unica presentata dalla grammatica *C90*, come definita dal documento *ISO/IEC 9899:1990*, quando implementata da un *parser* costruito con *Bison*. Lo strumento permette, però, di gestire la problematica facilmente, tramite la definizione di regole di precedenza e associatività.

Ambiguità nella nomenclatura dei tipi di dato non nativi. Nel linguaggio *C90*, la distinzione tra i nomi dei tipi di dato non nativi (`TYPE_IDENTIFIER`) e identificatori comuni (`IDENTIFIER`) rappresenta una sfida significativa. Questi, infatti, condividono lo stesso pattern, e quindi non è in alcun modo possibile distinguerli a livello di *Lexer*. Tale ambiguità richiede meccanismi sofisticati per differenziare le due categorie, sovente affidandosi ad un'analisi contestuale.

Nell'ambito di studio di questa tesi, non è stata reputata giustificabile l'aggiunta della complessità che la risoluzione del problema avrebbe richiesto. È stato, piuttosto, considerato preferibile aggiungere una limitazione sulla nomenclatura dei tipi di dato non nativi, non esplicitamente richiesta dallo standard *C90*. Vengono in questo modo giustificate le logiche di *pattern matching* presentate nel paragrafo *4.2.6*.

5.2 Grammatica utilizzata dal *Parser*

5.2.1 Grammatica del linguaggio *C90*

Andiamo ora ad allegare la grammatica del linguaggio *C90* così come specificata dal documento *ISO/IEC 9899:1990*, senza modifica alcuna, se non la rinominazione dei *token* secondo quanto definito dal *Lexer*, ed il supporto per il *token* **`TYPE_IDENTIFIER`**.

```
<identifier> ::= IDENTIFIER

<constant> ::= CONSTANT

<string_literal> ::= STRING_LITERAL

<type_identifier> ::= TYPE_IDENTIFIER

<primary_expression> ::= <identifier>
  | <constant>
  | <string_literal>
  | RO_BRACKET <expression> RC_BRACKET
  ;

<postfix_expression> ::= <primary_expression>
  | <postfix_expression> SO_BRACKET <expression> SC_BRACKET
  | <postfix_expression> RO_BRACKET RC_BRACKET
  | <postfix_expression> RO_BRACKET <argument_expression_list>
      RC_BRACKET
  | <postfix_expression> DOT_OP <identifier>
  | <postfix_expression> PTR_OP <identifier>
  | <postfix_expression> INC_OP
  | <postfix_expression> DEC_OP
  ;

<argument_expression_list> ::= <assignment_expression>
  | <argument_expression_list> COMMA <assignment_expression>
  ;

<unary_expression> ::= <postfix_expression>
```

```
  | INC_OP <unary_expression>
  | DEC_OP <unary_expression>
  | <unary_operator> <cast_expression>
  | SIZEOF <unary_expression>
  | SIZEOF RO_BRACKET <type_name> RC_BRACKET
  ;

<unary_operator> ::= AMPERSAND
  | ASTERISK
  | ADD_OP
  | SUB_OP
  | BITWISE_NOT_OP
  | LOGICAL_NOT_OP
  ;

<cast_expression> ::= <unary_expression>
  | RO_BRACKET <type_name> RC_BRACKET <cast_expression>
  ;

<multiplicative_expression> ::= <cast_expression>
  | <multiplicative_expression> ASTERISK <cast_expression>
  | <multiplicative_expression> DIV_OP <cast_expression>
  | <multiplicative_expression> MOD_OP <cast_expression>
  ;

<additive_expression> ::= <multiplicative_expression>
  | <additive_expression> ADD_OP <multiplicative_expression>
  | <additive_expression> SUB_OP <multiplicative_expression>
  ;

<shift_expression> ::= <additive_expression>
  | <shift_expression> LEFT_OP <additive_expression>
  | <shift_expression> RIGHT_OP <additive_expression>
  ;

<relational_expression> ::= <shift_expression>
  | <relational_expression> L_OP <shift_expression>
  | <relational_expression> G_OP <shift_expression>
  | <relational_expression> LE_OP <shift_expression>
  | <relational_expression> GE_OP <shift_expression>
  ;

<equality_expression> ::= <relational_expression>
  | <equality_expression> EQ_OP <relational_expression>
```

```
  | <equality_expression> NE_OP <relational_expression>
  ;

<and_expression> ::= <equality_expression>
  | <and_expression> AMPERSAND <equality_expression>
  ;

<exclusive_or_expression> ::= <and_expression>
  | <exclusive_or_expression> BITWISE_XOR_OP <and_expression>
  ;

<inclusive_or_expression> ::= <exclusive_or_expression>
  | <inclusive_or_expression> BITWISE_OR_OP <exclusive_or_expression>
  ;

<logical_and_expression> ::= <inclusive_or_expression>
  | <logical_and_expression> AND_OP <inclusive_or_expression>
  ;

<logical_or_expression> ::= <logical_and_expression>
  | <logical_or_expression> OR_OP <logical_and_expression>
  ;

<conditional_expression> ::= <logical_or_expression>
  | <logical_or_expression> TERNARY_OP <expression> COLON
      <conditional_expression>
  ;

<assignment_expression> ::= <conditional_expression>
  | <unary_expression> <assignment_operator> <assignment_expression>
  ;

<assignment_operator> ::= EQUAL
  | MUL_ASSIGN
  | DIV_ASSIGN
  | MOD_ASSIGN
  | ADD_ASSIGN
  | SUB_ASSIGN
  | LEFT_ASSIGN
  | RIGHT_ASSIGN
  | AND_ASSIGN
  | XOR_ASSIGN
  | OR_ASSIGN
  ;
```

```
<expression> ::= <assignment_expression>
  | <expression> COMMA <assignment_expression>
  ;

<constant_expression> ::= <conditional_expression>

<declaration> ::= <declaration_specifiers> SEMICOLON
  | <declaration_specifiers> <init_declarator_list> SEMICOLON
  ;

<declaration_specifiers> ::= <storage_class_specifier>
  | <storage_class_specifier> <declaration_specifiers>
  | <type_specifier>
  | <type_specifier> <declaration_specifiers>
  | <type_qualifier>
  | <type_qualifier> <declaration_specifiers>
  ;

<init_declarator_list> ::= <init_declarator>
  | <init_declarator_list> COMMA <init_declarator>
  ;

<init_declarator> ::= <declarator>
  | <declarator> EQUAL <initializer>
  ;

<storage_class_specifier> ::= TYPEDEF
  | EXTERN
  | STATIC
  | AUTO
  | REGISTER
  ;

<type_specifier> ::= VOID
  | CHAR
  | SHORT
  | INT
  | LONG
  | FLOAT
  | DOUBLE
  | SIGNED
  | UNSIGNED
  | <struct_or_union_specifier>
```

```
  | <enum_specifier>
  | <type_identifier>
  ;

<struct_or_union_specifier> ::=
  | <struct_or_union> <identifier> CO_BRACKET
      <struct_declaration_list> CC_BRACKET
  | <struct_or_union> <type_identifier> CO_BRACKET
      <struct_declaration_list> CC_BRACKET
  | <struct_or_union> CO_BRACKET <struct_declaration_list> CC_BRACKET
  | <struct_or_union> <identifier>
  | <struct_or_union> <type_identifier>
  ;

<struct_or_union> ::= STRUCT
  | UNION
  ;

<struct_declaration_list> ::= <struct_declaration>
  | <struct_declaration_list> <struct_declaration>
  ;

<struct_declaration> ::=
  | <specifier_qualifier_list> <struct_declarator_list> SEMICOLON
  ;

<specifier_qualifier_list> ::=
  | <type_specifier> <specifier_qualifier_list>
  | <type_specifier>
  | <type_qualifier> <specifier_qualifier_list>
  | <type_qualifier>
  ;

<struct_declarator_list> ::= <struct_declarator>
  | <struct_declarator_list> COMMA <struct_declarator>
  ;

<struct_declarator> ::= <declarator>
  | <declarator> COLON <constant_expression>
  | COLON <constant_expression>
  ;

<enum_specifier> ::= ENUM CO_BRACKET <enumerator_list> CC_BRACKET
  | ENUM <identifier> CO_BRACKET <enumerator_list> CC_BRACKET
```

```
  | ENUM <identifier>
  ;

<enumerator_list> ::= <enumerator>
  | <enumerator_list> COMMA <enumerator>
  ;

<enumerator> ::= <identifier>
  | <identifier> EQUAL <constant_expression>
  ;

<type_qualifier> ::= CONST
  | VOLATILE
  ;

<declarator> ::= <pointer> <direct_declarator>
  | <direct_declarator>
  ;

<direct_declarator> ::= <identifier>
  | RO_BRACKET <declarator> RC_BRACKET
  | <direct_declarator> SO_BRACKET <constant_expression> SC_BRACKET
  | <direct_declarator> SO_BRACKET SC_BRACKET
  | <direct_declarator> RO_BRACKET <parameter_type_list> RC_BRACKET
  | <direct_declarator> RO_BRACKET <identifier_list> RC_BRACKET
  | <direct_declarator> RO_BRACKET RC_BRACKET

<pointer> ::= ASTERISK
  | ASTERISK <type_qualifier_list>
  | ASTERISK <pointer>
  | ASTERISK <type_qualifier_list> <pointer>
  ;

<type_qualifier_list> ::= <type_qualifier>
  | <type_qualifier_list> <type_qualifier>
  ;

<parameter_type_list> ::= <parameter_list>
  | <parameter_list> COMMA ELLIPSIS
  ;

<parameter_list> ::= <parameter_declaration>
  | <parameter_list> COMMA <parameter_declaration>
  ;
```

```
<parameter_declaration> ::= <declaration_specifiers> <declarator>
  | <declaration_specifiers> <abstract_declarator>
  | <declaration_specifiers>
  ;

<identifier_list> ::= <identifier>
  | <identifier_list> COMMA <identifier>
  ;

<type_name> ::= <specifier_qualifier_list>
  | <specifier_qualifier_list> <abstract_declarator>
  ;

<abstract_declarator> ::= <pointer>
  | <direct_abstract_declarator>
  | <pointer> <direct_abstract_declarator>
  ;

<direct_abstract_declarator> ::=
  | RO_BRACKET <abstract_declarator> RC_BRACKET
  | SO_BRACKET SC_BRACKET
  | SO_BRACKET <constant_expression> SC_BRACKET
  | <direct_abstract_declarator> SO_BRACKET SC_BRACKET
  | <direct_abstract_declarator> SO_BRACKET <constant_expression>
      SC_BRACKET
  | RO_BRACKET RC_BRACKET
  | RO_BRACKET <parameter_type_list> RC_BRACKET
  | <direct_abstract_declarator> RO_BRACKET RC_BRACKET
  | <direct_abstract_declarator> RO_BRACKET <parameter_type_list>
      RC_BRACKET
  ;

<initializer> ::= <assignment_expression>
  | CO_BRACKET <initializer_list> CC_BRACKET
  | CO_BRACKET <initializer_list> COMMA CC_BRACKET
  ;

<initializer_list> ::= <initializer>
  | <initializer_list> COMMA <initializer>
  ;

<statement> ::= <labeled_statement>
  | <compound_statement>
```

```
  | <expression_statement>
  | <selection_statement>
  | <iteration_statement>
  | <jump_statement>
  ;

<labeled_statement> ::= <identifier> COLON <statement>
  | CASE <constant_expression> COLON <statement>
  | DEFAULT COLON <statement>
  ;

<compound_statement> ::= CO_BRACKET CC_BRACKET
  | CO_BRACKET <statement_list> CC_BRACKET
  | CO_BRACKET <declaration_list> CC_BRACKET
  | CO_BRACKET <declaration_list> <statement_list> CC_BRACKET
  ;

<declaration_list> ::= <declaration>
  | <declaration_list> <declaration>
  ;

<statement_list> ::= <statement>
  | <statement_list> <statement>
  ;

<expression_statement> ::= SEMICOLON
  | <expression> SEMICOLON
  ;

<selection_statement> ::=
  | IF RO_BRACKET <expression> RC_BRACKET <statement>
  | IF RO_BRACKET <expression> RC_BRACKET <statement> ELSE
      <statement>
  | SWITCH RO_BRACKET <expression> RC_BRACKET <statement>
  ;

<iteration_statement> ::=
  | WHILE RO_BRACKET <expression> RC_BRACKET <statement>
  | DO <statement> WHILE RO_BRACKET <expression> RC_BRACKET SEMICOLON
  | FOR RO_BRACKET <expression_statement> <expression_statement>
      RC_BRACKET <statement>
  | FOR RO_BRACKET <expression_statement> <expression_statement>
      <expression> RC_BRACKET <statement>
  ;
```

```
<jump_statement> ::= GOTO <identifier> SEMICOLON
  | CONTINUE SEMICOLON
  | BREAK SEMICOLON
  | RETURN SEMICOLON
  | RETURN <expression> SEMICOLON
  ;

<translation_unit> ::= <external_declaration>
  | <translation_unit> <external_declaration>
  ;

<external_declaration> ::= <function_definition>
  | <declaration>
  ;

<function_definition> ::=
  | <declaration_specifiers> <declarator> <declaration_list>
      <compound_statement>
  | <declaration_specifiers> <declarator> <compound_statement>
  | <declarator> <declaration_list> <compound_statement>
  | <declarator> <compound_statement>
  ;

<program> ::= <translation_unit>
```

5.2.2 Grammatica della documentazione

Integriamo la grammatica del linguaggio *C90*, con le regole di produzione necessarie a riconoscere i costrutti della documentazione funzionale.

```
<comment> ::= OPEN_COMMENT <v_rule_list> CLOSE_COMMENT
  | OPEN_COMMENT CLOSE_COMMENT
  ;

<external_declaration> ::= <function_definition>
  | <declaration>
  | <comment>
  ;

<v_rule_list> ::= <v_rule>
  | <v_rule_list> <v_rule>
```

```
  ;

<v_rule> ::= <v_rule_param>
  | <v_rule_recursive>
  | <v_rule_description>
  ;

<v_rule_param> ::=
  | V_PARAM <v_keyword_list> V_COMMA <v_identifier> <v_string>
  | V_PARAM <v_keyword_list> V_COMMA <v_identifier>
  | V_PARAM <v_identifier> <v_string>
  | V_PARAM <v_identifier>
  | V_PARAM <v_keyword_list> <v_string>
  | V_PARAM <v_keyword_list>
  ;

<v_rule_recursive> ::= V_RECURSIVE

<v_rule_description> ::= V_DESCRIPTION <v_string>

<v_keyword_list> ::= <v_keyword>
  | <v_keyword_list> V_PLUS <v_keyword>
  ;

<v_identifier> ::= V_IDENTIFIER

<v_string> ::= V_STRING

<v_keyword> ::= V_KEYWORD
```

5.2.2.1 Filtraggio dei *token* della documentazione funzionale

Si noti che la regola di produzione **`comment`**, viene esclusivamente supportata a livello di **`external_declaration`**. Questo è sensato, perché la documentazione funzionale è esclusivamente prevista all'esterno della definizione delle funzioni. Ciò, però, comporta che il *Parser* non sia istruito rispetto alla gestione del token **`V_OPEN_COMMENT`** dentro al corpo di una funzione. In ultima analisi questo significa che, in assenza di ulteriori logiche, la sola presenza di un commento all'interno di una funzione possa mandare il *Parser* in errore.

È stato quindi implementato, all'interno della funzione **`yylex`**, un meccanismo aggiuntivo. Viene infatti operato un filtraggio dei *token*, in modo che quelli relativi alla documentazione funzionale, se definiti all'interno del corpo di una funzione, vengano scartati prima di arrivare al *Parser*.

5.3 Esempi di *AST*

Alleghiamo ora degli esempi di *Abstract Syntax Tree*, generati dal parser con l'analisi di due semplici funzioni *C90*. I relativi AST possono essere trovati rispettivamente nelle immagini **Figura 18** e **Figura 19**.

`helloworld.c`:

```
int main(void) {
  printf("Hello, World!");
  return 0;
}
```

`hypotenuse.c`:

```
double hypotenuse(double a, double b) {
  return sqrt(pow(a, 2) + pow(b, 2));
}
```

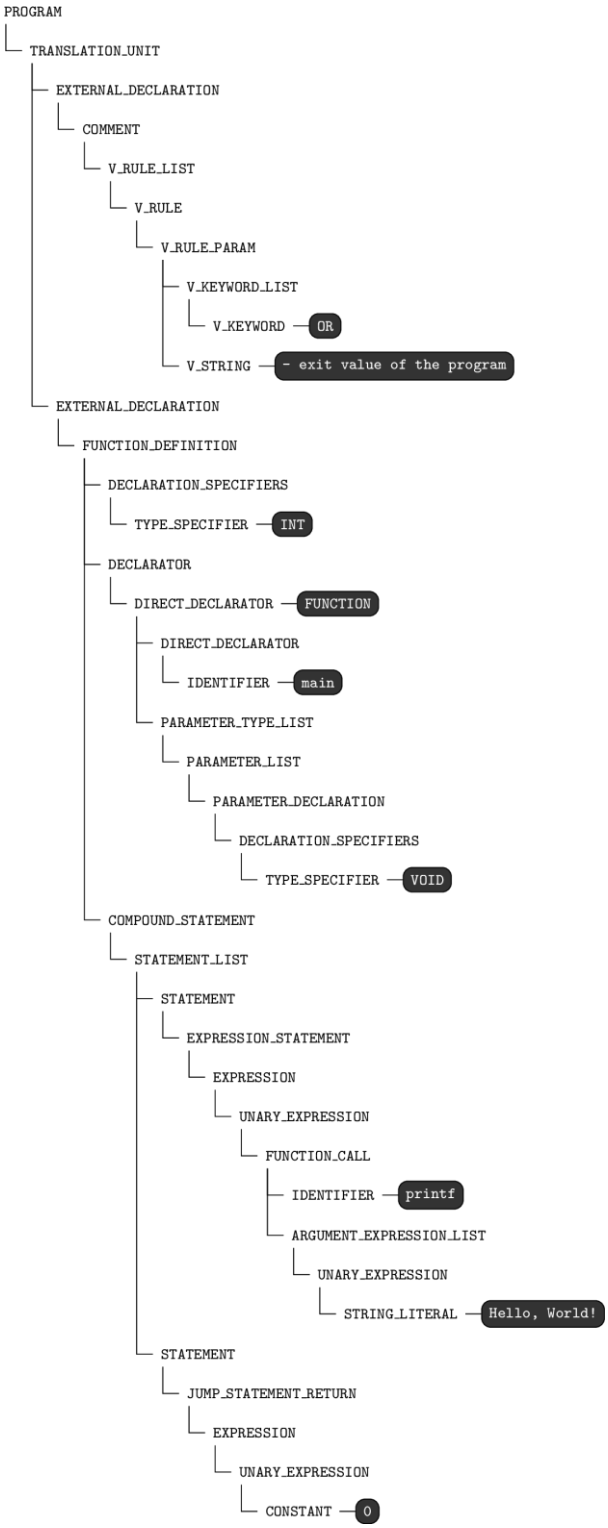

Figura 18, *AST* del programma **`helloworld.c`**

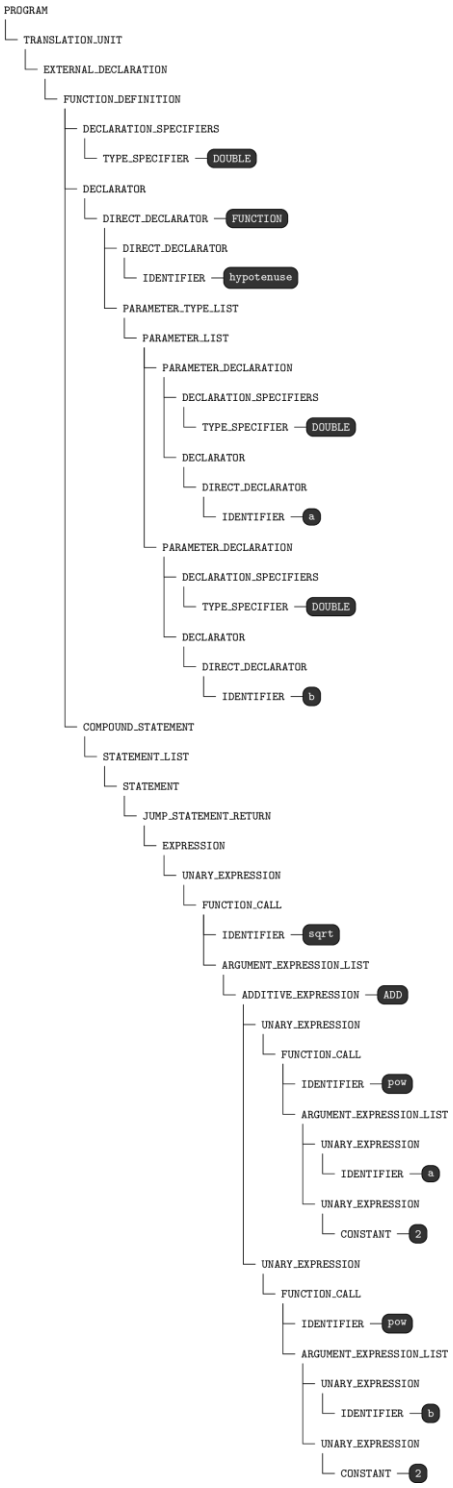

Figura 19, *AST* del programma **`hypotenuse.c`**

Capitolo 6

Analyzer

6.1 Introduzione

Analisi statica. L'*Analyzer* è il modulo dell'applicativo responsabile dell'*analisi statica* del codice sorgente. Opera sull'*AST* utilizzando le funzioni di esplorazione della libreria *Graph* per stabilire, in ultima analisi, se per ogni funzione la relativa documentazione funzionale sia stata redatta correttamente. A tal fine, viene operata una rudimentale *data flow analysis*, che permette di mappare, mediante la costruzione di un *grafo delle dipendenze* degli identificatori, le relazioni tra le variabili, e come queste si evolvano a seguito di operazioni di assegnazione e chiamata a funzione.

Destinazioni d'uso. Nell'ambito della scrittura della documentazione funzionale, risulta utile poter esprimere come un parametro, o più in generale una variabile, venga utilizzato all'interno di una funzione. Definiamo, a tal fine, la nozione di *destinazione d'uso*, derivando il concetto da quanto trattato dall'insegnamento di *Laboratorio di Ingegneria Informatica*. Tra le *destinazioni d'uso*, possiamo quindi annoverare:

- `IP`: indica che una variabile viene utilizzata come input di una funzione;
- `IK`: indica che una variabile viene ottenuta quale input da tastiera;
- `IF`: indica che una variabile viene ottenuta quale input da file;

- `OP`: indica che una variabile viene utilizzata quale destinazione di un'operazione di assegnazione;
- `OV`: indica che una variabile viene stampata sullo schermo;
- `OF`: indica che una variabile viene stampata su di un file;
- `OR`: *destinazione d'uso* che esula da quanto abbiamo definito, non potendo di fatto essere associata ad una variabile, bensì ad una funzione; permette infatti di indicare qualora una funzione restituisca un valore non `void`.

6.1.1 *Data flow analysis*

La *data flow analysis* è una tecnica di *analisi statica* che valuta il flusso delle informazioni, in termini di valore delle variabili, attraverso il codice sorgente. Tale tipo di analisi si concretizza nella costruzione di un *grafo delle dipendenze* degli identificatori, che rappresenta come i valori assegnati a questi siano interdipendenti. Valutare l'evoluzione di tale *grafo* in risposta alle operazioni effettuate sugli identificatori, permette di capire come le variabili vengano trasformate e propagate durante l'esecuzione di una funzione. Tale processo permette di identificare potenziali difetti del codice sorgente, quali la presenza di variabili non inizializzate e accessi alla memoria non sicuri.

L'applicativo utilizza le tecniche di *data flow analysis* per etichettare le variabili con le relative *destinazioni d'uso*, in seguito a operazioni di assegnazione o di chiamata a funzione.

Entriamo ora nel merito dell'implementazione dell'*Analyzer*, trattando in che modo il *grafo delle dipendenze* evolva in risposta a operazioni di assegnazione o di chiamata a funzione, e in che modo le *destinazioni d'uso* vengano eventualmente ereditate.

6.1.1.1 Operazioni di assegnazione

Le operazioni di assegnazione aggiornano il *grafo delle dipendenze*, e nello specifico i nodi associati all'identificatore presente nella parte sinistra dell'operazione (*l-value*), e i nodi associati agli identificatori presenti alla destra dell'operatore (*r-value*). A seguito di un'operazione di assegnazione, vengono inoltre aggiornate e propagate alcune *destinazioni d'uso* degli identificatori coinvolti. È doveroso, però, distinguere alcune casistiche.

Assegnazione semplice. Quando si verifica un'operazione di assegnazione semplice (`=`) il valore di una variabile viene riassegnato e, di conseguenza, il *grafo delle dipendenze* degli identificatori viene modificato come segue:

- Viene cancellato ogni arco, entrante o uscente, del nodo del *grafo delle dipendenze* corrispondente all'identificatore della *l-value*; questo perché l'assegnazione semplice opera una riassegnazione, e quindi ogni dipendenza pregressa viene persa;
- In virtù del medesimo ragionamento, vengono inizializzate le destinazioni d'uso dell'identificatore della *l-value*;
- Vengono creati degli archi che congiungano l'identificatore della *l-value*, ad ogni identificatore della *r-value*; questo per riflettere la dipendenza ora instaurata del termine alla sinistra dell'operatore di assegnazione, rispetto ai termini alla destra di esso.

Assegnazione composta. Le operazioni di assegnazione composta (`*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`) invece, fanno sì che il valore venga aggiornato, modificando il *grafo delle dipendenze* come segue:

- Gli archi del nodo del grafo delle dipendenze corrispondente all'identificatore della *l-value* vengono mantenuti, perché le assegnazioni composte non operano una riassegnazione;
- Le destinazioni d'uso della *l-value* vengono mantenute;

- Vengono creati, similarmente al caso dell'assegnazione semplice, archi che congiungano l'identificatore della *l-value*, con gli identificatori della *r-value*

Assegnazione a puntatori, membri di strutture e array. Quando viene operata un'assegnazione, sia essa semplice o composta, su di un puntatore, sul membro di una struttura, o sul membro di un array, allora in generale non possiamo dire abbia avuto luogo la riassegnazione della relativa variabile. Il *grafo delle dipendenze* viene pertanto aggiornato similarmente a quanto fatto nel caso di assegnazione composta.

Aggiornamento delle *destinazioni di uso*. A valle della manipolazione del grafo delle dipendenze, la logica di aggiornamento *delle destinazioni d'uso* delle variabili è la medesima:

- `OP`: all'identificatore della *l-value* viene attribuita la *destinazione d'uso* `OP`, in quanto è stato oggetto di assegnazione;
- `IK`, `IF`: qualora almeno uno degli identificatori della *r-value* sia stato etichettato con la *destinazione d'uso* `IK`, allora questa caratteristica viene ereditata dall'identificatore della *l-value*; similarmente viene fatto per la *destinazione d'uso* `IF`; tale logica permette di tenere traccia delle variabili ottenute come input da tastiera o file.

6.1.1.2 Operazioni di chiamata a funzione

Le operazioni di chiamata a funzione hanno come principale esito l'aggiornamento delle *destinazioni d'uso* degli identificatori utilizzati quali argomenti, nel dettaglio:

- `IK`: quando un identificatore viene utilizzato tra i *varargs* di `scanf`, a questo viene assegnata la *destinazione d'uso* `IK`;
- `IF`: quando un identificatore viene utilizzato tra i *varargs* di `fscanf`, a questo viene assegnato la *destinazione d'uso* `IF`;

- **`OP`**: quando un identificatore viene utilizzato quale argomento di una funzione che opera, rispetto a questo, un'assegnazione, vi viene associata la *destinazione d'uso* **`OP`**; si noti che si comportano in tal modo i *varargs* delle funzioni **`scanf`** e **`fscanf`**;

- **`OV`**: quando un identificatore viene utilizzato tra i *varargs* di **`printf`**, a questo viene assegnato la *destinazione d'uso* **`OV`**;

- **`OF`**: quando un identificatore viene utilizzato tra i *varargs* di **`fprintf`**, a questo viene assegnato la *destinazione d'uso* **`OF`**;

6.1.1.3 Parametri funzionali

Al fine di valutare l'esito che le operazioni di chiamata a funzione hanno sulla *data flow analysis*, l'*Analyzer* introduce il concetto di *destinazione d'uso* dei parametri funzionali. Viene pertanto tenuta traccia non esclusivamente delle *destinazioni d'uso* degli identificatori all'interno della funzione, ma anche delle *destinazioni d'uso* dei parametri delle funzioni a livello di programma. Le *destinazioni d'uso* degli identificatori, anzi, hanno specificatamente il compito di guidare la logica di determinazione delle *destinazioni d'uso* dei parametri, essendo questi, in ultima analisi, ciò che viene effettivamente trattato a livello di documentazione funzionale.

Aggiornamento delle *destinazioni d'uso* dei parametri. Ogni parametro è caratterizzato da un identificatore. Abbiamo già illustrato come agli identificatori, in fase di *data flow analysis*, vengano associate delle *destinazioni d'uso*, progressivamente aggiornate dalle operazioni di assegnazione e chiamata a funzione. In quest'ottica, intendiamo le *destinazioni d'uso* dei parametri come *storico* delle *destinazioni d'uso* progressivamente associate al relativo identificatore all'interno della funzione.

Al termine di ogni operazione, che eventualmente comporterà l'aggiornamento delle *destinazioni d'uso* come abbiamo visto, l'*Analyzer* allinea le *destinazioni d'uso*

di ogni parametro a quelle del relativo identificatore. Le *destinazioni d'uso* dei parametri non verranno mai resettate, rappresentando di fatto un *riassunto* di come l'identificatore associato sia stato utilizzato all'interno del corpo della funzione.

Le *destinazioni d'uso* dei parametri caratterizzano quindi le funzioni. Venendo tenuta traccia di queste informazioni a livello di programma, è evidente si possa ora estendere il meccanismo di attribuzione delle *destinazioni d'uso* agli identificatori in seguito ad un'operazione di chiamata a funzione. Nello specifico, ogni qualvolta avvenga una simile operazione, l'*Analyzer* verifica se le *destinazioni d'uso* dei parametri della funzione chiamata siano state processate. In tal caso, queste informazioni vengono utilizzate per aggiornare le *destinazioni d'uso* degli identificatori utilizzati quali argomenti.

6.1.1.4 Ordine di valutazione delle funzioni

In virtù di quanto detto, è importante che l'*Analyzer* valuti le funzioni seguendo uno specifico ordine. Nel caso specifico, vogliamo che l'*Analyzer* processi una funzione solo a valle della valutazione di tutte le funzioni chiamate all'interno di questa, che siano state definite nel programma.

Ordinamento *DFS post-order*. In tale casistica, trova evidentemente utilizzo l'algoritmo di ordinamento *DFS post-order*. Ricordando quanto abbiamo trattato nel paragrafo *3.2.4.3*, l'algoritmo in questione ha la caratteristica di elaborare ogni nodo, solo a valle della visita di tutti i nodi discendenti.

La soluzione al problema si limita, quindi, alla creazione di un *grafo* che rappresenti le dipendenze tra le funzioni, seguita dall'applicazione dell'algoritmo *DFS post-order*, offerto dalla libreria *Graph*.

***Funzioni ricorsive* e catene di *funzioni ricorsive*.** La presenza di *funzioni ricorsive* rappresenta una significativa sfida nella determinazione dell'ordine di valutazione delle funzioni nell'*analisi statica* del codice. È possibile, inoltre, siano presenti *catene di funzioni ricorsive*, ovvero gruppi di funzioni che si chiamano vicendevolmente in

modo ricorsivo. In tali condizioni, il flusso di controllo del programma si ripete all'interno della stessa funzione, creando cicli nel *grafo delle dipendenze* prima definito. Tali cicli rendono difficile stabilire chiaramente un punto di inizio e di termine per l'*analisi statica*.

Algoritmo di *Kosaraju*. Queste considerazioni hanno motivato la decisione di non supportare i sorgenti che presentino ricorsione. L'*Analyzer*, pertanto, a monte della valutazione delle *destinazioni d'uso* dei parametri funzionali, verifica la presenza di chiamate ricorsive nel programma.

Trovare nel sorgente *funzioni ricorsive* risulta immediato, è infatti sufficiente ricercare, nel *grafo delle dipendenze* delle funzioni, nodi che presentino un *self-loop*.

La logica di ispezione delle *catene di funzioni ricorsive*, invece, richiede un approccio più sofisticato. È però immediato rendersi conto che una *catena di funzioni ricorsive* costituisca, nel *grafo delle dipendenze*, una *componente fortemente connessa*. Trova pertanto applicazione, l'algoritmo di *Kosaraju* che abbiamo presentato nel paragrafo *3.2.4.4*. In ultima analisi, possiamo concludere che l'esistenza di *componenti fortemente connesse* che abbiano al loro interno più di un singolo nodo, è sufficiente per intercettare la presenza di *catene di funzioni ricorsive* nel sorgente.

6.1.2 Confronto con la documentazione

Il *Parser* è stato implementato in modo da estrarre dal sorgente la documentazione funzionale, i cui dettagli possono quindi essere trovati all'interno dell'*Abastract Syntax Tree*. L'*Analyzer* può in questo modo, quindi, operare un confronto tra le *destinazioni d'uso* dei parametri funzionali calcolate come illustrato, e le *destinazioni d'uso* dei parametri funzionali documentati.

Questa cruciale verifica permette di individuare eventuali discrepanze, generando avvertimenti che possano aiutare il programmatore ad operare dovute correzioni. L'*Analyzer* restituisce avvertimenti nei seguenti casi:

- *Unnecessary destination found*: quando viene riscontrata la presenza di una *destinazione d'uso* non necessaria, ovvero quando questa viene trovata nella documentazione funzionale, ma non tra i risultati dell'*Analyzer*;
- *Destination not found*: quando non viene trovata una *destinazione d'uso*, ovvero quando tra i risultati dell'*Analyzer* è presente una *destinazione d'uso* non documentata;
- *Documentation not found*: quando una funzione non risulta documentata.

Tale meccanismo assicura che la documentazione funzionale rifletta accuratamente l'uso dei parametri all'interno del codice, garantendo la qualità e l'affidabilità della documentazione stessa.

6.2 Strutture di dati e algoritmi

Entriamo ora nel merito delle strutture di dati e degli algoritmi che supportano il funzionamento dell'*Analyzer*, che abbiamo informalmente presentato. L'*Analyzer* è organizzato in sotto-moduli.

6.2.1 Sotto-modulo *destination_map*

```
HashMap* analyzer_destination_map_create(void);
int* analyzer_destination_flag_create(void);
void analyzer_destination_map_free(HashMap *destination_map);
```

Il sotto-modulo *destination_map* contiene una serie di funzioni che agevolano la creazione e la gestione delle *HashMap* contenenti le *destinazioni d'uso* di identificatori e parametri. Presentiamo brevemente le funzioni della libreria:

- **`analyzer_destination_map_create`**: crea una *HashMap* con 7 coppie chiave-valore, dove la chiave è l'etichetta della *destinazione d'uso*, ed il valore un puntatore ad un *booleano*, impostato per *default* a **`false`**;
- **`analyzer_destination_flag_create`**: alloca in memoria un valore *booleano*, impostato per *default* a **`false`**;
- **`analyzer_destination_map_free`**: libera la memoria occupata dall'*HashMap*, e dai relativi valori *booleani*.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_destination_map.h`** e **`mod_analyzer_destination_map.c`**.

6.2.2 Sotto-modulo *report_identifier*

```
typedef struct AnalyzerReportIdentifier {
  HashMap *destination_map;
} AnalyzerReportIdentifier;

AnalyzerReportIdentifier* analyzer_report_identifier_create();

void analyzer_report_identifier_compute_heredity(
  const AnalyzerReportIdentifier *parent,
  const AnalyzerReportIdentifier *child
);

void analyzer_report_identifier_free(
  AnalyzerReportIdentifier *report_identifier
);
```

Il sotto-modulo *report_identifier* supporta l'*Analyzer* nella gestione degli identificatori. Viene definita una *struttura di dati*, l'*AnalyzerReportIdentifier*, contenente una *HashMap* per tracciare le *destinazioni d'uso*. Diamo una panoramica delle funzionalità:

- **`analyzer_report_identifier_create`**: inizializza e restituisce un nuovo *AnalyzerReportIdentifier*;
- **`analyzer_report_identifier_compute_heredity`**: propaga le *destinazioni d'uso* tra identificatore genitore e figlio, permettendo di tracciarne l'evoluzione;
- **`analyzer_report_identifier_free`**: libera la memoria allocata dall'*AnalyzerReportIdentifier*.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_report_identifier.h`** e **`mod_analyzer_report_identifier.c`**.

6.2.3 Sotto-modulo *report_parameter*

```
typedef struct AnalyzerReportParameter {
  HashMap *destination_map;
} AnalyzerReportParameter;

typedef struct AnalyzerReportParameterList {
  int parameters_count;
  AnalyzerReportParameter **parameter_reports;
  AnalyzerReportParameter *vararg_report;
  HashMap *parameter_reports_map;
  bool destination_or;
} AnalyzerReportParameterList;

AnalyzerReportParameter* analyzer_report_parameter_create();

AnalyzerReportParameterList* analyzer_report_parameter_list_create(
  int parameters_count
```

```
);

void analyzer_report_parameter_list_free(
  AnalyzerReportParameterList *report_parameter_list
);
```

Il sotto-modulo *report_parameter* permette all'*Analyzer* di rappresentare e gestire la lista dei parametri di una funzione. La *struttura di dati AnalyzerReportParameter* contiene un'*HashMap* con le *destinazioni d'uso* del singolo parametro. La *struttura di dati AnalyzerReportParameterList* aggrega, e tiene traccia, degli *AnalyzerReportParameter* contemporaneamente in un *array* (**`parameter_reports`**) e in un'*HashMap* (**`parameter_reports_map`**). Viene inoltre gestito un *AnalyzerReportParameter* speciale (**`vararg_report`**) per rappresentare le *destinazioni di uso* dei *varargs*. Discutiamo brevemente le funzionalità:

- **`analyzer_report_parameter_create`**: crea e restituisce un nuovo *AnalyzerReportParameter*;
- **`analyzer_report_parameter_list_create`**: crea e restituisce un nuovo *AnalyzerReportParameterList*;
- **`analyzer_report_parameter_list_free`**: libera la memoria associate all'*AnalyzerReportParameterList*.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_report_parameter.h`** e **`mod_analyzer_report_parameter.c`**.

6.2.4 Sotto-modulo *report_function*

```
typedef struct AnalyzerReportFunction {
  Graph *identifier_dependencies;
  HashMap *identifier_reports;
```

```
  AnalyzerReportParameterList *parameter_list_report;
  bool process_documentation;
} AnalyzerReportFunction;

AnalyzerReportFunction *analyzer_report_function_compute(
  const Graph *ast,
  const GraphNode *function_ast_node,
  const AnalyzerReportProgram *report_program
);

AnalyzerReportFunction *analyzer_report_function_create(
  int parameters_count
);

void analyzer_report_function_add_identifier_dependency(
  const AnalyzerReportFunction *report function,
  const char *parent_identifier,
  const char *child_identifier
);

void analyzer_report_function_free(
  AnalyzerReportFunction *report_function
);

void analyzer_report_function_register_identifier(
  const AnalyzerReportFunction *report_function,
  const char *identifier
);

void analyzer_report_function_reset_identifier(
  const AnalyzerReportFunction *report_function,
  const char *identifier
);
```

Il sotto-modulo *report_function* è fondamentale per la rappresentazione e l'analisi delle funzioni nell'*Analyzer*. Tale sotto-modulo introduce una *struttura di dati*, l'*AnalyzerReportFunction*, contenente: un *grafo* (**`identifier_dependencies`**) per rappresentare le dipendenze tra gli identificatori; un'*HashMap* (**`identifier_reports`**) per tenere traccia dell'*AnalyzerReportIdentifier* associato ad ogni identificatore; il riferimento all'*AnalyzerReportParameterList* della funzione (**`parameter_list_report`**); un valore *booleano* (**`process_documentation`**) che

stabilisce se sia necessario elaborare la documentazione. Il sotto-modulo introduce le seguenti funzionalità:

- **`analyzer_report_function_compute`**: processa un *AnalyzerReportFunction* analizzando il relativo *GraphNode* nell'*Abastract* Syntax Tree;
- **`analyzer_report_function_create`**: inizializza e restituisce un *AnalyzerReportFunction*;
- **`analyzer_report_function_add_identifier_dependency`**: stabilisce la dipendenza tra due identificatori;
- **`analyzer_report_function_free`**: libera la memoria allocate da un *AnalyzerReportFunction*;
- **`analyzer_report_function_register_identifier`**: aggiunge un identificatore al *grafo* delle dipendenze, e alla mappa dei report;
- **`analyzer_report_function_reset_identifier`**: dato un identificatore, cancella ogni arco entrante e uscente del relativo nodo nel *grafo* delle dipendenze; resetta, inoltre, la mappa delle *destinazioni d'uso*.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_report_function.h`** e **`mod_analyzer_report_function.c`**.

6.2.5 Sotto-modulo *operation_assignment*

```
void analyzer_report_function_operation_assignment_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportFunction *report_function
);

void analyzer_report_function_operation_declaration_compute(
  const Graph *ast,
```

```
  const GraphNode *ast_node,
  const AnalyzerReportFunction *report_function
);
```

Il sotto-modulo *operation_assignment* è da intendersi come un'estensione del modulo *report_function*. Contiene infatti le logiche che permettono di gestire le operazioni di assegnazione e dichiarazione. Presentiamo brevemente le funzionalità introdotte:

- **`analyzer_report_function_operation_assignment_compute`**: analizza il nodo nell'*AST* corrispondente alle operazioni di assegnazione, per aggiornare l'*AnalyzerReportFunction*;
- **`analyzer_report_function_operation_declaration_compute`**: analizza il nodo nell'*AST* corrispondente alle operazioni di dichiarazione, per aggiornare l'*AnalyzerReportFunction*.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_report_function_operation_assignment.h`**, **`mod_analyzer_report_function_operation_assignment.c`**.

6.2.6 Sotto-modulo *operation_call*

```
void analyzer_report_function_operation_call_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportProgram *report_program,
  const AnalyzerReportFunction *report_function
);
```

Il sotto-modulo *operation_call*, similarmente al sotto-modulo *operation_assignment*, deve essere inteso come estensione del modulo

report_function. Permette infatti a quest'ultimo di processare le operazioni di chiamata a funzione. Presentiamo la funzionalità:

- **`analyzer_report_function_operation_call_compute`**: analizza il nodo nell'*AST* corrispondente alle operazioni di chiamata a funzione, per aggiornare l'*AnalyzerReportFunction*;

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_report_function_operation_call.h`**, **`mod_analyzer_report_function_operation_call.c`**.

6.2.7 Sotto-modulo *report_documentation*

```
typedef struct AnalyzerReportDocumentation {
  HashMap* parameters;
  bool destination_or;
} AnalyzerReportDocumentation;

AnalyzerReportDocumentation* analyzer_report_documentation_create(
  const Graph *ast,
  const GraphNode *ast_node
);

void analyzer_report_documentation_free(
  AnalyzerReportDocumentation *analyzer_report_documentation
);
```

Il sotto-modulo *report_documentation* è responsabile dell'estrazione della documentazione funzionale dall'*AST* e della sua gestione. Introduce una *struttura di dati*, l'*AnalyzerReportDocumentation*, caratterizzato dalla presenza di un'*HashMap* che contenga gli *AnalyzerReportParameter* (**`parameters`**) dei parametri documentati. Contiene inoltre un valore booleano (**`destination_or`**) per tracciare la presenza, nella documentazione funzionale, della *destinazione d'uso* **`OR`**. Presentiamo brevemente le funzionalità introdotte:

- **`analyzer_report_documentation_create`**: dato il relativo nodo nell'*AST*, crea, computa e restituisce un *AnalyzerReportDocumentation*;
- **`analyzer_report_documentation_free`**: libera la memoria occupata dall'*AnalyzerReportDocumentation*.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_report_documentation.h`**, **`mod_analyzer_report_documentation.c`**.

6.2.8 Sotto-modulo *report_program*

```
typedef struct AnalyzerReportProgram {
  Graph *function_dependencies;
  HashMap *function_reports;
  HashMap *documentation_reports;
} AnalyzerReportProgram;

AnalyzerReportProgram *analyzer_report_program_create(
  const Graph *ast
);

void analyzer_report_program_free(
  AnalyzerReportProgram *report_program
);
```

Il sotto-modulo *report_program* costituisce il cardine dell'analisi dell'intero codice sorgente. Viene in esso, infatti, dichiarata *AnalyzerReportProgram*, una *struttura di dati* contenente: il *grafo delle dipendenze* delle funzioni (**`function_dependencies`**); un'*HashMap* contenente gli *AnalyzerReportFunction* (**`function_reports`**); un'*HashMap* contenente gli *AnalyzerReportDocumentation* (**`documentation_reports`**). Riassumiamo brevemente le funzionalità introdotte:

- **`analyzer_report_program_create`**: crea, processa e restituisce l'*AnalyzerReportProgram*; è inoltre responsabile del controllo dell'assenza di ricorsione nel codice sorgente;
- **`analyzer_report_program_free`**: libera la memoria associata all'*AnalyzerReportProgram*.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_report_program.h`**, **`mod_analyzer_report_program.c`**.

6.2.9 Sotto-modulo *common*

```
List* analyzer_get_identifier_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
);

bool analyzer_check_array_access(
  const Graph *ast,
  const GraphNode *ast_node
);

bool analyzer_check_member_access(
  const Graph *ast,
  const GraphNode *ast_node
);

bool analyzer_check_pointer_access(
  const Graph *ast,
  const GraphNode *ast_node
);

char* analyzer_get_ast_node_type(
  const GraphNode *ast_node
);

char* analyzer_get_ast_node_value(
  const GraphNode *ast_node
```

```
);

char* analyzer_get_identifier_ast_node_value(
  const Graph *ast,
  const GraphNode *node
);
```

Concludiamo, infine, presentando il sotto-modulo *common*, contenente una serie di logiche utilizzate dagli altri sotto-moduli. Andiamo a fare una panoramica:

- **`analyzer_get_identifier_ast_node_list`**: dato un *ASTNode*, estrae dal relativo *sottoalbero* la lista dei riferimenti agli *ASTNode* degli identificatori contenuti; si tratta di una funzione di esplorazione che utilizza l'algoritmo *DFS pre-order*;
- **`analyzer_check_array_access`**: dato un *ASTNode*, verifica che contenga un accesso ad un *array*;
- **`analyzer_check_member_access`**: dato un *ASTNode*, verifica che contenga un accesso al membro di una struttura;
- **`analyzer_check_pointer_access`**: dato un *ASTNode*, verifica che contenga un accesso ad un puntatore;
- **`analyzer_get_ast_node_type`**: dato un *ASTNode* ne restituisce, se esiste, il valore del relativo campo con chiave **`"type"`**;
- **`analyzer_get_ast_node_value`**: dato un *ASTNode* ne restituisce, se esiste, il valore del relativo campo con chiave **`"value"`**;
- **`analyzer_get_identifier_ast_node_value`**: dato un *ASTNode*, ricerca all'interno del sottoalbero l'*ASTNode* di un identificatore, e ne restituisce il valore del campo con chiave **`"value"`**.

Per una completa comprensione delle funzionalità, si invita alla lettura del codice sorgente della libreria, che si può trovare nei file **`mod_analyzer_common`**, **`mod_analyzer_common.c`**.

6.2.10 Algoritmi di esplorazione

L'*Analyzer* opera le proprie valutazioni, in ultima analisi, ispezionando l'*Abstract Syntax Tree*. Essendo l'*AST* un *albero*, e quindi un caso particolare di *grafo*, tali ispezioni vengono condotte utilizzando algoritmi di esplorazione. Viene fatto largamente uso degli algoritmi *DFS pre-order* e *DFS post-order*, e nello specifico delle loro versioni leggermente modificate *DFS pre-order find*, *DFS pre-order find first* e *DFS post-order find all*.

Si rimanda al paragrafo *3.2.4* per una presentazione dettagliata delle funzioni in questione. Ci limitiamo a ricordare che questi algoritmi modificano la logica di aggiunta dei *GraphNode* alla lista dei risultati, richiedendo infatti che venga anche rispettata una condizione di equivalenza, rispetto ad una lista di stringhe data, di un campo del nodo stesso. Questo permette di ricercare nell'*AST* solo alcune tipologie di costrutti, ad esempio solo i costrutti legati a identificatori, o a chiamate a funzione.

Tali algoritmi inoltre permettono, sotto determinate condizioni, di interrompere l'ispezione secondo logiche che andremo ad approfondire.

6.2.10.1 *DFS pre-order find*

L'algoritmo *DFS pre-order find* esegue un'ispezione del *grafo* secondo l'ordine di visita *DFS pre-order*. Quando viene trovato un *GraphNode* che rispetti la condizione di equivalenza sul campo, allora viene ignorata l'ispezione del relativo *sottoalbero*. Il funzionamento viene illustrato nell'immagine **Figura 20**.

Questa funzione è utile per ricercare i nodi di *primo livello*. Ad esempio, per valutare i termini che compaiono nella parte destra di un'espressione, ci interessa

ricercare gli identificatori e le chiamate a funzione di *primo livello*, in modo da escludere gli identificatori interni agli argomenti delle chiamate a funzione.

6.2.10.2 *DFS pre-order find first*

L'algoritmo *DFS pre-order find* esegue un'ispezione del *grafo* secondo l'ordine di visita *DFS pre-order*. Quando viene trovato un *GraphNode* che rispetti la condizione di equivalenza, questo viene restituito dalla funzione, interrompendo l'algoritmo. Il funzionamento viene illustrato nell'immagine **Figura 21**.

Questa funzione è utile per ricercare il primo elemento di un determinato tipo all'interno di un *sottoalbero*. Ad esempio, è necessaria quando si vuole ottenere il primo identificatore che compare nel *sottoalbero* di un nodo dell'*AST*.

6.2.10.3 *DFS post-order find all*

L'algoritmo *DFS pre-order find* esegue un'ispezione del *grafo* secondo l'ordine di visita *DFS post-order*. L'ordine di valutazione degli elementi non viene alterato rispetto all'algoritmo *standard*, ma vengono aggiunti al risultato solo i *GraphNode* che rispettino la condizione di equivalenza. Il funzionamento viene illustrato nell'immagine **Figura 22**.

Questa funzione permette di ricercare elementi ordinati rispettando l'ordine di esecuzione delle istruzioni. Questo è di fondamentale importanza nella logica dell'*Analyzer* per valutare, nel corretto ordine, ad esempio, le operazioni di assegnazione e di chiamata a funzione. Per chiarire ulteriormente il concetto facciamo il seguente esempio:

```
printf("%d", sum = a + b);
```

In tal caso, l'operazione di assegnazione a **`sum`** risulta avere la precedenza rispetto all'operazione di chiamata a funzione, nonostante **`printf`** compaia prima sia nel

codice sorgente, che nell'*AST*. Se l'*Analyzer* non rispettasse il corretto ordine di valutazione delle operazioni, nel seguente caso, verrebbe in generale etichettato con `ov` solamente l'identificatore `sum`, escludendo erroneamente `a` e `b`, perché le dipendenze con questi non sarebbero ancora state instaurate.

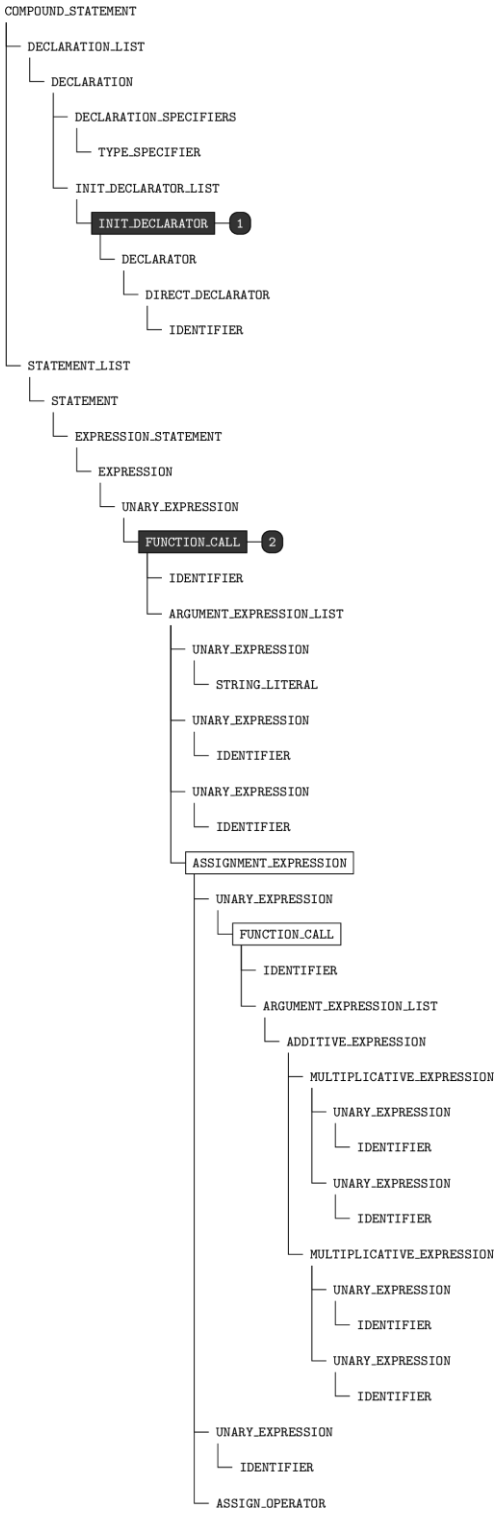

Figura 20, risultato dell'esecuzione dell'algoritmo *DFS pre-order find* su di un *AST*. Vengono ricercati **`INIT_DECLARATOR`**, **`FUNCTION_CALL`** e **`ASSIGNMENT_EXPRESSION`**.

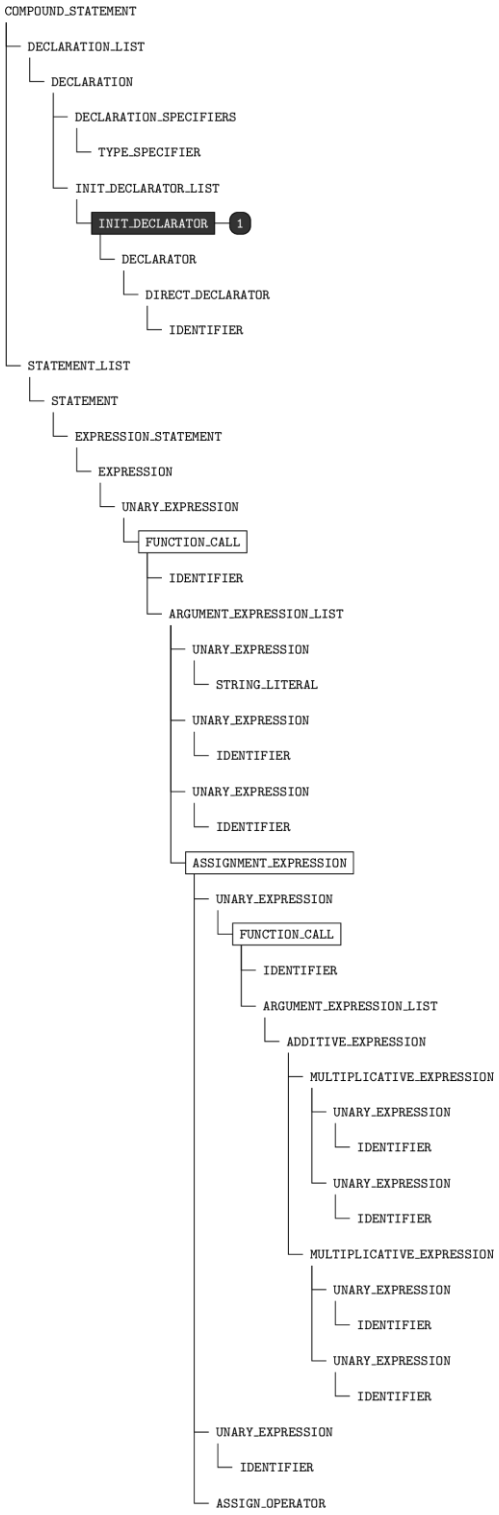

Figura 21, risultato dell'esecuzione dell'algoritmo *DFS pre-order find first* su di un *AST*. Vengono ricercati **`INIT_DECLARATOR`**, **`FUNCTION_CALL`** e **`ASSIGNMENT_EXPRESSION`**.

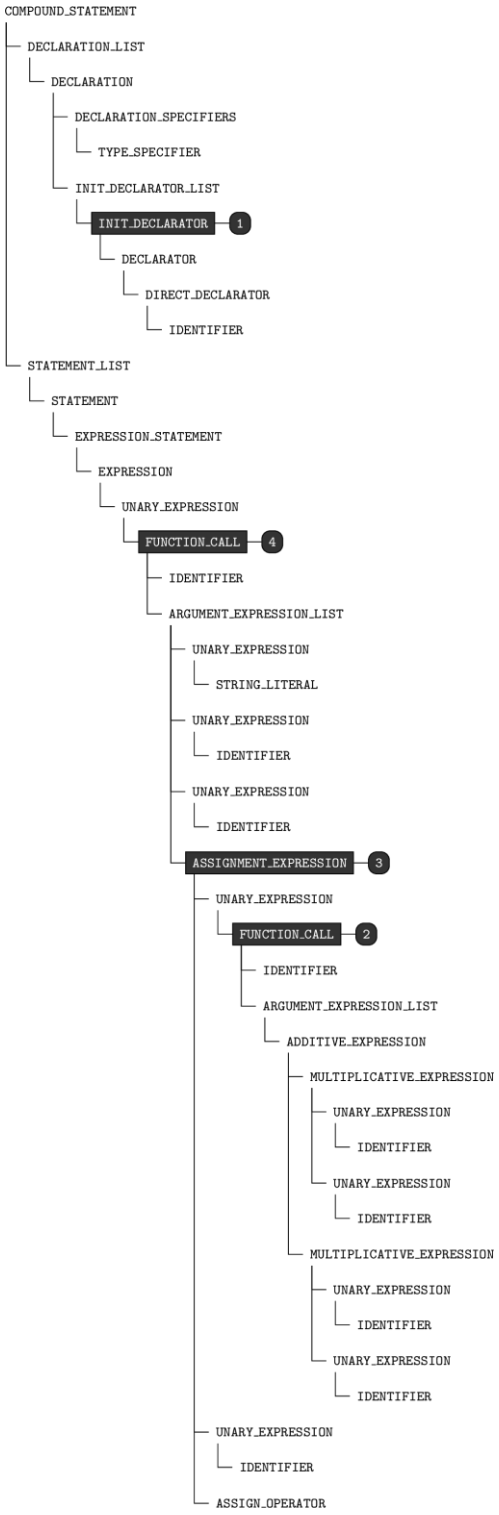

Figura 22, risultato dell'esecuzione dell'algoritmo *DFS post-order find all* su di un *AST*. Vengono ricercati **`INIT_DECLARATOR`**, **`FUNCTION_CALL`** e **`ASSIGNMENT_EXPRESSION`**.

Capitolo 7

Input e *Output*

Concludiamo la trattazione discutendo il funzionamento dell'applicativo in termini di *input* atteso e *output* generato.

7.1 *Input*

L'applicativo è utilizzabile da riga di comando e prevede un singolo argomento, ovvero il percorso di un file che rispetti lo standard *C90* e le limitazioni che abbiamo trattato nel Capitolo 2.

7.2 *Output*

L'*output* generato dal programma consiste in una serie di *log* ed eventuali *warning* stampati sul terminale. Questi messaggi includono:

- Avvertimenti della presenza di caratteri non riconosciuti nell'*input* individuati in fase di *analisi lessicale*;
- Avvertimenti della presenza di errori nel codice sorgente individuati in fase di *analisi sintattica*;
- *Log* che segnalano la presa in carico, da parte dell'*Analyzer*, della valutazione di una funzione;

- Avvertimenti della presenza di *destinazioni d'uso* non necessarie;
- Avvertimenti dell'assenza di *destinazioni d'uso* necessarie;
- Avvertimenti dell'assenza di documentazione funzionale;
- Avvertimenti sulla presenza di ricorsione.

Qualora non venga rilevato alcun errore nella documentazione funzionale, viene restituito un messaggio di successo.

7.3 Esempi

Alleghiamo ora alcuni esempi di esecuzione del programma. Nello specifico andremo ad includere esclusivamente file di *input* che contengano errori nella definizione della documentazione funzionale, perché più interessanti da trattare.

7.3.1 Esempio 1

7.3.1.1 *Input*

```
/**
 * Legge double da tastiera
 *
 * [param] IP + IK, result
 */
void leggi_double_da_tastiera(double *result) {
  scanf("%lf", result);
} /* leggiDouble */

/**
 * Stampa un double su file
 *
 * [param] IP + OV, etich
```

```
 * [param] IP, file
 * [param] IP, d
 */
void stampa_double_su_file(type_FILE *file, char etich[], double d) {
  printf("Printing %s on file\n", etich);
  fprintf(file, "%lf", d);
} /* leggiDouble */

/*
 * Riceve 2 double da tastiera e li stampa su file.
 *
 * [param] IP, etich1 - Etichetta per l'input 1 da tastiera
 * [param] IP, etich2 - Etichetta per l'input 1 da tastiera
 */
void stampa_2_double_su_file(const char etich1[], const char
etich2[]) {
  double *d1, *d2;

  leggi_double_da_tastiera(d1);
  leggi_double_da_tastiera(d2);

  stampa_double_su_file(file, etich1, d1);
  stampa_double_su_file(file, etich2, d2);
} /* leggi2Double */
```

7.3.1.2 *Output*

```
[LOG] Evaluating function leggi_double_da_tastiera.
[LOG] Evaluating function stampa_double_su_file.
[LOG] Evaluating function stampa_2_double_su_file.
[WARNING] Function 'stampa_2_double_su_file': destination OV for
parameter 'etich1' not found.
[WARNING] Function 'stampa_2_double_su_file': destination OV for
parameter 'etich2' not found.
[WARNING] Function 'stampa_double_su_file': destination OF for
parameter 'd' not found.
[WARNING] Function 'leggi_double_da_tastiera': destination OP for
parameter 'result' not found.
```

7.3.2 Esempio 2

7.3.2.1 *Input*

```
#include <stdio.h>
#include <stdbool.h>

/**
 * Sospende l'esecuzione finche' l'utente non preme un tasto
 */
void tasto(void) {
  printf("\nPremere un tasto qualsiasi per uscire.\n");
  fflush(stdin);
  getchar();
} /* tasto */

/**
 * Legge un intero da tastiera.
 *
 * [param] IP + OV, etich
 * [param] OR
 */
int leggiInt(const char etich[]) {
  int i;

  printf("%s", etich);
  scanf("%d", &i);

  return i;
} /* leggiInt */

/**
 * Legge un double da tastiera.
 *
 * [param] IP + OV, etich - Etichetta per l'input da tastiera
 * [param] OR
 */
double leggiDouble(const char etich[]) {
  double d;

  printf("%s", etich);
```

```
  scanf("%lf", &d);

  return d;
} /* leggiDouble */

/**
 * Legge un booleano da tastiera.
 *
 * [param] IP + OV, etich
 * [param] OR
 */
type_bool leggiBool(const char etich[]) {
  printf("[Usa: 0 per il valore false; 1 per true] ");

  return leggiInt(etich);
} /* leggiBool */

/*
 * Legge un carattere da tastiera.
 *
 * [param] IP + OV, etich - Etichetta per l'input da tastiera
 * [param] OR
 */
char leggiChar(const char etich[]) {
  char c;

  printf("%s", etich);
  fflush(stdin);
  scanf("%c", &c);

  return c;
} /* leggiChar */

/*
 * Legge due interi da tastiera
 *
 * [param] IP + OV, etich1 - Etichetta per l'input 1 da tastiera
 * [param] IP + OV, etich2 - Etichetta per l'input 1 da tastiera
 * [param] IOP, i1 - Int 1 da leggere
 * [param] IOP, i2 - Int 2 da leggere
 */
void leggi2Int(
  const char etich1[],
  const char etich2[],
```

```
  int* i1,
  int* i2
) {
  *i1 = leggiInt(etich1);
  *i2 = leggiInt(etich2);
} /* leggi2Int */

/*
 * Legge due double da tastiera.
 *
 * [param] IP + OV, etich1 - Etichetta per l'input 1 da tastiera
 * [param] IP + OV, etich2 - Etichetta per l'input 1 da tastiera
 * [param] IOP, d1 - Double 1 da leggere
 * [param] IOP, d2 - Double 2 da leggere
 */
void leggi2Double(
  const char etich1[],
  const char etich2[],
  double* d1,
  double* d2
) {
  *d1 = leggiDouble(etich1);
  *d2 = leggiDouble(etich2);
} /* leggi2Double */

/*
 * Legge una stringa da tastiera.
 *
 * [param] IP + OV, etich - Etichetta per l'input da tastiera
 * [param] IOP, s - Stringa da leggere
 */
void leggiStringa(const char etich[], char s[]) {
  printf("%s", etich);
  scanf("%s", s);
} /* leggiStringa */

/*
 * Legge due stringhe da tastiera.
 *
 * [param] IP, etich - Etichetta per l'input da tastiera
 * [param] IP, s1 - Stringa 1 da leggere
 * [param] IP + IK, s2 - Stringa 2 da leggere
 */
void leggi2Stringhe(const char etich[], char s1[], char s2[]) {
```

```
  printf("%s", etich);
  leggiStringa("Inserisci la prima stringa: ", s1);
  leggiStringa("Inserisci la seconda stringa: ", s2);
} /* leggiDati */

/*
 * Stampa un booleano su video.
 *
 * [param] IP + OV, etich - Etichetta per l'input da tastiera
 * [param] IP, b - Booleano da stampare
 */
void stampaBool(const char etich[], type_bool b) {
  printf("%s", etich);

  if (b) {
    printf("true");
  } else {
    printf("false");
  }

  printf("\n");
} /* stampaBool */

/*
 * [param] IP, nVia - Numero di viaggiatori
 * [param] IP, kmP - Contachilometri al momento della partenza
 * [param] IP, kmA - Contachilometri al momento dell'arrivo
 * [param] IP, kmLitro - Numero di km percorsi mediamente
 * [param] IP, costoCarbu - Costo di un litro di carburante
 * [param] OR - Contributo personale per il carburante
 */
double spesaPersonaCarbu(
  int nVia,
  int kmP,
  int kmA,
  double kmLitro,
  double costoCarbu
) {
  return (((kmA - kmP) / kmLitro) * costoCarbu) / nVia;
} /* spesaPersonaCarbu */

/*
 * [param] IP, m - Numero da valutare se e` multiplo
 * [param] IP, s - Numero da valutare se e` sottomultiplo
```

```
 * [param] OR - Indicazione se $m e` multiplo di $s
 */
type_bool isMultiplo(int m, int s) {
  return (m % s) == 0;
} /* isMultiplo */

/*
 * [param] IP, c - Carattere da valutare
 * [param] OR - Indicazione se $c e` una cifra decimale
*/
type_bool isDigit(char c) {
  return '0' <= c && c<='9';
} /* isDigit */

/*
 * [param] IP, c - Carattere da convertire in intero
 * [param] OR - Valore intero corrispondente a $c
*/
int digitToInt(char c) {
  return c - '0';
} /* digitToInt */

/*
 * [param] IP, n1 - Primo numero della serie
 * [param] IP, n2 - Secondo numero della serie
 * [param] IP, n3 - Terzo numero della serie
*/
type_bool isOrder(int n1, int n2, int n3) {
  return (n1 <= n2) && (n2 <= n3);
} /* isOrder */

/*
 * [param] IP, b - Base del rettangolo
 * [param] IP, h - Altezza del rettangolo
 * [param] IOP, p - Perimetro del rettangolo
 * [param] IOP, a - Area del rettangolo
*/
void rettangoPeriArea(double b, double h, double* p, double* a) {
  *p = 2 * (b + h);
  *a = b * h;
} /* rettangoPeriArea */
```

7.3.2.2 *Output*

```
[WARNING] Unrecognized character with ASCII code 96 found at line
168, column 43.
[WARNING] Unrecognized character with ASCII code 96 found at line
169, column 43.
[WARNING] Unrecognized character with ASCII code 36 found at line
170, column 32.
[WARNING] Unrecognized character with ASCII code 96 found at line
170, column 36.
[WARNING] Unrecognized character with ASCII code 36 found at line
170, column 50.
[WARNING] Unrecognized character with ASCII code 36 found at line
178, column 32.
[WARNING] Unrecognized character with ASCII code 96 found at line
178, column 36.
[WARNING] Unrecognized character with ASCII code 36 found at line
186, column 48.
[LOG] Evaluating function leggiStringa.
[LOG] Evaluating function leggi2Stringhe.
[LOG] Evaluating function spesaPersonaCarbu.
[LOG] Evaluating function isOrder.
[LOG] Evaluating function digitToInt.
[LOG] Evaluating function tasto.
[LOG] Evaluating function isDigit.
[LOG] Evaluating function leggiDouble.
[LOG] Evaluating function stampaBool.
[LOG] Evaluating function isMultiplo.
[LOG] Evaluating function leggi2Double.
[LOG] Evaluating function leggiInt.
[LOG] Evaluating function leggiBool.
[LOG] Evaluating function leggi2Int.
[LOG] Evaluating function leggiChar.
[LOG] Evaluating function rettangoPeriArea.
[WARNING] Function 'leggi2Stringhe': destination IK for parameter
's1' not found.
[WARNING] Function 'leggi2Stringhe': destination OP for parameter
's1' not found.
[WARNING] Function 'leggi2Stringhe': destination OP for parameter
's2' not found.
[WARNING] Function 'leggi2Stringhe': destination OV for parameter
'etich' not found.
[WARNING] Function 'isOrder': destination OR not found.
```

```
[WARNING] Function 'leggiStringa': destination IK for parameter 's'
not found.
```

7.3.3 Esempio 3

7.3.3.1 *Input*

```
#include <stdio.h>
#include <stdbool.h>
#include <stdlib.h>
#include <assert.h>

typedef struct type_data {
  int g, m, a;
} type_data;

typedef struct {
  int votiOttenuti;
  type_bool eletto;
} type_candidato;

typedef struct {
  char* nome;
  int votiOttenuti;
  int numeroCandidati;
  int numeroConsiglieriAttribuiti;
  type_candidato* candidati;
} type_lista;

typedef struct {
  type_data dataVotazione;
  int numeroConsiglieri;
  int numeroListe;
  type_lista* liste;
} type_elezione;

typedef struct {
  float valore;
  type_bool massimo;
} type_resto;
```

```
/**
 * Sospende l'esecuzione finche' l'utente non preme un tasto
 */
void tasto(void) {
  printf("\nPremere un tasto qualsiasi per uscire.\n");
  fflush(stdin);
  getchar();
} /* tasto */

/**
 * Legge un intero da tastiera.
 *
 * [param] IP + OV, etich
 * [param] OR
 */
int leggiInt(const char etich[]) {
  int i;

  printf("%s", etich);
  scanf("%d", &i);

  return i;
} /* leggiInt */

/**
 * Legge un double da tastiera.
 *
 * [param] IP + OV, etich - Etichetta per l'input da  tastiera
 * [param] OR
 */
double leggiDouble(const char etich[]) {
  double d;

  printf("%s", etich);
  scanf("%lf", &d);

  return d;
} /* leggiDouble */

/**
 * Legge un booleano da tastiera.
 *
 * [param] IP + OV, etich
```

```
 * [param] OR
 */
type_bool leggiBool(const char etich[]) {
  printf("[Usa: 0 per il valore false; 1 per true] ");

  return leggiInt(etich);
} /* leggiBool */

/*
 * Legge un carattere da tastiera.
 *
 * [param] IP + OV, etich - Etichetta per l'input da  tastiera
 * [param] OR
 */
char leggiChar(const char etich[]) {
  char c;

  printf("%s", etich);
  fflush(stdin);
  scanf("%c", &c);

  return c;
} /* leggiChar */

/*
 * Legge due interi da tastiera
 *
 * [param] IP + OV, etich1 - Etichetta per l'input 1 da  tastiera
 * [param] IP + OV, etich2 - Etichetta per l'input 1 da  tastiera
 * [param] IOP, i1 - Int 1 da leggere
 * [param] IOP, i2 - Int 2 da leggere
 */
void leggi2Int(
  const char etich1[],
  const char etich2[],
  int* i1,
  int* i2
) {
  *i1 = leggiInt(etich1);
  *i2 = leggiInt(etich2);
} /* leggi2Int */

/*
 * Legge due double da tastiera.
```

```
 *
 * [param] IP + OV, etich1 - Etichetta per l'input 1 da  tastiera
 * [param] IP + OV, etich2 - Etichetta per l'input 1 da  tastiera
 * [param] IOP, d1 - Double 1 da leggere
 * [param] IOP, d2 - Double 2 da leggere
 */
void leggi2Double(
  const char etich1[],
  const char etich2[],
  double* d1,
  double* d2
) {
  *d1 = leggiDouble(etich1);
  *d2 = leggiDouble(etich2);
} /* leggi2Double */

/*
 * Legge una stringa da tastiera.
 *
 * [param] IP + OV, etich - Etichetta per l'input da  tastiera
 * [param] IOP + IK, s - Stringa da leggere
 */
void leggiStringa(const char etich[], char s[]) {
  printf("%s", etich);
  scanf("%s", s);
} /* leggiStringa */

/*
 * Legge due stringhe da tastiera.
 *
 * [param] IP + OV, etich - Etichetta per l'input da  tastiera
 * [param] IP + IK, s1 - Stringa 1 da leggere
 * [param] IP + IK, s2 - Stringa 2 da leggere
 */
void leggi2Stringhe(const char etich[], char s1[], char s2[]) {
  printf("%s", etich);
  leggiStringa("Inserisci la prima stringa: ", s1);
  leggiStringa("Inserisci la seconda stringa: ", s2);
} /* leggiDati */

/*
 * Stampa un booleano su video.
 *
 * [param] IP + OV, etich - Etichetta per l'input da  tastiera
```

```
 * [param] IP, b - Booleano da stampare
 */
void stampaBool(const char etich[], type_bool b) {
  printf("%s", etich);

  if (b) {
  printf("true");
  } else {
  printf("false");
  }

  printf("\n");
} /* stampaBool */

/*
 * Legge una Data da tastiera.
 *
 * [param] IP + OV, etich - Etichetta per l'input da  tastiera.
 * [param] IOP, d - Data letta da tastiera.
 */
void scanData(const char etich[], type_data* d) {
  printf("%s", etich);
  d->g = leggiInt("Giorno: ");
  d->m = leggiInt("Mese: ");
  d->a = leggiInt("Anno: ");
} /* scanData */

/*
 * Alloca e legge un array pieno di Data da tastiera.
 *
 * [param] IP + OV, etiche - Etichetta per l'input da  tastiera.
 * [param] IOP, n - Grandezza di $a.
 * [param] OR
 */
type_data* allocateScanArrayData(const char etiche[], int* n) {
  type_data* a;
  int i;
  printf("%s", etiche);
  *n = leggiInt("Numero di date: ");
  a = malloc(*n * sizeof(int));
  assert(a != NULL);
  for (i = 0; i < *n; i++) {
    printf("Data[%d]: ", i);
    scanData("", &a[i]);
```

```
  } /* for */
  return a;
} /* allocateScanArrayData */

/*
 * Stampa una Data su video.
 *
 * [param] IP + OV, etich - Etichetta per l'output a video.
 * [param] IP + OV, d - Data da stampare.
 */
void printData(const char etich[], const type_data* d) {
  printf("%s", etich);
  printf("{ %d / %d / %d }\n", d->g, d->m, d->a);
}  /* printData */

/*
 * Stampa un array di Data su video.
 *
 * [param] IP + OV, etich - Etichetta per l'output a video.
 * [param] IP + OV, a - Array da stampare.
 * [param] IP, n - Grandezza di $a.
 */
void printArrayData(const char etich[], const type_data a[], int n) {
  int i;
  printf("%s", etich);
  for (i= 0; i< n; i++) {
    printf("a[%d]: ", i);
    printData("", &a[i]);
  } /* for */
} /* printArrayData */

/**
 * Stampa un messaggio di errore.
 *
 * [param] IP + OV, msg - Messaggio da stampare a video
 */
void stampaErrore(const char msg[]) {
  printf("\n[ERRORE] %s\n\n", msg);
} /* stampaErrore */

/**
 * [param] IP + OV, msg - Messaggio da stampare a video
 * [param] OR - Stringa di lunghezza dinamica inserita
 */
```

```
char* leggiStringaLunghezzaDinamica(const char msg[]) {
  int bufferSize = 10,
      bufferCount = 1,
      length = 0;

  char current;

  char* input = malloc(bufferSize * bufferCount);
  assert(input != NULL);

  printf("%s", msg);

  while((current = getchar()) == '\n');

  /* Ciclo eseguito fintantoché l'utente non "manda a capo" */
  do {
    if (length >= bufferSize * bufferCount) {
      bufferCount += 1;
      input = realloc(
        input, sizeof(char) * bufferCount * bufferSize
      );
    } /* if */

    input[length++] = current;
  } while ((current = getchar()) != '\n'); /* do-while */

  input = realloc(input, sizeof(char) * length);

  return input;
} /* leggiStringaLunghezzaDinamica */

/**
 * Legge un intero positivo.
 *
 * [param] IP + OV, msg - Stringa da stampare a video
 * [param] OR - Intero inserito da tastiera
 */
int leggiIntPositivo(const char msg[]) {
  int i;

  while((i = leggiInt(msg)) < 0) {
    stampaErrore("Il valore inserito non e' valido");
  } /* while */
```

```
  return i;
} /* leggiIntPositivo */

/**
 * Legge un intero strettamente positivo.
 *
 * [param] IP + OV, msg - Stringa da stampare a video
 * [param] OR - Intero inserito da tastiera
 */
int leggiIntStrettamentePositivo(const char msg[]) {
  int i;

  while((i = leggiInt(msg)) <= 0) {
    stampaErrore("Il valore inserito non e' valido");
  } /* while */

  return i;
} /* leggiIntStrettamentePositivo */

/**
 * [param] IP, data - Data da validare
 * [param] OR - Booleano che determina se la data sia o meno valida.
 */
type_bool validaData(const type_data data) {
  if (data.g <= 0 ||
      data.m <= 0 ||
      data.m > 12) {
    return false;
  } /* if */

  switch (data.m) {
    case 1:   /* Gennaio */
    case 3:   /* Marzo */
    case 5:   /* Maggio */
    case 7:   /* Luglio */
    case 8:   /* Agosto */
    case 10:  /* Ottobre */
    case 12:  /* Dicembre */
      return data.g <= 31;

    case 4:   /* Aprile */
    case 6:   /* Giugno */
    case 9:   /* Settembre */
    case 11:  /* Novembre */
```

```
      return data.g <= 30;

    case 2:   /* Febbraio */
      if (data.a % 4 == 0) {
        return data.g <= 29; /* Anno bisestile */
      } /* if */
      else {
        return data.g <= 28; /* Anno non bisestile */
      } /* else */
  } /* switch */

  return false;
} /* validaData */

/**
 * Legge una data, e garantisce che sia valida.
 *
 * [param] OR - Data inserita dall'utente.
 */
type_data leggiData() {
  type_data data;

  type_bool dataValida;

  scanData("DATA DELLA VOTAZIONE\n", &data);

  while(!(dataValida = validaData(data))) {
    stampaErrore("La data inserita non e' valida.");
    scanData("DATA DELLA VOTAZIONE\n", &data);
  } /* while */

  printf("\n");

  return data;
} /* leggiData */

/**
 * Data una lista, legge quanti voti ha ottenuto ogni suo candidato.
 *
 * [param] IOP, li
 * [param] OR
 */
void leggiCandidatiLista(type_lista * li) {
  int i;
```

```
  li->candidati = malloc(
    sizeof(type_candidato) * li->numeroCandidati
  );

  assert(li->candidati != NULL);

  for (i = 0; i < li->numeroCandidati; i++) {
    printf("| CANDIDATO %d\n", i);

    li->candidati[i].votiOttenuti = leggiIntPositivo(
      "| | Voti ottenuti dal candidato: "
    );
  } /* for */
} /* leggiCandidatiLista */

/**
 * [param] IOP, ele
 */
void leggiListe(type_elezione* ele) {
  int i;

  ele->liste = malloc(sizeof(type_lista) * ele->numeroListe);
  assert(ele->liste != NULL);

  for (i = 0; i < ele->numeroListe; i++) {
    printf("\nLISTA %d\n", i);

    ele->liste[i].nome = leggiStringaLunghezzaDinamica(
      "| Nome della lista: "
    );

    ele->liste[i].votiOttenuti = leggiIntPositivo(
      "| Voti ottenuti dalla lista: "
    );

    ele->liste[i].numeroCandidati = leggiIntStrettamentePositivo(
      "| Numero di candidati: "
    );

    leggiCandidatiLista(&ele->liste[i]);
  } /* for */

  printf("\n");
```

```
} /* leggiListe */

/**
 * [param] IOP, ele
 */
void leggiElezione(type_elezione *ele) {
  printf("--- INSERIMENTO DATI ELEZIONE ---\n\n");

  ele->dataVotazione = leggiData();

  ele->numeroConsiglieri = leggiIntStrettamentePositivo(
    "Numero dei consiglieri da eleggere: "
  );

  ele->numeroListe = leggiIntStrettamentePositivo(
    "Numero delle liste: "
  );

  leggiListe(ele);
} /* leggiElezione */

/**
 * [param] IP, ele
 * [param] OR - Valore del quorum calcolato
 */
float calcolaQuorum(const type_elezione ele) {
  int i, votiTotali = 0;

  for (i = 0; i < ele.numeroListe; i++) {
    votiTotali += ele.liste[i].votiOttenuti;
  } /* for */

  return (float) votiTotali / ele.numeroConsiglieri;
} /* calcolaQuorum */

/**
 * [param] IP, numeroListe - Numero delle liste.
 * [param] IP, resti
 */
void calcolaProssimoRestoMassimo(
  int numeroListe,
  type_resto* resti
) {
  int i, maxRestoIndex;
```

```
  float maxResto = 0;

  for (i = 0; i < numeroListe; i++) {
    if (resti[i].massimo == false && maxResto < resti[i].valore) {
      maxResto = resti[i].valore;
      maxRestoIndex = i;
    } /* if */
  } /* for */

  resti[maxRestoIndex].massimo = true;
} /* calcolaProssimoRestoMassimo */

/**
 * [param] IOP, resti
 * [param] IP, ele
 */
void calcolaRestiListe(
  type_resto* resti,
  const type_elezione ele,
  float quorum
) {
  int i;

  for (i = 0; i < ele.numeroListe; i++) {
    resti[i].massimo = false;
    resti[i].valore += (float) ele.liste[i].votiOttenuti
      - quorum * ele.liste[i].numeroConsiglieriAttribuiti;
  } /* for */
} /* calcolaRestiListe */

/**
 * [param] IOP, ele
 * [param] IP, quorum - Valore del quorum.
 * [param] OR - Numero dei consiglieri rimanenti
 */
int calcolaConsiglieriAttribuitiInizialmente(
  type_elezione* ele, float quorum) {
  int i, consiglieriRimanenti = ele->numeroConsiglieri;

  for (i = 0; i < ele->numeroListe; i++) {
    /*
      Si potrebbe non utilizzare la variabile $consiglieriAttribuiti,
      ma preferisco farlo a favore della leggibilità del codice.
```

```
    */

    int consiglieriAttribuiti = ele->liste[i].votiOttenuti / quorum;

    ele->liste[i].numeroConsiglieriAttribuiti =
      consiglieriAttribuiti;

    consiglieriRimanenti -= consiglieriAttribuiti;
  } /* for */

  return consiglieriRimanenti;
} /* calcolaConsiglieriAttribuitiInizialmente */

/**
 * [param] IOP, ele
 */
void attribuisciConsiglieri(type_elezione* ele) {
  int i, consiglieriRimanenti;

  float quorum;

  type_resto * resti = malloc(
    sizeof(type_resto ) * ele->numeroListe
  );

  assert(resti != NULL);

  /* Calcolo del quorum */

  quorum = calcolaQuorum(*ele);

  /*
    Calcolo dei consiglieri attribuiti inizialmente, e quindi dei
    consiglieri rimanenti.
  */

  consiglieriRimanenti = calcolaConsiglieriAttribuitiInizialmente(
    ele, quorum
  );

  /* Calcolo dei resti delle liste, e quindi dei resti massimi */

  calcolaRestiListe(resti, *ele, quorum);
```

```
  for (i = 0; i < consiglieriRimanenti; i++) {
    calcolaProssimoRestoMassimo(ele->numeroListe, resti);
  } /* for */

  /* Completamento del calcolo dei consiglieri attribuiti */

  for (i = 0; i < ele->numeroListe; i++) {
    if (resti[i].massimo) {
      ele->liste[i].numeroConsiglieriAttribuiti += 1;
    } /* if */
  } /* for */

  /* Deallocazione della memoria */

  free(resti);
} /* attribuisciConsiglieri */

/**
 * [param] IOP, li
 */
void trovaProssimoCandidatoElettoLista(type_lista* li) {
  int i;

  int maxVotiCandidato = 0, maxVotiCandidatoIndex;

  for (i = 0; i < li->numeroCandidati; i++) {
    if (
      li->candidati[i].eletto == false &&
      maxVotiCandidato < li->candidati[i].votiOttenuti
    ) {
      maxVotiCandidato = li->candidati[i].votiOttenuti;
      maxVotiCandidatoIndex = i;
    } /* if */
  } /* for */

  li->candidati[maxVotiCandidatoIndex].eletto = true;
} /* trovaProssimoCandidatoElettoLista */

/**
 * [param] IP + IK, li
 */
void trovaCandidatiElettiLista(type_lista* li) {
  int i;
```

```
  for (i = 0; i < li->numeroConsiglieriAttribuiti; i++) {
    trovaProssimoCandidatoElettoLista(li);
  } /* for */
} /* trovaCandidatiElettiLista */

/**
 * Individua per ogni lista i candidati eletti.
 *
 * [param] IP, ele
 */
void trovaEletti(type_elezione * ele) {
  int i;

  attribuisciConsiglieri(ele);

  for (i = 0; i < ele->numeroListe; i++) {
    trovaCandidatiElettiLista(&ele->liste[i]);
  } /* for */
} /* trovaEletti */

/**
 * [param] IP + OV, li - Lista da stampare.
 * [param] IP, filtraEletti
 */
void stampaCandidatiLista(
  const type_lista li,
  type_bool filtraEletti
) {
  int i;

  for (i = 0; i < li.numeroCandidati; i++) {
    if (!filtraEletti || li.candidati[i].eletto) {
      printf(
        "Voti candidato %d: %d\n",
        i,
        li.candidati[i].votiOttenuti
      );
    } /* if */
  } /* for */
} /* stampaCandidatiLista */

/**
 * [param] IP + OV, ele - Elezione da stampare.
 */
```

```
void stampaElezione(const type_elezione ele) {
  int i;

  printf("ELEZIONE\n\n");

  printf(
    "Data della votazione: %d/%d/%d\n",
    ele.dataVotazione.g,
    ele.dataVotazione.m,
    ele.dataVotazione.a
  );

  printf(
    "Numero di consiglieri da eleggere: %d\n\n",
    ele.numeroConsiglieri
  );

  for (i = 0; i < ele.numeroListe; i++) {
    printf(
      "LISTA %s. Voti: %d. Numero candidati: %d\n",
      ele.liste[i].nome,
      ele.liste[i].votiOttenuti,
      ele.liste[i].numeroCandidati
    );

    stampaCandidatiLista(ele.liste[i], false);

    printf("\n");
  } /* for */
} /* stampaEletti */

/**
 * [param] IP, ele
 */
void stampaEletti(const type_elezione ele) {
  int i;

  printf("CONSIGLIERI ELETTI\n\n");

  for (i = 0; i < ele.numeroListe; i++) {
    printf(
      "LISTA %s. Voti: %d. Consiglieri attribuiti: %d\n",
      ele.liste[i].nome,
      ele.liste[i].votiOttenuti,
```

```
      ele.liste[i].numeroConsiglieriAttribuiti
    );

    stampaCandidatiLista(ele.liste[i], true);

    printf("\n");
  } /* for */
} /* stampaEletti */

/**
 * [param] IP, ele
 */
void deallocaElezione(type elezione* ele) {
  int i;

  for (i = 0; i < ele->numeroListe; i++) {
    free(ele->liste[i].nome);
    free(ele->liste[i].candidati);
  } /* for */

  free(ele->liste);
} /* deallocaElezione */
```

7.3.3.2 *Output*

```
[WARNING] Unrecognized character with ASCII code 36 found at line
197, column 34.
[WARNING] Unrecognized character with ASCII code 36 found at line
230, column 33.
[WARNING] Unrecognized character with ASCII code 195 found at line
268, column 31.
[WARNING] Unrecognized character with ASCII code 169 found at line
268, column 32.
[WARNING] Unrecognized character with ASCII code 36 found at line
516, column 47.
[WARNING] Unrecognized character with ASCII code 195 found at line
517, column 52.
[WARNING] Unrecognized character with ASCII code 160 found at line
517, column 53.
[LOG] Evaluating function leggiStringa.
[LOG] Evaluating function leggi2Stringhe.
[LOG] Evaluating function leggiInt.
```

```
[LOG] Evaluating function stampaErrore.
[LOG] Evaluating function leggiIntPositivo.
[LOG] Evaluating function stampaCandidatiLista.
[LOG] Evaluating function deallocaElezione.
[LOG] Evaluating function leggiStringaLunghezzaDinamica.
[LOG] Evaluating function validaData.
[LOG] Evaluating function stampaEletti.
[LOG] Evaluating function leggiCandidatiLista.
[LOG] Evaluating function stampaElezione.
[LOG] Evaluating function leggiIntStrettamentePositivo.
[LOG] Evaluating function calcolaQuorum.
[LOG] Evaluating function calcolaConsiglieriAttribuitiInizialmente.
[LOG] Evaluating function calcolaRestiListe.
[LOG] Evaluating function calcolaProssimoRestoMassimo.
[LOG] Evaluating function attribuisciConsiglieri.
[LOG] Evaluating function tasto.
[LOG] Evaluating function scanData.
[LOG] Evaluating function leggiData.
[LOG] Evaluating function leggiListe.
[LOG] Evaluating function leggiElezione.
[LOG] Evaluating function leggiDouble.
[LOG] Evaluating function stampaBool.
[LOG] Evaluating function printData.
[LOG] Evaluating function printArrayData.
[LOG] Evaluating function trovaProssimoCandidatoElettoLista.
[LOG] Evaluating function trovaCandidatiElettiLista.
[LOG] Evaluating function trovaEletti.
[LOG] Evaluating function leggi2Double.
[LOG] Evaluating function leggiBool.
[LOG] Evaluating function leggi2Int.
[LOG] Evaluating function leggiChar.
[LOG] Evaluating function allocateScanArrayData.
[WARNING] Function 'leggi2Stringhe': destination OP for parameter
's1' not found.
[WARNING] Function 'leggi2Stringhe': destination OP for parameter
's2' not found.
[WARNING] Function 'stampaEletti': destination OV for parameter 'ele'
not found.
[WARNING] Function 'leggiCandidatiLista': destination OR unnecessary.
[WARNING] Function 'calcolaRestiListe': no documentation found for
parameter with identifier 'quorum'.
[WARNING] Function 'calcolaProssimoRestoMassimo': destination OP for
parameter 'resti' not found.
```

```
[WARNING] Function 'trovaCandidatiElettiLista': destination IK for
parameter 'li' unnecessary.
[WARNING] Function 'trovaCandidatiElettiLista': destination OP for
parameter 'li' not found.
[WARNING] Function 'trovaEletti': destination OP for parameter 'ele'
not found.
```

7.3.4 Esempio 4

7.3.4.1 *Input*

```
/**
 * Legge double da tastiera
 *
 * [param] IOP + IK, result
 */
void leggi_double_da_tastiera(double *result) {
  scanf("%lf", result);
} /* leggiDouble */

/**
 * Stampa un double su file
 *
 * [param] IP + OV, etich
 * [param] IP, file
 * [param] IP + OF, d
 */
void stampa_double_su_file(type_FILE *file, char etich[], double d) {
  printf("Printing %s on file\n", etich);
  fprintf(file, "%lf", d);

  stampa_2_double_su_file(etich, etich); /* Recursive call */
} /* leggiDouble */

/*
 * Riceve 2 double da tastiera e li stampa su file.
 *
 * [param] IP + OV, etich1 - Etichetta per l'input 1 da tastiera
 * [param] IP + OV, etich2 - Etichetta per l'input 1 da tastiera
 */
```

```
void stampa_2_double_su_file(
  const char etich1[],
  const char etich2[]
) {
  double *d1, *d2;

  leggi_double_da_tastiera(d1);
  leggi_double_da_tastiera(d2);

  stampa_double_su_file(file, etich1, d1);
  stampa_double_su_file(file, etich2, d2);
} /* leggi2Double */
```

7.3.4.2 *Output*

```
[ERROR] Recursive call chains detected.
Call chain: stampa_double_su_file stampa_2_double_su_file
```

Capitolo 8

Codice sorgente

8.1 Librerie funzionali

8.1.1 `lib_buffer`

8.1.1.1 `lib_buffer.h`

```
#ifndef LIB_BUFFER_H
#define LIB_BUFFER_H

typedef struct Buffer {
  char *value;
  unsigned long size;
  unsigned long length;
} Buffer;

#endif

Buffer  *buffer_create(void);
void    buffer_append_character(Buffer *buffer, char character);
void    buffer_append_integer(Buffer *buffer, int integer);
void    buffer_append_string(Buffer *buffer, const char *string);
void    buffer_free(Buffer *buffer);
void    buffer_reset(Buffer *buffer);
```

8.1.1.2 `lib_buffer.c`

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "lib_buffer.h"

#define INITIAL_BUFFER_SIZE 8

/********** RESERVED FUNCTION DECLARATIONS **********/

void buffer_scale(Buffer *buffer, unsigned long length);

/********** LIBRARY FUNCTIONS **********/

/**
 * buffer_create
 *
 * Function to create a Buffer.
 *
 * [param] OR - the newly created Buffer
 */
Buffer *buffer_create(void) {
  Buffer *buffer = malloc(sizeof(Buffer));
  assert(buffer != NULL);

  buffer->length = 0;
  buffer->size = INITIAL_BUFFER_SIZE;

  buffer->value = malloc(sizeof(char) * buffer->size);
  assert(buffer->value != NULL);

  buffer->value[0] = '\0';

  return buffer;
}

/**
 * buffer_append_character
 *
```

```
 * Function to append a single character to a Buffer.
 *
 * [param] IOP, buffer
 * [param] IP, character
 */
void buffer_append_character(Buffer *buffer, char character) {
  buffer_scale(buffer, 1);

  buffer->value[buffer->length] = character;
  buffer->value[buffer->length + 1] = '\0';
  buffer->length += 1;
}

/**
 * buffer_append_integer
 *
 * Function to append an integer to a Buffer.
 *
 * [param] IOP, buffer
 * [param] IP, integer
 */
void buffer_append_integer(Buffer *buffer, int integer) {
  char *string = malloc(
    sizeof(char) * (int) (ceil(log10(integer)) + 2)
  );

  assert(string != NULL);

  sprintf(string, "%d", integer);

  buffer_append_string(buffer, string);

  free(string);
}

/**
 * buffer_append_string
 *
 * Function to append a string to a Buffer.
 *
 * [param] IOP, buffer
 * [param] IP, string
 */
void buffer_append_string(Buffer *buffer, const char *string) {
```

```
  const char *current;

  buffer_scale(buffer, strlen(string));

  for (current = string; *current != '\0'; current += 1) {
    buffer->value[buffer->length] = *current;
    buffer->length += 1;
  }

  buffer->value[buffer->length] = '\0';
}

/**
 * buffer_reset
 *
 * Function to reset the content and the size of a Buffer.
 *
 * [param] IOP, buffer
 */
void buffer_reset(Buffer *buffer) {
  buffer->length = 0;
  buffer->size = INITIAL_BUFFER_SIZE;

  free(buffer->value);

  buffer->value = malloc(sizeof(char) * buffer->size);
  assert(buffer->value != NULL);

  buffer->value[0] = '\0';
}

/********** CLEAN-UP FUNCTIONS **********/

/**
 * buffer_free
 *
 * Function to free the memory used by a Buffer.
 *
 * [param] IOP, buffer
 */
void buffer_free(Buffer *buffer) {
  free(buffer->value);
  buffer->value = NULL;
```

```
  free(buffer);
  buffer = NULL;
}

/********** RESERVED FUNCTIONS **********/

/**
 * buffer_scale
 *
 * Function to scale up a Buffer if needed.
 *
 * [param] IOP, buffer
 * [param] IP, length
 */
void buffer_scale(Buffer *buffer, unsigned long length) {
  unsigned total length = buffer->length + length;

  if (buffer->size > total_length) {
    return;
  }

  while (buffer->size <= total_length) {
    buffer->size *= 2;
  }

  buffer->value = realloc(
    buffer->value,
    sizeof(char) * (buffer->size + 1)
  );

  assert(buffer->value != NULL);
}
```

8.1.2 `lib_common`

8.1.2.1 `lib_common.h`

```
#include <stdbool.h>

#include "lib_list.h"

bool list_contains_string(const List *list, const char *string);
bool validate_character(int character);
char* get_pointer_address(const void *pointer);
```

8.1.2.2 `lib_common.c`

```
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "lib_common.h"

#define POINTER_ADDRESS_LENGTH 10

/********** LIBRARY FUNCTIONS *********/

/**
 *  list_contains_string
 *
 *  Checks whether a string is contained in a list.
 *
 *  [param] IP, list
 *  [param] IP, string
 *  [param] OR - the resulting boolean value
 */
bool list_contains_string(const List *list, const char *string) {
  ListNode *iterator;
```

```
  for (
    iterator = list_get_iterator(list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    if (strcmp(iterator->value, string) == 0) {
      return true;
    }
  }

  return false;
}

/**
 * validate_character
 *
 * Checks whether a character is an ANSI-C valid one.
 *
 * [param] IP, character
 * [param] OR - the resulting boolean value
 */
bool validate_character(int character) {
  return
    character == -1
    || (character >= 9 && character <= 13)
    || (character >= 32 && character <= 35)
    || (character >= 37 && character <= 63)
    || (character >= 65 && character <= 95)
    || (character >= 97 && character <= 126);
}

/*
 * get_pointer_address
 *
 * Returns a string containing the address of a pointer.
 *
 * [param] IP, pointer
 * [param] OR - a string containing the address
 */
char* get_pointer_address(const void *pointer) {
  char *address = malloc(sizeof(char) * POINTER_ADDRESS_LENGTH + 1);
  assert(address != NULL);

  sprintf(address, "%p", pointer);
```

```
  return address;
}
```

8.1.3 `lib_dfa`

8.1.3.1 `lib_dfa.h`

```
#include <stdio.h>

#include "lib_graph.h"

#ifndef LIB_DFA_H
#define LIB_DFA_H

typedef struct DFA {
  Graph *graph;
  GraphNode *state;
  FILE *source;
  int character;
  int line;
  int column;
} DFA;

typedef struct DFAMatch {
  int type;
  char *value;
  int line;
  int column;
} DFAMatch;

typedef GraphNode DFAState;
typedef List DFAStateGroup;

#endif

DFA *dfa_create(FILE *source);

DFAMatch *dfa_get_next_match(DFA *dfa);

DFAState *dfa_add_root(const DFA *dfa, const char *label);

DFAState *dfa_add_state(const DFA *dfa, const char *label);
```

```
DFAState *dfa_add_state_epsilon(
  const DFA *dfa,
  GraphNode *state,
  const char *prefix
);

DFAState *dfa_add_state_group_epsilon(
  const DFA *dfa,
  DFAStateGroup *state,
  const char *prefix
);

DFAStateGroup *dfa add state group(
  const DFA *dfa,
  unsigned int count,
  const char *prefix
);

int *dfa_string_to_transitions(const char *string);

void dfa_epsilon_transition_state_group_to_state(
  List *states_a,
  GraphNode *state_b
);

void dfa_epsilon_transition_state_to_state(
  GraphNode *state_a,
  GraphNode *state_b
);

void dfa_free(DFA *dfa);

void dfa_match_free(DFAMatch *match);

void dfa_set_leaf(const GraphNode *state, int token);

void dfa_set_leaves(const List *states, int token);

void dfa_transition_state_group_to_state(
  const DFAStateGroup *state_a,
  DFAState *state_b,
  int transition_value
);
```

```
void dfa_transition_state_group_to_state_group(
  const DFAStateGroup *state_a,
  DFAStateGroup *state_b,
  int transitions_count,
  int *transition_values
);

void dfa_transition_state_to_state(
  DFAState *state_a,
  DFAState *state_b,
  int transition_value
);

void dfa_transition_state_to_state_group(
  DFAState *state_a,
  const DFAStateGroup *states b,
  int transitions_count,
  int *transition_values
);
```

8.1.3.2 `lib_dfa.c`

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "lib buffer.h"
#include "lib_common.h"
#include "lib_dfa.h"

#define TRANSITION "TRANSITION"
#define FLUSH "FLUSH"
#define ROOT "ROOT"
#define MATCH "MATCH"
#define TRANSITION_EPSILON_VALUE 0

/********** RESERVED FUNCTION DECLARATIONS **********/

DFAMatch *dfa_match_create(
```

```
  int type,
  const char *value,
  int line,
  int column
);

DFAState *dfa_get_child_epsilon(const DFA *dfa);

DFAState *dfa_get_child_state(const DFA *dfa);

bool dfa_check_flush(const DFAState *state);

bool dfa check root(const DFAState *state);

int dfa_check_match(const DFAState *state);

void dfa_eat_character(DFA *dfa);

void dfa_edge_free(GraphEdge* edge);

void dfa_node_free(DFAState *state);

/********** LIBRARY FUNCTIONS **********/

/**
 * dfa_create
 *
 * Initializes a new empty DFA
 *
 * [param] IP, source
 * [param] OR - the newly created DFA
 */
DFA *dfa_create(FILE *source) {
  DFA *dfa = malloc(sizeof(DFA));
  assert(dfa != NULL);

  dfa->graph = graph_create();
  dfa->source = source;
  dfa->line = 1;
  dfa->column = 0;

  dfa_eat_character(dfa);

  return dfa;
```

```
}

/**
 * dfa_get_next_match
 *
 * Returns the next match
 *
 * [param] IOP, dfa
 * [param] OR + IF - the next DFAMatch, or NULL
 */
DFAMatch *dfa_get_next_match(DFA *dfa) {
  DFAState *next_state;

  DFAMatch *match;

  int match type;

  Buffer *buffer = buffer_create();

  while (dfa->character != EOF || !dfa_check_root(dfa->state)) {
    if (dfa_check_flush(dfa->state)) {
      buffer_reset(buffer);
    }

    if ((next_state = dfa_get_child_state(dfa)) != NULL) {
      buffer_append_character(buffer, (char) dfa->character);
      dfa_eat_character(dfa);
      dfa->state = next_state;
    }
    else if ((next_state = dfa_get_child_epsilon(dfa)) != NULL) {
      dfa->state = next_state;
    }
    else {
      printf(
        "[WARNING] No transition found. Skipping character with "
        "ASCII code %d at line %d, column %d.\n",
        dfa->character,
        dfa->line,
        dfa->column
      );

      dfa_eat_character(dfa);
    }
```

```
    if ((match_type = dfa_check_match(dfa->state)) != -1) {
      match = dfa_match_create(
        match_type,
        buffer->value,
        dfa->line,
        dfa->column
      );

      buffer_free(buffer);

      return match;
    }
  }

  buffer_free(buffer);

  return NULL;
}

/**
 * dfa_add_root
 *
 * Adds a root to the DFA.
 *
 * [param] IP, dfa
 * [param] IP, label
 * [param] OR - the newly created DFAState
 */
DFAState *dfa_add_root(const DFA *dfa, const char *label) {
  int *flush_value, *root_value;

  DFAState *state = graph_add_node(dfa->graph, label);

  flush_value = malloc(sizeof(int));
  root_value = malloc(sizeof(int));

  *flush_value = 1;
  *root_value = 1;

  hashmap_set(state->fields, FLUSH, flush_value);
  hashmap_set(state->fields, ROOT, root_value);

  return state;
}
```

```
/**
 * dfa_add_state
 *
 * Adds a DFAState to the DFA.
 *
 * [param] IP, dfa
 * [param] IP, label
 * [param] OR - the newly created DFAState
 */
DFAState *dfa_add_state(const DFA *dfa, const char *label) {
  return graph_add_node(dfa->graph, label);
}

/**
 * dfa add state epsilon
 *
 * Given a DFAState $state, creates a new DFAState and connects it to
 * $state with an epsilon transition.
 *
 * [param] IP, dfa
 * [param] IOP, state
 * [param] IP, prefix
 * [param] OR - the newly created DFAState
 */
DFAState *dfa_add_state_epsilon(
  const DFA *dfa,
  DFAState *state,
  const char *prefix
) {
  DFAState *state_epsilon;

  char *label = malloc(sizeof(char) * (strlen(prefix) + 9));
  assert(label != NULL);

  sprintf(label, "%s_epsilon", prefix);

  state_epsilon = dfa_add_state(dfa, label);

  dfa_epsilon_transition_state_to_state(state, state_epsilon);

  /* Clean up */

  free(label);
```

```
  label = NULL;

  return state_epsilon;
}

/**
 * dfa_add_state_group_epsilon
 *
 * Given a DFAStateGroup $state, creates a new DFAState and connects
 * it to $state with an epsilon transition.
 *
 * [param] IP, dfa
 * [param] IOP, state
 * [param] IP, prefix
 * [param] OR -
 */
DFAState *dfa_add_state_group_epsilon(
  const DFA *dfa,
  DFAStateGroup *state,
  const char *prefix
) {
  DFAState *state_epsilon;

  char *label = malloc(sizeof(char) * (strlen(prefix) + 9));
  assert(label != NULL);

  sprintf(label, "%s_epsilon", prefix);

  state_epsilon = dfa_add_state(dfa, label);

  dfa_epsilon_transition_state_group_to_state(state, state_epsilon);

  /* Clean up */

  free(label);
  label = NULL;

  return state_epsilon;
}

/**
 * dfa_add_state_group
 *
 * Adds a new DFAStateGroup to the DFA.
```

```
 *
 * [param] IP, dfa
 * [param] IP, count
 * [param] IP, prefix
 * [param] OR - the newly created DFAStateGroup
 */
DFAStateGroup *dfa_add_state_group(
  const DFA *dfa,
  unsigned int count,
  const char *prefix
) {
  unsigned int i;

  List *result = list_create_empty();

  char *label = malloc(
    sizeof(char) * (strlen(prefix) + (int) ceil(log10(count)) + 2)
  );

  assert(label != NULL);

  for (i = 0; i < count; i += 1) {
    sprintf(label, "%s_%d", prefix, i);
    list_push_back(result, dfa_add_state(dfa, label));
  }

  /* Clean up */

  free(label);
  label = NULL;

  return result;
}

/**
 * dfa_string_to_transitions
 *
 * Helper function to compute an array of integers containing the
 * ASCII codes of the characters of a string.
 *
 * [param] IP, string
 * [param] OR - an array of integers
 */
int *dfa_string_to_transitions(const char *string) {
```

```
  int i;

  int *result = malloc(sizeof(int) * strlen(string));

  for (i = 0; i < strlen(string); i += 1) {
    result[i] = (int) string[i];
  }

  return result;
}

/**
 * dfa epsilon transition state group to state
 *
 * Creates an epsilon transition between a DFAStateGroup and a
 * DFAState.
 *
 * [param] IOP, state_a
 * [param] IOP, state_b
 */
void dfa_epsilon_transition_state_group_to_state(
  DFAStateGroup *state_a,
  DFAState *state_b
) {
  dfa_transition_state_group_to_state(
    state_a,
    state_b,
    TRANSITION_EPSILON_VALUE
  );
}

/**
 * dfa_epsilon_transition_state_to_state
 *
 * Creates an epsilon transition between two DFAStates.
 *
 * [param] IOP, state_a
 * [param] IOP, state_b
 */
void dfa_epsilon_transition_state_to_state(
  DFAState *state_a,
  DFAState *state_b
) {
  dfa_transition_state_to_state(
```

```
    state_a,
    state_b,
    TRANSITION_EPSILON_VALUE
  );
}

/**
 * dfa_free
 *
 * Frees the memory allocated by a DFA.
 *
 * [param] IOP, dfa
 */
void dfa_free(DFA *dfa) {
  HashMapIterator *iterator;

  for (
    iterator = hashmap_get_iterator(dfa->graph->nodes);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    dfa_node_free(iterator->value);
  }

  graph_free(dfa->graph);

  free(dfa);
  dfa = NULL;
}

/**
 * dfa_match_free
 *
 * Frees the memory allocated by a DFAMatch.
 *
 * [param] IOP, match
 */
void dfa_match_free(DFAMatch *match) {
  free(match->value);
  match->value = NULL;

  free(match);
  match = NULL;
}
```

```
/**
 * dfa_set_leaf
 *
 * Tags a DFAState as leaf.
 *
 * [param] IOP, state
 * [param] IP, match
 */
void dfa_set_leaf(const DFAState *state, int match) {
  int *match_value = malloc(sizeof(int));
  assert(match_value != NULL);

  *match_value = match;

  hashmap set(state->fields, MATCH, match value);
}

/*
 * dfa_set_leaves
 *
 * Tags a DFAStateGroup as leaf.
 *
 * [param] IOP, state
 * [param] IP, match
 */
void dfa_set_leaves(const DFAStateGroup *state, int match) {
  ListNode *iterator;

  int *match_value;

  for (
    iterator = list_get_iterator(state);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    match_value = malloc(sizeof(int));
    assert(match_value != NULL);

    *match_value = match;

    hashmap_set(
      ((DFAState*) iterator->value)->fields,
      MATCH,
```

```
      match_value
    );
  }
}

/**
 * dfa_transition_state_group_to_state
 *
 * Creates an transition between a DFAStateGroup and a DFAState.
 *
 * [param] IOP, state_a
 * [param] IOP, state_b
 */
void dfa_transition_state_group_to_state(
  const DFAStateGroup *state_a,
  DFAState *state b,
  int transition_value
) {
  ListNode *iterator;

  for (
    iterator = list_get_iterator(state_a);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    dfa_transition_state_to_state(
      iterator->value,
      state_b,
      transition_value
    );
  }
}

/**
 * dfa_transition_state_group_to_state_group
 *
 * Creates an transition between a two DFAStateGroups.
 *
 * [param] IOP, state_a
 * [param] IOP, state_b
 * [param] IP, transitions_count
 * [param] IP, transition_value
 * [param] OR -
 */
```

```
void dfa_transition_state_group_to_state_group(
  const DFAStateGroup *state_a,
  DFAStateGroup *state_b,
  int transitions_count,
  int *transition_values
) {
  ListNode *iterator_a;

  for (
    iterator_a = list_get_iterator(state_a);
    iterator_a != NULL;
    iterator_a = list_iterate(iterator_a)
  ) {
    dfa_transition_state_to_state_group(
      iterator_a->value,
      state b,
      transitions_count,
      transition_values
    );
  }
}

/**
 * dfa_transition_state_to_state
 *
 * Creates an transition between a two DFAStates.
 *
 * [param] IOP, state_a
 * [param] IOP, state_b
 * [param] IP, transition_value
 */
void dfa_transition_state_to_state(
  DFAState *state_a,
  DFAState *state_b,
  int transition_value
) {
  int *transition = malloc(sizeof(int));
  *transition = transition_value;

  graph_edge_set_field(
    graph_add_edge(state_a, state_b),
    TRANSITION,
    transition
  );
```

```
}

/**
 * dfa_transition_state_to_state_group
 *
 * Creates an transition between a DFAState and a DFAStateGroup.
 *
 * [param] IOP, state_a
 * [param] IOP, state_b
 * [param] IP, transitions_count
 * [param] IP, transition_values
 */
void dfa transition state to state group(
  DFAState *state_a,
  const DFAStateGroup *state_b,
  int transitions count,
  int *transition_values
) {
  int i = 0;

  ListNode *state_iterator;

  if (state_b->length != transitions_count) {
    exit(-1);
  }

  for (
    state_iterator = list_get_iterator(state_b);
    state_iterator != NULL;
    state_iterator = list_iterate(state_iterator)
  ) {
    dfa_transition_state_to_state(
      state_a,
      state_iterator->value,
      transition_values[i]
    );

    i += 1;
  }
}

/********** RESERVED FUNCTIONS **********/

/**
```

```
 * dfa_match_create
 *
 * Creates a DFAMatch.
 *
 * [param] IP, type
 * [param] IP, value
 * [param] IP, line
 * [param] IP, column
 * [param] OR - the newly created DFAMatch
 */
DFAMatch *dfa_match_create(
  int type,
  const char *value,
  int line,
  int column
) {
  DFAMatch *match = malloc(sizeof(DFAMatch));
  assert(match != NULL);

  match->type = type;
  match->line = line;
  match->column = column - (int) strlen(value);

  /* Setting the value by copy */
  match->value = malloc(sizeof(char) * (strlen(value) + 1));
  assert(match->value != NULL);
  strcpy(match->value, value);

  return match;
}

/**
 * dfa_get_child_epsilon
 *
 * Given the current DFAState of a DFA, looks for an epsilon
 * transition.
 *
 * [param] IP, dfa
 * [param] OR - the next DFAState, or NULL
 */
DFAState *dfa_get_child_epsilon(const DFA *dfa) {
  ListNode *iterator;

  int *transition;
```

```
  for (
    iterator = list_get_iterator(dfa->state->children_edges);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    transition = graph_edge_get_field(iterator->value, TRANSITION);

    if (transition && *transition == TRANSITION_EPSILON_VALUE) {
      return ((GraphEdge*) iterator->value)->child_node;
    }
  }

  return NULL;
}

/**
 * dfa_get_child_state
 *
 * Given the current DFAState of a DFA, looks for a transition.
 *
 * [param] IOP, dfa
 * [param] OR - the next DFAState, or NULL
 */
DFAState *dfa_get_child_state(const DFA *dfa) {
  ListNode *iterator;

  int *transition;

  for (
    iterator = list_get_iterator(dfa->state->children_edges);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    transition = graph_edge_get_field(iterator->value, TRANSITION);

    if (transition && *transition == dfa->character) {
      return ((GraphEdge*) iterator->value)->child_node;
    }
  }

  return NULL;
}
```

```
/**
 * dfa_check_flush
 *
 * Checks whether a DFAState is tagged with FLUSH label.
 *
 * [param] IP, state
 * [param] OR - resulting boolean value
 */
bool dfa_check_flush(const DFAState *state) {
  char *flush_value = hashmap_get(state->fields, FLUSH);

  return flush_value != NULL && *flush_value == 1;
}

/**
 * dfa check root
 *
 * Checks whether a DFAState is tagged with ROOT label.
 *
 * [param] IP, state
 * [param] OR - resulting boolean value
 */
bool dfa_check_root(const DFAState *state) {
  char *root_value = hashmap_get(state->fields, ROOT);

  return root_value != NULL && *root_value == 1;
}

/**
 * dfa_check_match
 *
 * Checks whether a DFAState is tagged with MATCH label.
 *
 * [param] IP, state
 * [param] OR - the value of the match, or NULL
 */
int dfa_check_match(const DFAState *state) {
  char *match_value = hashmap_get(state->fields, MATCH);

  if (match_value == NULL) {
    return -1;
  }

  return *((int*) match_value);
```

```
}

/**
 * dfa_eat_character
 *
 * Consumes a character from the input.
 *
 * [param] IOP, dfa
 */
void dfa_eat_character(DFA *dfa) {
  dfa->character = getc(dfa->source);
  dfa->column += 1;

  while (!validate_character(dfa->character)) {
    printf(
      "[WARNING] Unrecognized character with ASCII code %d found at "
      "line %d, column %d.\n",
      dfa->character,
      dfa->line,
      dfa->column
    );

    dfa->character = getc(dfa->source);
    dfa->column += 1;
  }

  if (dfa->character == 10) {
    dfa->column = 0;
    dfa->line += 1;
  }
}

/**
 * dfa_edge_free
 *
 * Frees the memory allocated by a transition.
 *
 * [param] IP, edge
 */
void dfa_edge_free(GraphEdge* edge) {
  HashMapIterator *iterator;

  for (
    iterator = hashmap_get_iterator(edge->fields);
```

```
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    if (iterator->value != NULL) {
      free(iterator->value);
      iterator->value = NULL;
    }
  }
}

/**
 * dfa_node_free
 *
 * Frees the memory allocated by a DFAState.
 *
 * [param] IP, node
 */
void dfa_node_free(DFAState *state) {
  HashMapIterator *hashmap_iterator;

  ListNode *list_iterator;

  /* Freeing the values of the matches and of the roots */
  for (
    hashmap_iterator = hashmap_get_iterator(state->fields);
    hashmap_iterator != NULL;
    hashmap_iterator = hashmap_iterate(hashmap_iterator)
  ) {
    if (hashmap_iterator->value != NULL) {
      free(hashmap_iterator->value);
      hashmap_iterator->value = NULL;
    }
  }

  /* Freeing the values of the transitions */
  for (
    list_iterator = list_get_iterator(state->children_edges);
    list_iterator != NULL;
    list_iterator = list_iterate(list_iterator)
  ) {
    dfa_edge_free(list_iterator->value);
  }
}
```

8.1.4 `lib_graph`

8.1.4.1 `lib_graph.h`

```
#include "lib_list.h"
#include "lib_hashmap.h"

#ifndef LIB_GRAPH_H
#define LIB_GRAPH_H

typedef struct GraphNode {
  char *identifier;
  HashMap *fields;
  List *children_edges;
  List *parent_edges;
} GraphNode;

typedef struct GraphEdge {
  GraphNode *child_node;
  GraphNode *parent_node;
  HashMap *fields;
} GraphEdge;

typedef struct Graph {
  HashMap *nodes;
} Graph;

#endif

Graph* graph_create(void);

GraphEdge* graph_add_edge(
  GraphNode *parent_node,
  GraphNode *child_node
);

GraphNode* graph_add_node(Graph *graph, const char *label);

void graph_copy_edges(GraphNode *node_a, const GraphNode *node_b);
```

```
void graph_remove_edge(GraphEdge *edge);

void graph_remove_edges(GraphNode *node);

void* graph_edge_get_field(
  const GraphEdge *edge,
  const char *key
);

void* graph_edge_set_field(
  GraphEdge *edge,
  const char *key,
  void *value
);

void* graph node get field(
  const GraphNode *node,
  const char *key
);

void* graph_node_set_field(
  GraphNode *node,
  const char *key,
  void *value
);

GraphNode* graph_dfs_preorder_find_first(
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
);

List* graph_bfs(
  const Graph *graph,
  const List *roots
);

List* graph_dfs_postorder(
  const Graph *graph,
  const List *roots
);

List* graph_dfs_postorder_find_all(
```

```
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
);

List* graph_dfs_preorder_find(
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
);

List* graph_get_scc(const Graph *graph);

List* graph get self loops(const Graph *graph);

void graph_free(Graph *graph);
```

8.1.4.2 `lib_graph.c`

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "lib_common.h"
#include "lib_graph.h"
#include "lib hashmap.h"
#include "lib_list.h"

/********** RESERVED FUNCTION DECLARATIONS **********/

GraphNode *graph_precedence_list_pop(
  List *precedence_list,
  const HashMap *visitation_map
);

GraphNode* graph_visitation_map_get_first(
  const Graph *graph,
  const HashMap *visitation_map
);
```

```
List *graph_precedence_list_create(
  const Graph *graph,
  const List *roots
);

void graph_edge_list_free(List *edge_list);

void graph_node_free(GraphNode* node);

HashMap *graph_visitation_map_create(const Graph *graph);

void graph visitation map free(HashMap *visitation map);

/********** LIBRARY FUNCTIONS **********/

/*
 * graph_create
 *
 * Initializes a new empty Graph.
 *
 * [param] OR - the newly created Graph
 */
Graph *graph_create(void) {
  Graph *graph = (Graph*) malloc(sizeof(Graph));
  assert(graph != NULL);

  graph->nodes = hashmap_create();

  return graph;
}

/**
 * graph_add_edge
 *
 * Given a parent and a child GraphNode, creates a GraphEdge between
 * them.
 *
 * [param] IOP, parent_node
 * [param] IOP, child_node
 * [param] OR - the newly created GraphEdge
 */
GraphEdge *graph_add_edge(
  GraphNode *parent_node,
```

```
  GraphNode *child_node
) {
  GraphEdge *edge = malloc(sizeof(GraphEdge));
  assert(edge != NULL);

  edge->parent_node = parent_node;
  edge->child_node = child_node;
  edge->fields = hashmap_create();

  list_push_back(parent_node->children_edges, edge);
  list_push_back(child_node->parent_edges, edge);

  return edge;
}

/*
 * graph_add_node
 *
 * Creates a GraphNode within a Graph.
 *
 * [param] IOP, graph
 * [param] IP, label
 * [param] OR - the newly created GraphNode
 */
GraphNode *graph_add_node(Graph *graph, const char *label) {
  GraphNode *node = (GraphNode*) malloc(sizeof(GraphNode));
  assert(node != NULL);

  if (hashmap_get(graph->nodes, label) != NULL) {
    printf(
      "[ERROR] Graph node with duplicate identifier %s\n",
      label
    );

    exit(-1);
  }

  node->fields = hashmap_create();
  node->children_edges = list_create_empty();
  node->parent_edges = list_create_empty();

  /* Setting the label by copy */
  node->identifier = malloc(sizeof(char) * (strlen(label) + 1));
  assert(node->identifier != NULL);
```

```
  strcpy(node->identifier, label);

  hashmap_set(graph->nodes, label, node);

  return node;
}

/**
 * graph_copy_edges
 *
 * Given two GraphNodes, copies each GraphEdge of the second one in
 * the first one.
 *
 * [param] IOP, node_a
 * [param] IP, node_b
 */
void graph_copy_edges(GraphNode *node_a, const GraphNode *node_b) {
  ListNode *iterator;
  GraphEdge *edge;

  for (
    iterator = list_get_iterator(node_b->children_edges);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    edge = graph_add_edge(
      node_a,
      (GraphNode*) ((GraphEdge*) iterator->value)->child_node
    );

    hashmap_free(edge->fields);

    edge->fields = hashmap_copy(
      ((GraphEdge*) iterator->value)->fields
    );
  }
}

/**
 * graph_remove_edge
 *
 * Deletes a GraphEdge from the Graph.
 *
 * [param] IOP, edge
```

```
 */
void graph_remove_edge(GraphEdge *edge) {
  ListNode *iterator;

  GraphEdge *current;

  List *node_deletion_list = list_create_empty();
  List *edge_deletion_list = list_create_empty();

  for (
    iterator = list_get_iterator(edge->parent_node->children_edges);
    iterator != NULL;
    iterator = list iterate(iterator)
  ) {
    current = iterator->value;

    if (
      strcmp(
        current->child_node->identifier,
        edge->child_node->identifier
      ) == 0
      && strcmp(
        current->parent_node->identifier,
        edge->parent_node->identifier
      ) == 0
    ) {
      list_push_back(node_deletion_list, iterator);
      list_push_back(edge_deletion_list, iterator->value);
    }
  }

  for (
    iterator = list_get_iterator(edge->child_node->parent_edges);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    current = iterator->value;

    if (
      strcmp(
        current->child_node->identifier,
        edge->child_node->identifier
      ) == 0
      && strcmp(
```

```
        current->parent_node->identifier,
        edge->parent_node->identifier
      ) == 0
    ) {
      list_push_back(node_deletion_list, iterator);
      list_push_back(edge_deletion_list, iterator->value);
    }
  }

  /* Clean-up */

  for (
    iterator = list get iterator(node deletion list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    list_remove(edge->child_node->parent_edges, iterator->value);
  }

  graph_edge_list_free(edge_deletion_list);
  list_free(node_deletion_list);
  list_free(edge_deletion_list);
}

/*
 * graph_remove_edges
 *
 * Deletes every GraphEdge of a GraphNode.
 *
 * [param] IOP, node
 */
void graph_remove_edges(GraphNode *node) {
  ListNode *iterator;

  List *deletion_list = list_create_empty();

  for (
    iterator = list_get_iterator(node->children_edges);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    list_push_back(deletion_list, iterator->value);
  }
```

```
  for (
    iterator = list_get_iterator(deletion_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    graph_remove_edge(iterator->value);
  }

  list_free(deletion_list);
  deletion_list = list_create_empty();

  for (
    iterator = list get iterator(node->parent edges);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    list_push_back(deletion_list, iterator->value);
  }

  for (
    iterator = list_get_iterator(deletion_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    graph_remove_edge(iterator->value);
  }

  /* Clean-up */

  list_free(deletion_list);
}

/**
 * graph_edge_get_field
 *
 * Gets the value of a field of a GraphEdge.
 *
 * [param] IP, edge
 * [param] IP, key
 * [param] OR - the value of the field
 */
void *graph_edge_get_field(const GraphEdge *edge, const char *key) {
  return hashmap_get(edge->fields, key);
}
```

```
/**
 * graph_edge_set_field
 *
 * Sets the value of a field of a GraphEdge.
 *
 * [param] IOP, edge
 * [param] IP, key
 * [param] IP, value
 * [param] OR - the value of the field
 */
void *graph_edge_set_field(
  GraphEdge *edge,
  const char *key,
  void *value
) {
  return hashmap_set(edge->fields, key, value);
}

/**
 * graph_node_get_field
 *
 * Gets the value of a field of a GraphNode.
 *
 * [param] IP, node
 * [param] IP, key
 * [param] OR - the value of the field
 */
void *graph_node_get_field(const GraphNode *node, const char *key) {
  return hashmap_get(node->fields, key);
}

/**
 * graph_node_set_field
 *
 * Sets the value of a field of a GraphNode.
 *
 * [param] IOP, node
 * [param] IP, key
 * [param] IP, value
 * [param] OR - the value of the field
 */
void *graph_node_set_field(
  GraphNode *node,
```

```
  const char *key,
  void *value
) {
  return hashmap_set(node->fields, key, value);
}

/********** EXPLORATION FUNCTIONS **********/

/*
 * graph_dfs_preorder_find_first
 *
 * Exploration algorithm derived from the DFS preorder algorithm.
 * It returns the first encountered GraphNode which matches.
 *
 * [param] IP, graph
 * [param] IP, roots - the list of the GraphNodes to use as roots
 * [param] IP, key - the key of the field to match
 * [param] IP, values - the list of the accepted values
 * [param] OR - the matched GraphNode
 */
GraphNode *graph_dfs_preorder_find_first(
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
) {
  ListNode *iterator;

  GraphNode *current, *child;

  bool *visited;

  char *value;

  /* Initializing the visitation map */

  HashMap *visitation_map = graph_visitation_map_create(graph);

  /* Initializing the precedence list */

  List *precedence_list = list_copy(roots);

  /* Initializing the stack */
```

```
List *stack = list_create_empty();

current = graph_precedence_list_pop(
  precedence_list,
  visitation_map
);

if (current != NULL) {
  list_push_back(stack, current);
}

/* Iterating as long as the stack is non-empty */

while (!list_is_empty(stack)) {
  current = list_pop_back(stack);

  /* Visiting the current GraphNode */

  visited = hashmap_get(visitation_map, current->identifier);

  *visited  = true;

  /* Retrieving the value associated to the given key and */
  /* checking whether the result has been found.          */

  value = graph_node_get_field(current, key);

  if (value != NULL && list_contains_string(values, value)) {
    graph_visitation_map_free(visitation_map);
    list_free(stack);
    list_free(precedence_list);

    return current;
  }

  /* Iterating over the non visited adjacent GraphNodes */

  for (
    iterator = list_get_iterator_reverse(current->children_edges);
    iterator != NULL;
    iterator = list_iterate_reverse(iterator)
  ) {
    child   = ((GraphEdge*) iterator->value)->child_node;
    visited = hashmap_get(visitation_map, child->identifier);
```

```
      if (!*visited) {
        list_push_back(stack, child);
      }
    }

    /* Checks whether the stack is empty, and if so fills it with  */
    /* the next GraphNode from the precedence list.                */

    if (list_is_empty(stack)) {
      current = graph_precedence_list_pop(
        precedence_list,
        visitation map
      );

      if (current != NULL) {
        list_push_back(stack, current);
      }
    }
  }

  /* Clean-up */

  graph_visitation_map_free(visitation_map);
  list_free(stack);
  list_free(precedence_list);

  return NULL;
}

/*
 * graph_bfs
 *
 * BFS exploration algorithm. It returns the list of encountered
 * GraphNodes. It works on the sub-graphs that contain the roots.
 *
 * [param] IP, graph
 * [param] IP, roots - the list of the GraphNodes to use as roots
 * [param] OR - the list of encountered GraphNode
 */
List *graph_bfs(const Graph *graph, const List *roots) {
  bool *visited;

  ListNode *iterator;
```

```
GraphNode *current;

List *result = list_create_empty();

/* Initializing the visitation map */

HashMap *visitation_map = graph_visitation_map_create(graph);

/* Initializing the queue */

List *queue = list_copy(roots);

/* Iterating as long as the queue is non-empty */

while (!list is empty(queue)) {
  current = list_pop_front(queue);
  visited = hashmap_get(visitation_map, current->identifier);

  /* Checks whether the node has already been visited */

  if (!*visited) {
    *visited = true;

    /* Iterates over the adjacent GraphNodes */

    for (
      iterator = list_get_iterator(current->children_edges);
      iterator != NULL;
      iterator = list_iterate(iterator)
    ) {
      list_push_back(
        queue,
        ((GraphEdge*) iterator->value)->child_node
      );
    }

    /* Adds the current GraphNode to the result list */

    list_push_back(result, current);
  }
}

/* Clean up */
```

```
  graph_visitation_map_free(visitation_map);
  list_free(queue);

  return result;
}

/*
 * graph_dfs_postorder
 *
 * DFS postorder exploration algorithm. It returns the list of
 * encountered GraphNodes. It works on the whole graph.
 *
 * [param] IP, graph
 * [param] IP, roots - the list of the GraphNodes to use as roots
 * [param] OR - the list of encountered GraphNode
 */
List *graph_dfs_postorder(const Graph *graph, const List *roots) {
  ListNode *iterator;

  GraphNode *current, *child;

  bool *visited, *found;

  List *result = list_create_empty();

  /* Initialized the visitation map */

  HashMap *visitation_map = graph_visitation_map_create(graph);

  /* Initialized the found map */

  HashMap *found_map = graph_visitation_map_create(graph);

  /* Initializes the precedence list */

  List *precedence_list = graph_precedence_list_create(graph, roots);

  /* Initializes the stack */

  List *stack = list_create_empty();

  current = graph_precedence_list_pop(
    precedence_list,
```

```
    visitation_map
  );

  if (current != NULL) {
    list_push_back(stack, current);
  }

  /* Iterates as long as the stack is non-empty */

  while (!list_is_empty(stack)) {
    current = list_get_last(stack);

    visited = hashmap get(visitation map, current->identifier);
    found   = hashmap_get(found_map, current->identifier);

    /* Checks whether the current GraphNode has already been found */
    /* and if so pops it from the stack and skips the iteration    */

    if (*found) {
      list_pop_back(stack);
      continue;
    }

    *visited  = true;
    *found    = true;

    /* Iterates over the adjacent GraphNodes */

    for (
      iterator = list_get_iterator_reverse(current->children_edges);
      iterator != NULL;
      iterator = list_iterate_reverse(iterator)
    ) {
      child   = ((GraphEdge*) iterator->value)->child_node;
      visited = hashmap_get(visitation_map, child->identifier);

      /* If the adjacent node is not yet visited, marks the    */
      /* parent as not found, and adds the child to the stack. */

      if (!*visited) {
        *found = false;
        list_push_back(stack, child);
      }
    }
```

```
    /* If the current node is found, adds it to the results list */

    if (*found) {
      list_pop_back(stack);
      list_push_back(result, current);
    }

    /* Checks whether the stack is empty, and if so fills it with  */
    /* the next GraphNode from the precedence list.                */

    if (list_is_empty(stack)) {
      current = graph precedence list pop(
        precedence_list,
        visitation_map
      );

      if (current != NULL) {
        list_push_back(stack, current);
      }
    }
  }

  /* Clean-up */

  graph_visitation_map_free(visitation_map);
  graph_visitation_map_free(found_map);
  list_free(stack);
  list_free(precedence_list);

  return result;
}

/*
 * graph_dfs_postorder_find_all
 *
 * Exploration algorithm derived from the DFS postorder algorithm.
 * It returns the list of the matching encountered GraphNodes. It
 * works on the sub-graphs that contain the roots.
 *
 * [param] IP, graph
 * [param] IP, roots - the list of the GraphNodes to use as roots
 * [param] IP, key - the key of the field to match
 * [param] IP, values - the list of the accepted values
```

```
 * [param] OR - the list matched GraphNodes
 */
List *graph_dfs_postorder_find_all(
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
) {
  ListNode *iterator;

  GraphNode *current, *child;

  bool *visited, *found;

  char *value;

  List *result = list_create_empty();

  /* Initialized the visitation map */

  HashMap *visitation_map = graph_visitation_map_create(graph);

  /* Initialized the found map */

  HashMap *found_map = graph_visitation_map_create(graph);

  /* Initialized the precedence list */

  List *precedence_list = list_copy(roots);

  /* Initialized the stack */

  List *stack = list_create_empty();

  /* Initializing the stack */

  current = graph_precedence_list_pop(
    precedence_list,
    visitation_map
  );

  if (current != NULL) {
    list_push_back(stack, current);
  }
```

```
/* Iterating as long as the stack is non-empty */

while (stack->length > 0) {
  current = list_get_last(stack);

  visited = hashmap_get(visitation_map, current->identifier);
  found   = hashmap_get(found_map, current->identifier);

  /* Checks whether the current GraphNode has already been found */
  /* and if so pops it from the stack and skips the iteration    */

  if (*found) {
    list_pop_back(stack);
    continue;
  }

  *visited  = true;
  *found    = true;

  /* Iterates over the adjacent GraphNodes */

  for (
    iterator = list_get_iterator_reverse(current->children_edges);
    iterator != NULL;
    iterator = list_iterate_reverse(iterator)
  ) {
    child   = ((GraphEdge*) iterator->value)->child_node;
    visited = hashmap_get(visitation_map, child->identifier);

    /* If the adjacent node is not yet visited, marks the    */
    /* parent as not found, and adds the child to the stack. */

    if (!*visited) {
      *found = false;
      list_push_back(stack, child);
    }
  }

  /* If the current node is found, checks whether it matches */
  /* the condition and, if so, adds it to the result list.   */

  if (*found) {
    list_pop_back(stack);
```

```
      value = graph_node_get_field(current, key);

      if (value != NULL && list_contains_string(values, value)) {
        list_push_back(result, current);
      }
    }

    /* Checks whether the stack is empty, and if so fills it with  */
    /* the next GraphNode from the precedence list.                */

    if (list_is_empty(stack)) {
      current = graph precedence list pop(
        precedence_list,
        visitation_map
      );

      if (current != NULL) {
        list_push_back(stack, current);
      }
    }
  }

  /* Clean-up */

  graph_visitation_map_free(visitation_map);
  graph_visitation_map_free(found_map);
  list_free(stack);
  list_free(precedence_list);

  return result;
}

/*
 * graph_dfs_preorder_find
 *
 * Exploration algorithm derived from the DFS preorder algorithm.
 * It returns the list of the matching encountered GraphNodes. When a
 * matching GraphNode is found, its children are ignored. It works on
 * the sub-graphs that contain the roots.
 *
 * [param] IP, graph
 * [param] IP, roots - the list of the GraphNodes to use as roots
 * [param] IP, key - the key of the field to match
```

```
 * [param] IP, values - the list of the accepted values
 * [param] OR - the list matched GraphNodes
 */
List *graph_dfs_preorder_find(
  const Graph *graph,
  const List *roots,
  const char *key,
  const List *values
) {
  ListNode *iterator;

  GraphNode *current, *child;

  bool *visited;

  char *value;

  List *result = list_create_empty();

  /* Initialized the visitation map */

  HashMap *visitation_map = graph_visitation_map_create(graph);

  /* Initializes the precedence list */

  List *precedence_list = list_copy(roots);

  /* Initializes the stack */

  List *stack = list_create_empty();

  current = graph_precedence_list_pop(
    precedence_list, visitation_map
  );

  if (current != NULL) {
    list_push_back(stack, current);
  }

  /* Iterates as long as the stack is non-empty */

  while (stack->length > 0) {
    current = list_pop_back(stack);
```

```
    visited = hashmap_get(visitation_map, current->identifier);

    /* Initializing the current node as found */

    *visited  = true;

    /* Checks whether the current node matches and, if so, pushes */
    /* it to the result list and skips the iteration.             */

    value = graph_node_get_field(current, key);

    if (value != NULL && list_contains_string(values, value)) {
      list push back(result, current);
      continue;
    }

    /* Iterates over the adjacent GraphNodes */

    for (
      iterator = list_get_iterator_reverse(current->children_edges);
      iterator != NULL;
      iterator = list_iterate_reverse(iterator)
    ) {
      child   = ((GraphEdge*) iterator->value)->child_node;
      visited = hashmap_get(visitation_map, child->identifier);

      if (!*visited) {
        list_push_back(stack, child);
      }
    }

    /* Checks whether the stack is empty, and if so fills it with  */
    /* the next GraphNode from the precedence list.                */

    if (list_is_empty(stack)) {
      current = graph_precedence_list_pop(
        precedence_list,
        visitation_map
      );

      if (current != NULL) {
        list_push_back(stack, current);
      }
    }
```

```
  }

  /* Clean-up */

  graph_visitation_map_free(visitation_map);
  list_free(stack);
  list_free(precedence_list);

  return result;
}

/*
 * graph get scc
 *
 * Kosaraju's algorithm to get the list of SCCs (strongly connected
 * components) of the Graph.
 *
 * [param] IP, graph
 * [param] OR - the list of SCCs, each SCC is a list of GraphNodes
 */
List *graph_get_scc(const Graph *graph) {
  bool *visited, *found;

  GraphNode *current, *child;

  ListNode *iterator;

  List *precedence_list;

  HashMap *visitation_map = graph_visitation_map_create(graph);
  HashMap *found_map = graph_visitation_map_create(graph);

  List *roots = list_create_empty();

  List *stack = list_create_empty();

  List *result = list_create_empty();

  List *scc = list_create_empty();

  /* Orders the GraphNodes of the Graph with a DFS post-order. The */
  /* value of the starting GraphNode is not important.             */

  List *dfs = graph_dfs_postorder(graph, roots);
```

```
/* Fills the roots list */

for (
  iterator = list_get_iterator(dfs);
  iterator != NULL;
  iterator = list_iterate(iterator)
) {
  list_push_front(roots, iterator->value);
}

/* Initializes the precedence list */

precedence_list = graph_precedence_list_create(graph, roots);

/* Initializes the stack */
current = graph_precedence_list_pop(
  precedence_list,
  visitation_map
);

if (current != NULL) {
  list_push_back(stack, current);
}

/* Doing a DFS exploration on the inverted graph, as indicated */
/* by the Kosaraju's algorithm                                 */

while (!list_is_empty(stack)) {
  current = list_get_last(stack);

  visited = hashmap_get(visitation_map, current->identifier);
  found   = hashmap_get(found_map, current->identifier);

  /* Checks whether the current GraphNode has already been found */
  /* and if so pops it from the stack and skips the iteration    */

  if (*found) {
    list_pop_back(stack);
    continue;
  }

  *visited  = true;
  *found    = true;
```

```
/* Iterates over the adjacent GraphNodes */

for (
  iterator = list_get_iterator_reverse(current->parent_edges);
  iterator != NULL;
  iterator = list_iterate_reverse(iterator)
) {
  child   = ((GraphEdge*) iterator->value)->parent_node;
  visited = hashmap_get(visitation_map, child->identifier);

  /* If the adjacent node is not yet visited, marks the    */
  /* parent as not found, and adds the child to the stack. */

  if (!*visited) {
    *found = false;
    list_push_back(stack, child);
  }
}

/* If the current node is found, adds it to the current SCC */

if (*found) {
  list_push_back(scc, current);
  list_pop_back(stack);
}

/* Checks whether the stack is empty, and if so fills it with  */
/* the next GraphNode from the precedence list.                */

if (list_is_empty(stack)) {
  current = graph_precedence_list_pop(
    precedence_list,
    visitation_map
  );

  /* Pushes the current SCC to the result list */

  list_push_back(result, scc);

  if (current != NULL) {
    /* Empties the current SCC */

    scc = list_create_empty();
```

```
        list_push_back(stack, current);
      }
    }
  }

  /* Clean-up */

  list_free(dfs);
  list_free(roots);
  list_free(stack);
  list_free(precedence_list);
  graph visitation map free(visitation map);
  graph_visitation_map_free(found_map);

  return result;
}

/*
 * graph_get_self_loops
 *
 * Gets the list of GraphNodes with self-loops.
 *
 * [param] IP, graph
 * [param] OR - the list of GraphNodes with self loops
 */
List *graph_get_self_loops(const Graph *graph) {
  HashMapIterator *hashmap_iterator;

  ListNode *list_iterator;

  bool *found;

  GraphNode *current, *child;

  HashMap *visitation_map = graph_visitation_map_create(graph);

  List *result = list_create_empty();

  for (
    hashmap_iterator = hashmap_get_iterator(graph->nodes);
    hashmap_iterator != NULL;
    hashmap_iterator = hashmap_iterate(hashmap_iterator)
  ) {
```

```
    current = hashmap_iterator->value;

    found = hashmap_get(visitation_map, current->identifier);

    for (
      list_iterator = list_get_iterator(current->children_edges);
      list_iterator != NULL;
      list_iterator = list_iterate(list_iterator)
    ) {
      child = ((GraphEdge*) list_iterator->value)->child_node;

      if (
        !*found
        && strcmp(current->identifier, child->identifier) == 0
      ) {
        *found = true;
        list_push_back(result, current);
      }
    }
  }

  /* Clean-up */

  graph_visitation_map_free(visitation_map);

  return result;
}

/********** CLEAN-UP FUNCTIONS **********/

/**
 * graph_free
 *
 * Frees the memory allocated by the Graph.
 *
 * [param] IOP, graph
 */
void graph_free(Graph *graph) {
  HashMapIterator *iterator;

  for (
    iterator = hashmap_get_iterator(graph->nodes);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
```

```
  ) {
    graph_node_free(iterator->value);
  }

  hashmap_free(graph->nodes);

  free(graph);
  graph = NULL;
}

/********** RESERVED FUNCTIONS **********/

/*
 * graph_precedence_list_pop
 *
 * Given a list of GraphNodes, pops every visited GraphNode, and the
 * first not yet visited one, returning it.
 *
 * [param] IOP, precedence_list
 * [param] IP, visitation_map
 * [param] OR - the first popped not yet visited GraphNode
 */
GraphNode *graph_precedence_list_pop(
  List *precedence_list,
  const HashMap *visitation_map
) {
  GraphNode *node;

  bool *visited;

  while (precedence_list->length > 0) {
    node = list_pop_front(precedence_list);

    visited = hashmap_get(visitation_map, node->identifier);

    if (!*visited) {
      return node;
    }
  }

  return NULL;
}

/**
```

```
 * graph_visitation_map_get_first
 *
 * Given a visitation map, returns the first not yet visited
 * GraphNode.
 *
 * [param] IP, graph
 * [param] IP, visitation_map
 * [param] OR - the first not yet visited GraphNode
 */
GraphNode* graph_visitation_map_get_first(
  const Graph *graph,
  const HashMap *visitation_map
) {
  HashMapIterator *iterator;

  GraphNode *result;

  for (
    iterator = hashmap_get_iterator(visitation_map);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    if (!*((bool*) iterator->value)) {
      result = hashmap_get(graph->nodes, iterator->key);

      free(iterator);
      iterator = NULL;

      return result;
    }
  }

  return NULL;
}

/*
 * graph_precedence_list_create
 *
 * Given a list of GraphNodes, creates a precedence spanning over the
 * while Graph.
 *
 * [param] IP, graph
 * [param] IP, roots
 * [param] OR - the newly created precedence list
```

```
 */
List *graph_precedence_list_create(
  const Graph *graph,
  const List *roots
) {
  bool *added;

  ListNode *iterator;

  GraphNode *current;

  HashMap *added_map = graph_visitation_map_create(graph);

  List *precedence_list = list_create_empty();

  for (
    iterator = list_get_iterator(roots);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    current = iterator->value;

    list_push_back(precedence_list, current);

    added = hashmap_get(added_map, current->identifier);
    *added = true;
  }

  while (
    (current = graph_visitation_map_get_first(graph, added_map))
      != NULL
  ) {
    list_push_back(precedence_list, current);

    added = hashmap_get(added_map, current->identifier);
    *added = true;
  }

  /* Clean-up */

  graph_visitation_map_free(added_map);

  return precedence_list;
}
```

```
/*
 * graph_edge_list_free
 *
 * Frees a list of GraphEdges.
 *
 * [param] IOP, edge_list
 */
void graph_edge_list_free(List *edge_list) {
  ListNode *iterator;

  GraphEdge *current;

  char *edge_address;

  HashMap *freed edges = hashmap create();

  for (
    iterator = list_get_iterator(edge_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    current = iterator->value;

    edge_address = get_pointer_address(current);

    /* Checking whether the edge has already been freed */

    if (hashmap_get(freed_edges, edge_address) == NULL) {
      /* Freeing the edge */

      hashmap_free(current->fields);

      free(current);
      current = NULL;

      /* Setting the edge as freed */

      hashmap_set(freed_edges, edge_address, "");
    }

    /* Clean-up */

    free(edge_address);
```

```
  }

  /* Clean-up */

  hashmap_free(freed_edges);
}

/**
 * graph_node_free
 *
 * Frees a GraphNode.
 *
 * [param] IOP, node
 */
void graph_node_free(GraphNode* node) {
  graph edge list free(node->children edges);

  hashmap_free(node->fields);

  list_free(node->children_edges);
  list_free(node->parent_edges);

  free(node->identifier);
  node->identifier = NULL;

  free(node);
  node = NULL;
}

/*
 * graph_visitation_map_create
 *
 * Given a Graph, creates its visitation map. A visitation map is a
 * HashMap that associates the identifier of each GraphNode to a
 * boolean, false by default.
 *
 * [param] IP, graph
 * [param] OR - the newly created visitation HashMap
 */
HashMap *graph_visitation_map_create(const Graph *graph) {
  HashMapIterator *iterator;

  bool *visited;
```

```
  HashMap *result = hashmap_create();

  for (
    iterator = hashmap_get_iterator(graph->nodes);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    visited = malloc(sizeof(bool));
    assert(visited != NULL);
    *visited = false;

    hashmap_set(
      result,
      ((GraphNode*) iterator->value)->identifier,
      visited
    );
  }

  return result;
}

/*
 * graph_visitation_map_free
 *
 * Frees a visitation HashMap.
 *
 * [param] IOP, visitation_map
 */
void graph_visitation_map_free(HashMap *visitation_map) {
  HashMapIterator *iterator;

  for (
    iterator = hashmap_get_iterator(visitation_map);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    free(iterator->value);
    iterator->value = NULL;
  }

  hashmap_free(visitation_map);
}
```

8.1.5 `lib_hashmap`

8.1.5.1 `lib_hashmap.h`

```
#include "lib_list.h"

#ifndef LIB_HASHMAP_H
#define LIB_HASHMAP_H

typedef struct HashMap {
  List **lists;
  int size;
  int length;
} HashMap;

typedef struct HashMapNode {
  char *key;
  void *value;
} HashMapNode;

typedef struct HashMapIterator {
  char *key;
  void *value;
  const HashMap *hashmap;
  ListNode *list_node;
  int list_index;
} HashMapIterator;

#endif

HashMap*          hashmap_copy(const HashMap *hashmap);
HashMap*          hashmap_create(void);
HashMapIterator*  hashmap_get_iterator(const HashMap *hashmap);
HashMapIterator*  hashmap_iterate(HashMapIterator *iterator);
void  hashmap_free(HashMap *hashmap);
void* hashmap_get(const HashMap *hashmap, const char *key);
void* hashmap_set(HashMap *hashmap, const char *key, void *value);
```

8.1.5.2 `lib_hashmap.c`

```
#include <assert.h>
#include <stdlib.h>
#include <string.h>

#include "lib_hashmap.h"
#include "lib_list.h"

#define HASHMAP_DEFAULT_SIZE 16

/********** RESERVED FUNCTION DECLARATIONS **********/

int hashmap_compute_key(
  const HashMap* hashmap,
  const char* key
);

int hashmap_get_next_list_index(
  const HashMap* hashmap,
  int starting_index
);

int hashmap_get_previous_list_index(
  const HashMap* hashmap,
  int starting_index
);

/********** LIBRARY FUNCTIONS **********/

/**
 * hashmap_copy
 *
 * This function creates a new HashMap copying the content of an
 * existing one.
 *
 * [param] IP, hashmap
 * [param] OR - the newly copied hashmap
 */
HashMap* hashmap_copy(const HashMap *hashmap) {
  char *key;

  HashMapIterator *iterator;
```

```
  HashMap *result = hashmap_create();

  for (
    iterator = hashmap_get_iterator(hashmap);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    key = malloc(sizeof(char) * (strlen(iterator->key) + 1));
    assert(key != NULL);
    strcpy(key, iterator->key);

    hashmap set(result, key, iterator->value);
  }

  return result;
}

/**
 * hashmap_create
 *
 * This function creates a new HashMap.
 *
 * [param] OR - the newly created hashmap
 */
HashMap* hashmap_create(void) {
  int i;

  HashMap* hashmap = malloc(sizeof(HashMap));
  assert(hashmap != NULL);

  hashmap->size = HASHMAP_DEFAULT_SIZE;
  hashmap->length = 0;

  hashmap->lists = malloc(HASHMAP_DEFAULT_SIZE * sizeof(List*));
  assert(hashmap->lists != NULL);

  for (i = 0; i < HASHMAP_DEFAULT_SIZE; i++) {
    hashmap->lists[i] = list_create_empty();
  }

  return hashmap;
}
```

```
/**
 * hashmap_get_iterator
 *
 * This function creates a new HashMapIterator, to iterate
 * forward over the HashMap.
 *
 * [param] IP, hashmap
 * [param] OR - the value of the HashMapIterator
 */
HashMapIterator* hashmap_get_iterator(const HashMap *hashmap) {
  HashMapIterator *iterator;

  int list index = hashmap get next list index(hashmap, -1);

  if (list_index == -1) {
    return NULL;
  }

  iterator = malloc(sizeof(HashMapIterator));
  assert(iterator != NULL);

  iterator->hashmap = hashmap;
  iterator->list_index = list_index;
  iterator->list_node = list_get_iterator(
    hashmap->lists[iterator->list_index]
  );

  iterator->value =
    ((HashMapNode*) iterator->list_node->value)->value;

  iterator->key =
    ((HashMapNode*) iterator->list_node->value)->key;

  return iterator;
}

/**
 * hashmap_iterate
 *
 * This function updates the HashMapIterator, and destroys it if
 * no longer needed.
 *
 * [param] IOP, iterator
 * [param] OR - the value of the next HashMapIterator
```

```
 */
HashMapIterator* hashmap_iterate(HashMapIterator *iterator) {
  iterator->list_node = list_iterate(iterator->list_node);

  if (iterator->list_node == NULL) {
    iterator->list_index = hashmap_get_next_list_index(
      iterator->hashmap,
      iterator->list_index
    );

    if (iterator->list_index == -1) {
      free(iterator);
      iterator = NULL;
      return NULL;
    }

    iterator->list_node = list_get_iterator(
      iterator->hashmap->lists[iterator->list_index]
    );
  }

  iterator->value =
    ((HashMapNode*) iterator->list_node->value)->value;

  iterator->key =
    ((HashMapNode*) iterator->list_node->value)->key;

  return iterator;
}

/**
 * hashmap_get
 *
 * This function retrieves the value associated to a certain key.
 *
 * [param] IP, hashmap
 * [param] IP, key
 * [param] OR - the value of the requested HashMapNode, or NULL
 */
void* hashmap_get(const HashMap *hashmap, const char *key) {
  ListNode* iterator;

  for (
    iterator = list_get_iterator(
```

```
      hashmap->lists[hashmap_compute_key(hashmap, key)]
    );
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    if (
      strcmp(((HashMapNode*) iterator->value)->key, key) == 0
    ) {
      return ((HashMapNode*) iterator->value)->value;
    }
  }

  return NULL;
}

/**
 * hashmap_set
 *
 * This function sets a new key-value couple.
 *
 * [param] IOP, hashmap
 * [param] IP, key
 * [param] IP, value
 * [param] OR - the value of the just created HashMapNode
 */
void* hashmap_set(
  HashMap* hashmap,
  const char *key,
  void *value
) {
  HashMapNode* node;
  ListNode* iterator;

  for (
    iterator = list_get_iterator(
      hashmap->lists[hashmap_compute_key(hashmap, key)]
    );
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    if (
      strcmp(((HashMapNode*) iterator->value)->key, key) == 0
    ) {
      ((HashMapNode*) iterator->value)->value = value;
```

```
      return value;
    }
  }

  node = malloc(sizeof(HashMapNode));
  assert(node != NULL);

  node->value = value;

  /* Setting the key by copy */
  node->key = malloc(sizeof(char) * (strlen(key) + 1));
  assert(node->key != NULL);
  strcpy(node->key, key);

  list_push_back(
    hashmap->lists[hashmap compute key(hashmap, key)],
    node
  );

  hashmap->length += 1;

  return value;
}

/********** CLEAN-UP FUNCTIONS **********/

/**
 * hashmap_free
 *
 * This method does not free the memory allocated either for the
 * value or the key of the HashMapNode. The task of freeing those
 * pointers is left to the user of the library.
 *
 * [param] IOP, hashmap
 */
void hashmap_free(HashMap* hashmap) {
  int i;

  ListNode *iterator;

  HashMapNode *hashmap_node;

  for (i = 0; i < hashmap->size; i++) {
    for (
```

```
      iterator = list_get_iterator(hashmap->lists[i]);
      iterator != NULL;
      iterator = list_iterate(iterator)
    ) {
      hashmap_node = (HashMapNode*) iterator->value;

      free(hashmap_node->key);
      hashmap_node->key = NULL;

      free(hashmap_node);
      hashmap_node = NULL;
    }

    list_free(hashmap->lists[i]);
  }

  free(hashmap->lists);
  hashmap->lists = NULL;

  free(hashmap);
  hashmap = NULL;
}

/********** RESERVED FUNCTIONS **********/

/**
 * hashmap_compute_key
 *
 * This function associates a key to one of the lists of the
 * hashmap.
 *
 * [param] IP, hashmap
 * [param] IP, key
 * [param] OR - the index of the list which the key is associated
 *              to
 */
int hashmap_compute_key(
  const HashMap *hashmap,
  const char *key
) {
  int hash_code = 0;

  while(*key != '\0') {
    hash_code += *key + 31;
```

```
    key += 1;
  }

  return hash_code % hashmap->size;
}

/**
 * hashmap_get_next_list_index
 *
 * This function computes the index of the list containing the
 * following item to the one with a certain index.
 *
 * [param] IP, hashmap
 * [param] IP, starting_index
 * [param] OR - the index of the list
 */
int hashmap_get_next_list_index(
  const HashMap *hashmap,
  int starting_index
) {
  if (starting_index >= hashmap->size - 1) return -1;

  if (
    hashmap->lists[starting_index + 1] != NULL
    && hashmap->lists[starting_index + 1]->length > 0
  ) {
    return starting_index + 1;
  }

  return hashmap_get_next_list_index(
    hashmap,
    starting_index + 1
  );
}

/**
 * hashmap_get_previous_list_index
 *
 * This function computes the index of the list containing the
 * previous item to the one with a certain index.
 *
 * [param] IP, hashmap
 * [param] IP, starting_index
 * [param] OR - the index of the list
```

```
 */
int hashmap_get_previous_list_index(
  const HashMap *hashmap,
  int starting_index
) {
  if (starting_index <= 0) return -1;

  if (
    hashmap->lists[starting_index - 1] != NULL
    && hashmap->lists[starting_index - 1]->length > 0
  ) {
    return starting_index - 1;
  }

  return hashmap_get_previous_list_index(
    hashmap,
    starting_index - 1
  );
}
```

8.1.6 `lib_list`

8.1.6.1 `lib_list.h`

```
#include <stdbool.h>

#ifndef LIB_LIST_H
#define LIB_LIST_H

typedef struct ListNode {
  void* value;
  struct ListNode* next;
  struct ListNode* prev;
} ListNode;

typedef struct List {
  struct ListNode* first;
  struct ListNode* last;
  unsigned int length;
} List;

#endif

List*     list_copy(const List *list);
List*     list_create(int length, ...);
List*     list_create_empty(void);
ListNode* list_get(const List *list, unsigned int index);
ListNode* list_get_iterator(const List *list);
ListNode* list_get_iterator_reverse(const List *list);
ListNode* list_iterate(const ListNode *iterator);
ListNode* list_iterate_reverse(const ListNode *iterator);
ListNode* list_push_back(List *list, void* value);
ListNode* list_push_front(List *list, void* value);
bool      list_is_empty(const List *list);
void*     list_get_last(const List *list);
void*     list_pop_back(List *list);
void*     list_pop_front(List *list);
void      list_free(List *list);
void      list_remove(List *list, ListNode *node);
```

8.1.6.2 `lib_list.c`

```
#include <assert.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "lib_list.h"

/********** LIBRARY FUNCTIONS **********/

/**
 * list_copy
 *
 * [param] IP, list
 * [param] OR - copy of the list
 */
List* list_copy(const List *list) {
  ListNode *iterator;

  List *result = list_create_empty();

  for (
    iterator = list_get_iterator(list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    list_push_back(result, iterator->value);
  }

  return result;
}

/**
 * list_create
 *
 * Creates a list of a certain length, and fills it with the
 * items passed as arguments.
 *
 * [param] IP, length
 * [param] OR - the list that has been created, filled with the
```

```
 *              items passed as arguments.
 */
List* list_create(int length, ...) {
  int i;

  va_list vl;

  List *list;
  ListNode *first_node, *last_node;

  list = (List*) malloc(sizeof(List));
  assert(list != NULL);

  first_node = (ListNode*) malloc(sizeof(ListNode));
  assert(first_node != NULL);

  last_node = (ListNode*) malloc(sizeof(ListNode));
  assert(last_node != NULL);

  first_node->value = NULL;
  first_node->next = last_node;
  first_node->prev = NULL;

  last_node->value = NULL;
  last_node->next = NULL;
  last_node->prev = first_node;

  list->first = first_node;
  list->last = last_node;

  list->length = 0;

  va_start(vl, length);

  for (i = 0; i < length; i++) {
    list_push_back(list, va_arg(vl, void*));
  }

  va_end(vl);

  return list;
}

/**
```

```
 * list_create_empty
 *
 * Creates an empty list.
 *
 * [param] OR - the list that has been created.
 */
List* list_create_empty(void) {
  List *list;

  ListNode *first_node, *last_node;

  list = (List*) malloc(sizeof(List));
  assert(list != NULL);

  first_node = (ListNode*) malloc(sizeof(ListNode));
  assert(first node != NULL);

  last_node = (ListNode*) malloc(sizeof(ListNode));
  assert(last_node != NULL);

  first_node->value = NULL;
  first_node->next = last_node;
  first_node->prev = NULL;

  last_node->value = NULL;
  last_node->next = NULL;
  last_node->prev = first_node;

  list->first = first_node;
  list->last = last_node;

  list->length = 0;

  return list;
}

/**
 * list_get
 *
 * Retrieves the item within a list with a certain index.
 *
 * [param] IP, list
 * [param] IP, index
 * [param] OR - the retrieved item, or NULL
```

```
 */
ListNode* list_get(const List *list, unsigned int index) {
  unsigned int current_index;
  ListNode *iterator;

  if (index > list->length / 2) {
    current_index = list->length - 1;
    iterator = list_get_iterator_reverse(list);

    while (current_index > index && iterator != NULL) {
      current_index -= 1;
      iterator = list_iterate_reverse(iterator);
    }
  } else {
    current_index = 0;
    iterator = list get iterator(list);

    while (current_index < index && iterator != NULL) {
      current_index += 1;
      iterator = list_iterate(iterator);
    }
  }

  return iterator;
}

/**
 * list_get_iterator
 *
 * Retrieves the front iterator of a list.
 *
 * [param] IP, list
 * [param] OR - The front iterator of a list, or NULL
 */
ListNode* list_get_iterator(const List *list) {
  if (list->first->next->next == NULL) {
    return NULL;
  }

  return list->first->next;
}

/**
 * list_get_iterator_reverse
```

```
 *
 * Retrieves the back iterator of a list.
 *
 * [param] IP, list
 * [param] OR - The back iterator of a list, or NULL
 */
ListNode* list_get_iterator_reverse(const List *list) {
  if (list->last->prev->prev == NULL) {
    return NULL;
  }

  return list->last->prev;
}

/**
 * list iterate
 *
 * Given an iterator, retrieves the next one going forward.
 *
 * [param] IP, iterator
 * [param] OR - the next iterator, or NULL
 */
ListNode* list_iterate(const ListNode *iterator) {
  if (iterator->next->value == NULL) {
    return NULL;
  }

  return iterator->next;
}

/**
 * list_iterate_reverse
 *
 * Given an iterator, retrieves the next one going backward.
 *
 * [param] IP, iterator
 * [param] OR - the next iterator, or NULL
 */
ListNode* list_iterate_reverse(const ListNode *iterator) {
  if (iterator->prev->value == NULL) {
    return NULL;
  }

  return iterator->prev;
```

```
}

/**
 * list_push_back
 *
 * Pushes an item in the back of the list.
 *
 * [param] IOP, list
 * [param] IP, value
 * [param] OR - the freshly created ListNode
 */
ListNode* list_push_back(List *list, void* value) {
  list->last->value = value;

  list->last->next = (ListNode*) malloc(sizeof(ListNode));
  assert(list->last->next != NULL);

  list->last->next->value = NULL;
  list->last->next->next = NULL;
  list->last->next->prev = list->last;

  list->last = list->last->next;

  list->length += 1;

  return list->last->prev;
}

/**
 * list_push_front
 *
 * Pushes an item in the front of the list.
 *
 * [param] IOP, list
 * [param] IP, value
 * [param] OR - the freshly created ListNode
 */
ListNode *list_push_front(List *list, void *value) {
  list->first->value = value;

  list->first->prev = (ListNode*) malloc(sizeof(ListNode));
  assert(list->first->prev != NULL);

  list->first->prev->value = NULL;
```

```
  list->first->prev->next = list->first;
  list->first->prev->prev = NULL;

  list->first = list->first->prev;

  list->length += 1;

  return list->first->next;
}

/**
 * list_is_empty
 *
 * Checks whether the list is empty.
 *
 * [param] IP, list
 * [param] OR - true when the list is empty
 */
bool list_is_empty(const List *list) {
  return list->length == 0;
}

/**
 * list_get_last
 *
 * Retrieves the last item of the list.
 *
 * [param] IP, list
 * [param] OR - the last item of the list
 */
void *list_get_last(const List *list) {
  return list->last->prev->value;
}

/**
 * list_pop_back
 *
 * Pops an item from the back of a list.
 *
 * [param] IOP, list
 * [param] OR - value of the popped item
 */
void *list_pop_back(List *list) {
  void *result;
```

```
  ListNode* back_node = list->last->prev;

  if (list_is_empty(list)) {
    return NULL;
  }

  list->last->prev = list->last->prev->prev;
  list->last->prev->next = list->last;

  list->length -= 1;

  result = back node->value;

  /* Clean-up */

  free(back_node);
  back_node = NULL;

  return result;
}

/**
 * list_pop_front
 *
 * Pops an item from the front of a list.
 *
 * [param] IOP, list
 * [param] OR - value of the popped item
 */
void *list_pop_front(List *list) {
  void *result;

  ListNode* front_node = list->first->next;

  if (list_is_empty(list)) {
    return NULL;
  }

  list->first->next = list->first->next->next;
  list->first->next->prev = list->first;

  list->length -= 1;
```

```
  result = front_node->value;

  /* Clean-up */

  free(front_node);
  front_node = NULL;

  return result;
}

/**
 * list_remove
 *
 * Removes an item from a list.
 *
 * [param] IOP, list
 * [param] IOP, node
 */
void list_remove(List *list, ListNode *node) {
  node->prev->next = node->next;
  node->next->prev = node->prev;

  list->length -= 1;

  free(node);
  node = NULL;
}

/********** CLEAN-UP FUNCTIONS **********/

/**
 * list_free
 *
 * This method does not free the memory allocated for the value
 * of the ListNode. The task of freeing those pointers is left
 * to the user of the library.
 *
 * [param] IOP, list
 */
void list_free(List *list) {
  ListNode *iterator;

  for (
    iterator = list_get_iterator(list);
```

```
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    free(iterator->prev);
    iterator->prev = NULL;
  }

  free(list->last->prev);
  list->last->prev = NULL;

  free(list->last);
  list->last = NULL;

  free(list);
  list = NULL;
}
```

8.2 Lexer

8.2.1 `mod_lexer`

8.2.1.1 `mod_lexer.h`

```
#include <stdio.h>

#include "lib_dfa.h"

#ifndef MOD_LEXER_H
#define MOD_LEXER_H

typedef struct Lexer {
  DFA *dfa;
} Lexer;

#endif

DFAMatch  *lexer_get_next_token(Lexer *lexer);
Lexer     *lexer_create(FILE *file);
void      lexer_free(Lexer *lexer);
```

8.2.1.2 `mod_lexer.c`

```
#include <stdlib.h>
#include <assert.h>

#include "lib_dfa.h"
#include "mod_lexer.h"
#include "mod_lexer_dfa_c90_comment.h"
#include "mod_lexer_dfa_c90_identifier.h"
#include "mod_lexer_dfa_c90_keyword.h"
#include "mod_lexer_dfa_c90_literal.h"
```

```
#include "mod_lexer_dfa_c90_numeric.h"
#include "mod_lexer_dfa_c90_symbol.h"
#include "mod_lexer_dfa_c90_skip.h"
#include "mod_lexer_dfa_c90_type_identifier.h"
#include "mod_lexer_dfa_v_doc.h"
#include "mod_lexer_dfa_v_doc_rule.h"

/**
 * lexer_create
 *
 * Creates a new Lexer.
 *
 * [param] IOP, source
 * [param] OR - The newly created Lexer
 */
Lexer *lexer create(FILE *source) {
  DFAState *state_dot,
    *state_type,
    *root_source_scope,
    *root_comment_scope,
    *root_doc_scope,
    *root_doc_rule_scope;

  Lexer *lexer;

  DFAStateGroup *state_lowercase = list_create_empty();
  DFAStateGroup *state_uppercase = list_create_empty();

  lexer = malloc(sizeof(Lexer));
  assert(lexer != NULL);

  lexer->dfa = dfa_create(source);

  /* Adding the roots to the lexer */

  root_source_scope = dfa_add_root(
    lexer->dfa, "root_source_scope"
  );

  root_comment_scope = dfa_add_root(
    lexer->dfa, "root_comment_scope"
  );

  root_doc_scope = dfa_add_root(
```

```
    lexer->dfa, "root_doc_scope"
  );

  root_doc_rule_scope = dfa_add_root(
    lexer->dfa, "root_doc_rule_scope"
  );

  /* Setting the source scope root as initial state */

  lexer->dfa->state = root_source_scope;

  lexer_dfa_c90_literal_create(
    lexer->dfa,
    root_source_scope,
    state_uppercase
  );

  lexer_dfa_c90_keyword_create(
    lexer->dfa,
    root_source_scope,
    state_lowercase,
    &state_type
  );

  lexer_dfa_c90_identifier_create(
    lexer->dfa,
    root_source_scope,
    state_lowercase,
    state_uppercase
  );

  lexer_dfa_c90_symbol_create(
    lexer->dfa,
    root_source_scope,
    root_comment_scope,
    &state_dot
  );

  lexer_dfa_c90_numeric_create(
    lexer->dfa,
    root_source_scope,
    state_dot
  );
```

```
  lexer_dfa_c90_skip_create(
    lexer->dfa,
    root_source_scope
  );

  lexer_dfa_c90_type_identifier_create(
    lexer->dfa,
    root_source_scope,
    state_type
  );

  lexer_dfa_c90_comment_create(
    lexer->dfa,
    root_source_scope,
    root_comment_scope,
    root doc scope
  );

  lexer_dfa_v_doc_create(
    lexer->dfa,
    root_comment_scope,
    root_doc_scope,
    root_doc_rule_scope
  );

  lexer_dfa_v_doc_rule_create(
    lexer->dfa,
    root_source_scope,
    root_comment_scope,
    root_doc_rule_scope
  );

  /* Clean-up */

  list_free(state_lowercase);
  list_free(state_uppercase);

  return lexer;
}

/**
 * lexer_get_next_token
 */
DFAMatch *lexer_get_next_token(Lexer *lexer) {
```

```
  return dfa_get_next_match(lexer->dfa);
}

/********** CLEAN-UP FUNCTIONS **********/

/**
 * lexer_free
 */
void lexer_free(Lexer *lexer) {
  dfa_free(lexer->dfa);

  free(lexer);
  lexer = NULL;
}
```

8.2.2 mod_lexer_dfa_c90_comment

8.2.2.1 mod_lexer_dfa_c90_comment.h

```
#include <stdlib.h>
#include <assert.h>

#include "lib_dfa.h"
#include "mod_lexer.h"
#include "mod_lexer_dfa_c90_comment.h"
#include "mod_lexer_dfa_c90_identifier.h"
#include "mod_lexer_dfa_c90_keyword.h"
#include "mod_lexer_dfa_c90_literal.h"
#include "mod_lexer_dfa_c90_numeric.h"
#include "mod_lexer_dfa_c90_symbol.h"
#include "mod_lexer_dfa_c90_skip.h"
#include "mod_lexer_dfa_c90_type_identifier.h"
#include "mod_lexer_dfa_v_doc.h"
#include "mod_lexer_dfa_v_doc_rule.h"

/**
 * lexer_create
 *
 * Creates a new Lexer.
 *
 * [param] IOP, source
 * [param] OR - The newly created Lexer
 */
Lexer *lexer_create(FILE *source) {
  DFAState *state_dot,
    *state_type,
    *root_source_scope,
    *root_comment_scope,
    *root_doc_scope,
    *root_doc_rule_scope;

  Lexer *lexer;

  DFAStateGroup *state_lowercase = list_create_empty();
  DFAStateGroup *state_uppercase = list_create_empty();
```

```
lexer = malloc(sizeof(Lexer));
assert(lexer != NULL);

lexer->dfa = dfa_create(source);

/* Adding the roots to the lexer */

root_source_scope = dfa_add_root(
  lexer->dfa, "root_source_scope"
);

root comment scope = dfa add root(
  lexer->dfa, "root_comment_scope"
);

root_doc_scope = dfa_add_root(
  lexer->dfa, "root_doc_scope"
);

root_doc_rule_scope = dfa_add_root(
  lexer->dfa, "root_doc_rule_scope"
);

/* Setting the source scope root as initial state */

lexer->dfa->state = root_source_scope;

lexer_dfa_c90_literal_create(
  lexer->dfa,
  root_source_scope,
  state_uppercase
);

lexer_dfa_c90_keyword_create(
  lexer->dfa,
  root_source_scope,
  state_lowercase,
  &state_type
);

lexer_dfa_c90_identifier_create(
  lexer->dfa,
  root_source_scope,
```

```
    state_lowercase,
    state_uppercase
  );

  lexer_dfa_c90_symbol_create(
    lexer->dfa,
    root_source_scope,
    root_comment_scope,
    &state_dot
  );

  lexer_dfa_c90_numeric_create(
    lexer->dfa,
    root_source_scope,
    state_dot
  );

  lexer_dfa_c90_skip_create(
    lexer->dfa,
    root_source_scope
  );

  lexer_dfa_c90_type_identifier_create(
    lexer->dfa,
    root_source_scope,
    state_type
  );

  lexer_dfa_c90_comment_create(
    lexer->dfa,
    root_source_scope,
    root_comment_scope,
    root_doc_scope
  );

  lexer_dfa_v_doc_create(
    lexer->dfa,
    root_comment_scope,
    root_doc_scope,
    root_doc_rule_scope
  );

  lexer_dfa_v_doc_rule_create(
    lexer->dfa,
```

```
    root_source_scope,
    root_comment_scope,
    root_doc_rule_scope
  );

  /* Clean-up */

  list_free(state_lowercase);
  list_free(state_uppercase);

  return lexer;
}

/**
 * lexer_get_next_token
 */
DFAMatch *lexer_get_next_token(Lexer *lexer) {
  return dfa_get_next_match(lexer->dfa);
}

/********** CLEAN-UP FUNCTIONS **********/

/**
 * lexer_free
 */
void lexer_free(Lexer *lexer) {
  dfa_free(lexer->dfa);

  free(lexer);
  lexer = NULL;
}
```

8.2.2.2 `mod_lexer_dfa_c90_comment.c`

```
#include <stdlib.h>
#include <string.h>

#include "lib_common.h"
#include "mod_lexer_dfa_c90_comment.h"
#include "mod_parser.tab.h"
```

```
#define PREFIX "comment_"
#define STATE_2_TRANSITIONS "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijk" \
  "lmnopqrstuvwxyz0123456789!\"#%&'()+,-./:;<=>?\\]^_{|}~ \t\v\f\n"

/*
 * lexer_dfa_c90_comment_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, source_scope_root
 * [param] IOP, comment_scope_root
 * [param] IOP, docs scope root
 */
void lexer_dfa_c90_comment_create(
  const DFA *dfa,
  DFAState *source_scope_root,
  DFAState *comment_scope_root,
  DFAState *docs_scope_root
) {
  int i;

  ListNode *state_iterator;

  int *state_2_transitions = dfa_string_to_transitions(
    STATE_2_TRANSITIONS
  );

  DFAStateGroup *state_2 = dfa_add_state_group(
    dfa, strlen(STATE_2_TRANSITIONS), PREFIX "state_2"
  );

  DFAState *state_3 = dfa_add_state(dfa, PREFIX "state_3");

  DFAState *state_4 = dfa_add_state(dfa, PREFIX "state_4");

  /*** TRANSITION FROM COMMENT SCOPE ROOT ***/

  dfa_transition_state_to_state_group(
    comment_scope_root,
    state_2,
    strlen(STATE_2_TRANSITIONS),
    state_2_transitions
  );
```

```
dfa_transition_state_to_state(
  comment_scope_root, state_3, '*'
);

dfa_epsilon_transition_state_to_state(
  comment_scope_root, docs_scope_root
);

/*** TRANSITION FROM STATE 2 ***/

dfa_transition_state_group_to_state_group(
  state 2,
  state_2,
  strlen(STATE_2_TRANSITIONS),
  state 2 transitions
);

dfa_transition_state_group_to_state(
  state_2, state_3, '*'
);

dfa_epsilon_transition_state_group_to_state(
  state_2, docs_scope_root
);

/*** TRANSITION FROM STATE 3 ***/

dfa_transition_state_to_state(state_3, state_3, '*');
dfa_transition_state_to_state(state_3, state_4, '/');
dfa_epsilon_transition_state_to_state(state_3, docs_scope_root);

/*** TRANSITION FROM STATE 3 TO STATE 2 ***/

i = 0;

for (
  state_iterator = list_get_iterator(state_2);
  state_iterator != NULL;
  state_iterator = list_iterate(state_iterator)
) {
  if (
    STATE_2_TRANSITIONS[i] != '['
    && STATE_2_TRANSITIONS[i] != '*'
```

```
      && STATE_2_TRANSITIONS[i] != '/'
    ) {
      dfa_transition_state_to_state(
        state_3,
        state_iterator->value,
        STATE_2_TRANSITIONS[i]
      );
    }

    i += 1;
  }

  /*** TRANSITION FROM STATE 4 ***/

  dfa_set_leaf(state_4, CLOSE_COMMENT);
  dfa epsilon transition state to state(state 4, source scope root);

  /* Clean-up */

  free(state_2_transitions);
  state_2_transitions = NULL;

  list_free(state_2);
}
```

8.2.3 mod_lexer_dfa_c90_identifier

8.2.3.1 mod_lexer_dfa_c90_identifier.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_c90_identifier_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  List *nodes_lowercase,
  List *nodes_uppercase
);
```

8.2.3.2 mod_lexer_dfa_c90_identifier.c

```
#include <string.h>
#include <stdlib.h>

#include "lib_common.h"
#include "mod_parser.tab.h"
#include "mod_lexer_dfa_c90_identifier.h"

#define PREFIX "identifier_"

#define STATE_2_TRANSITIONS "_ABCDEFGHIJKMNOPQRSTUVWXYZhjkmnopqxyz"

#define STATE_3_TRANSITIONS "_ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij" \
  "klmnopqrstuvwxyz"

#define STATE_4_TRANSITIONS "0123456789"

/*
 * lexer_dfa_c90_identifier_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
```

```
 * [param] IOP, source_scope_root
 * [param] IOP, nodes_lowercase
 * [param] IOP, nodes_uppercase
 */
void lexer_dfa_c90_identifier_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  List *nodes_lowercase,
  List *nodes_uppercase
) {
  DFAState *epsilon;

  int *state 2 transitions = dfa string to transitions(
    STATE_2_TRANSITIONS
  );

  int *state_3_transitions = dfa_string_to_transitions(
    STATE_3_TRANSITIONS
  );

  int *state_4_transitions = dfa_string_to_transitions(
    STATE_4_TRANSITIONS
  );

  DFAStateGroup *state_2 = dfa_add_state_group(
    dfa, strlen(STATE_2_TRANSITIONS), PREFIX "state_2"
  );

  DFAStateGroup *state_3 = dfa_add_state_group(
    dfa, strlen(STATE_3_TRANSITIONS), PREFIX "state_3"
  );

  DFAStateGroup *state_4 = dfa_add_state_group(
    dfa, strlen(STATE_4_TRANSITIONS), PREFIX "state_4"
  );

  /*** TRANSITION FROM SOURCE SCOPE ROOT ***/

  dfa_transition_state_to_state_group(
    source_scope_root,
    state_2,
    strlen(STATE_2_TRANSITIONS),
    state_2_transitions
  );
```

```
/*** TRANSITION FROM STATE 2 ***/

dfa_transition_state_group_to_state_group(
  state_2,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

dfa_transition_state_group_to_state_group(
  state_2,
  state 4,
  strlen(STATE_4_TRANSITIONS),
  state_4_transitions
);

epsilon = dfa_add_state_group_epsilon(
  dfa, state_2, PREFIX "state_2"
);

dfa_set_leaf(epsilon, IDENTIFIER);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

/*** TRANSITION FROM STATE 3 ***/

dfa_transition_state_group_to_state_group(
  state_3,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

dfa_transition_state_group_to_state_group(
  state_3,
  state_4,
  strlen(STATE_4_TRANSITIONS),
  state_4_transitions
);

dfa_transition_state_group_to_state_group(
  nodes_lowercase,
  state_3,
  strlen(STATE_3_TRANSITIONS),
```

```
  state_3_transitions
);

dfa_transition_state_group_to_state_group(
  nodes_uppercase,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

epsilon = dfa_add_state_group_epsilon(
  dfa, state_3, PREFIX "state_3"
);

dfa_set_leaf(epsilon, IDENTIFIER);
dfa epsilon transition state to state(epsilon, source scope root);

/*** TRANSITION FROM STATE 4 ***/

dfa_transition_state_group_to_state_group(
  state_4,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

dfa_transition_state_group_to_state_group(
  state_4,
  state_4,
  strlen(STATE_4_TRANSITIONS),
  state_4_transitions
);

dfa_transition_state_group_to_state_group(
  nodes_lowercase,
  state_4,
  strlen(STATE_4_TRANSITIONS),
  state_4_transitions
);

dfa_transition_state_group_to_state_group(
  nodes_uppercase,
  state_4,
  strlen(STATE_4_TRANSITIONS),
```

```
    state_4_transitions
  );

  epsilon = dfa_add_state_group_epsilon(
    dfa, state_4, PREFIX "state_4"
  );

  dfa_set_leaf(epsilon, IDENTIFIER);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  /* TRANSITION FROM LOWERCASE */

  epsilon = dfa add state group epsilon(
    dfa, nodes_lowercase, PREFIX "state_lowercase"
  );

  dfa_set_leaf(epsilon, IDENTIFIER);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  /* TRANSITION FROM UPPERCASE */

  epsilon = dfa_add_state_group_epsilon(
    dfa, nodes_uppercase, PREFIX "state_uppercase"
  );

  dfa_set_leaf(epsilon, IDENTIFIER);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  /* Clean-up */

  free(state_2_transitions);
  state_2_transitions = NULL;

  free(state_3_transitions);
  state_3_transitions = NULL;

  free(state_4_transitions);
  state_4_transitions = NULL;

  list_free(state_2);
  list_free(state_3);
  list_free(state_4);
}
```

8.2.4 mod_lexer_dfa_c90_keyword

8.2.4.1 mod_lexer_dfa_c90_keyword.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_c90_keyword_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  List *nodes_lowercase,
  GraphNode **node_type
);
```

8.2.4.2 mod_lexer_dfa_c90_keyword.c

```
#include <stdlib.h>
#include <assert.h>

#include "lib_common.h"
#include "mod_lexer_dfa_c90_keyword.h"
#include "mod_parser.tab.h"

#define PREFIX "keyword_"

#define STATE_TRANSITIONS "  autobreakcaseharonsttinuedefaultouble" \
  "elsenumxternfloatorgotoifntlongregisterturnshortignedzeoftaticr" \
  "uctwitchtypedefunionsignedvoidlatilewhile"

/*
 * lexer_dfa_c90_keyword_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, source_scope_root
 * [param] IOP, nodes lowercase
 * [param] IOP, node_type
```

```
 */
void lexer_dfa_c90_keyword_create(
  const DFA *dfa,
  DFAState *source_scope_root,
  DFAStateGroup *nodes_lowercase,
  DFAState **node_type
) {
  DFAState *epsilon, **states;

  char *label;

  unsigned int i;

  states = malloc(sizeof(DFAState*) * 143);
  assert(states != NULL);

  label = malloc(sizeof(char) * 18);
  assert(label != NULL);

  for (i = 2; i < 143; i += 1) {
    sprintf(label, PREFIX "state_%d", i);
    states[i] = dfa_add_state(dfa, label);
  }

  dfa_transition_state_to_state(
    source_scope_root, states[2], STATE_TRANSITIONS[2]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[6], STATE_TRANSITIONS[6]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[11], STATE_TRANSITIONS[11]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[27], STATE_TRANSITIONS[27]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[39], STATE_TRANSITIONS[39]
  );
```

```
dfa_transition_state_to_state(
  source_scope_root, states[51], STATE_TRANSITIONS[51]
);

dfa_transition_state_to_state(
  source_scope_root, states[58], STATE_TRANSITIONS[58]
);

dfa_transition_state_to_state(
  source_scope_root, states[62], STATE_TRANSITIONS[62]
);

dfa transition state to state(
  source_scope_root, states[66], STATE_TRANSITIONS[66]
);

dfa_transition_state_to_state(
  source_scope_root, states[70], STATE_TRANSITIONS[70]
);

dfa_transition_state_to_state(
  source_scope_root, states[82], STATE_TRANSITIONS[82]
);

dfa_transition_state_to_state(
  source_scope_root, states[110], STATE_TRANSITIONS[110]
);

dfa_transition_state_to_state(
  source_scope_root, states[117], STATE_TRANSITIONS[117]
);

dfa_transition_state_to_state(
  source_scope_root, states[128], STATE_TRANSITIONS[128]
);

dfa_transition_state_to_state(
  source_scope_root, states[138], STATE_TRANSITIONS[138]
);

dfa_transition_state_to_state(
  states[2], states[3], STATE_TRANSITIONS[3]
);
```

```
dfa_transition_state_to_state(
  states[3], states[4], STATE_TRANSITIONS[4]
);

dfa_transition_state_to_state(
  states[4], states[5], STATE_TRANSITIONS[5]
);

dfa_transition_state_to_state(
  states[6], states[7], STATE_TRANSITIONS[7]
);

dfa transition state to state(
  states[7], states[8], STATE_TRANSITIONS[8]
);

dfa_transition_state_to_state(
  states[8], states[9], STATE_TRANSITIONS[9]
);

dfa_transition_state_to_state(
  states[9], states[10], STATE_TRANSITIONS[10]
);

dfa_transition_state_to_state(
  states[11], states[12], STATE_TRANSITIONS[12]
);

dfa_transition_state_to_state(
  states[12], states[13], STATE_TRANSITIONS[13]
);

dfa_transition_state_to_state(
  states[13], states[14], STATE_TRANSITIONS[14]
);

dfa_transition_state_to_state(
  states[11], states[15], STATE_TRANSITIONS[15]
);

dfa_transition_state_to_state(
  states[15], states[16], STATE_TRANSITIONS[16]
);
```

```
  dfa_transition_state_to_state(
    states[16], states[17], STATE_TRANSITIONS[17]
  );

  dfa_transition_state_to_state(
    states[11], states[18], STATE_TRANSITIONS[18]
  );

  dfa_transition_state_to_state(
    states[18], states[19], STATE_TRANSITIONS[19]
  );

  dfa transition state to state(
    states[19], states[20], STATE_TRANSITIONS[20]
  );

  dfa_transition_state_to_state(
    states[20], states[21], STATE_TRANSITIONS[21]
  );

  dfa_transition_state_to_state(
    states[19], states[22], STATE_TRANSITIONS[22]
  );

  dfa_transition_state_to_state(
    states[22], states[23], STATE_TRANSITIONS[23]
  );

  dfa_transition_state_to_state(
    states[23], states[24], STATE_TRANSITIONS[24]
  );

  dfa_transition_state_to_state(
    states[24], states[25], STATE_TRANSITIONS[25]
  );

  dfa_transition_state_to_state(
    states[25], states[26], STATE_TRANSITIONS[26]
  );

  dfa_transition_state_to_state(
    states[27], states[28], STATE_TRANSITIONS[28]
  );
```

```
dfa_transition_state_to_state(
  states[28], states[29], STATE_TRANSITIONS[29]
);

dfa_transition_state_to_state(
  states[29], states[30], STATE_TRANSITIONS[30]
);

dfa_transition_state_to_state(
  states[30], states[31], STATE_TRANSITIONS[31]
);

dfa transition state to state(
  states[31], states[32], STATE_TRANSITIONS[32]
);

dfa_transition_state_to_state(
  states[32], states[33], STATE_TRANSITIONS[33]
);

dfa_transition_state_to_state(
  states[27], states[34], STATE_TRANSITIONS[34]
);

dfa_transition_state_to_state(
  states[34], states[35], STATE_TRANSITIONS[35]
);

dfa_transition_state_to_state(
  states[35], states[36], STATE_TRANSITIONS[36]
);

dfa_transition_state_to_state(
  states[36], states[37], STATE_TRANSITIONS[37]
);

dfa_transition_state_to_state(
  states[37], states[38], STATE_TRANSITIONS[38]
);

dfa_transition_state_to_state(
  states[39], states[40], STATE_TRANSITIONS[40]
);
```

```
  dfa_transition_state_to_state(
    states[40], states[41], STATE_TRANSITIONS[41]
  );

  dfa_transition_state_to_state(
    states[41], states[42], STATE_TRANSITIONS[42]
  );

  dfa_transition_state_to_state(
    states[39], states[43], STATE_TRANSITIONS[43]
  );

  dfa transition state to state(
    states[43], states[44], STATE_TRANSITIONS[44]
  );

  dfa_transition_state_to_state(
    states[44], states[45], STATE_TRANSITIONS[45]
  );

  dfa_transition_state_to_state(
    states[39], states[46], STATE_TRANSITIONS[46]
  );

  dfa_transition_state_to_state(
    states[46], states[47], STATE_TRANSITIONS[47]
  );

  dfa_transition_state_to_state(
    states[47], states[48], STATE_TRANSITIONS[48]
  );

  dfa_transition_state_to_state(
    states[48], states[49], STATE_TRANSITIONS[49]
  );

  dfa_transition_state_to_state(
    states[49], states[50], STATE_TRANSITIONS[50]
  );

  dfa_transition_state_to_state(
    states[51], states[52], STATE_TRANSITIONS[52]
  );
```

```
  dfa_transition_state_to_state(
    states[52], states[53], STATE_TRANSITIONS[53]
  );

  dfa_transition_state_to_state(
    states[53], states[54], STATE_TRANSITIONS[54]
  );

  dfa_transition_state_to_state(
    states[54], states[55], STATE_TRANSITIONS[55]
  );

  dfa transition state to state(
    states[51], states[56], STATE_TRANSITIONS[56]
  );

  dfa_transition_state_to_state(
    states[56], states[57], STATE_TRANSITIONS[57]
  );

  dfa_transition_state_to_state(
    states[58], states[59], STATE_TRANSITIONS[59]
  );

  dfa_transition_state_to_state(
    states[59], states[60], STATE_TRANSITIONS[60]
  );

  dfa_transition_state_to_state(
    states[60], states[61], STATE_TRANSITIONS[61]
  );

  dfa_transition_state_to_state(
    states[62], states[63], STATE_TRANSITIONS[63]
  );

  dfa_transition_state_to_state(
    states[62], states[64], STATE_TRANSITIONS[64]
  );

  dfa_transition_state_to_state(
    states[64], states[65], STATE_TRANSITIONS[65]
  );
```

```
dfa_transition_state_to_state(
  states[66], states[67], STATE_TRANSITIONS[67]
);

dfa_transition_state_to_state(
  states[67], states[68], STATE_TRANSITIONS[68]
);

dfa_transition_state_to_state(
  states[68], states[69], STATE_TRANSITIONS[69]
);

dfa transition state to state(
  states[70], states[71], STATE_TRANSITIONS[71]
);

dfa_transition_state_to_state(
  states[71], states[72], STATE_TRANSITIONS[72]
);

dfa_transition_state_to_state(
  states[72], states[73], STATE_TRANSITIONS[73]
);

dfa_transition_state_to_state(
  states[73], states[74], STATE_TRANSITIONS[74]
);

dfa_transition_state_to_state(
  states[74], states[75], STATE_TRANSITIONS[75]
);

dfa_transition_state_to_state(
  states[75], states[76], STATE_TRANSITIONS[76]
);

dfa_transition_state_to_state(
  states[76], states[77], STATE_TRANSITIONS[77]
);

dfa_transition_state_to_state(
  states[71], states[78], STATE_TRANSITIONS[78]
);
```

```
  dfa_transition_state_to_state(
    states[78], states[79], STATE_TRANSITIONS[79]
  );

  dfa_transition_state_to_state(
    states[79], states[80], STATE_TRANSITIONS[80]
  );

  dfa_transition_state_to_state(
    states[80], states[81], STATE_TRANSITIONS[81]
  );

  dfa transition state to state(
    states[82], states[83], STATE_TRANSITIONS[83]
  );

  dfa_transition_state_to_state(
    states[83], states[84], STATE_TRANSITIONS[84]
  );

  dfa_transition_state_to_state(
    states[84], states[85], STATE_TRANSITIONS[85]
  );

  dfa_transition_state_to_state(
    states[85], states[86], STATE_TRANSITIONS[86]
  );

  dfa_transition_state_to_state(
    states[82], states[87], STATE_TRANSITIONS[87]
  );

  dfa_transition_state_to_state(
    states[87], states[88], STATE_TRANSITIONS[88]
  );

  dfa_transition_state_to_state(
    states[88], states[89], STATE_TRANSITIONS[89]
  );

  dfa_transition_state_to_state(
    states[89], states[90], STATE_TRANSITIONS[90]
  );
```

```
  dfa_transition_state_to_state(
    states[90], states[91], STATE_TRANSITIONS[91]
  );

  dfa_transition_state_to_state(
    states[87], states[92], STATE_TRANSITIONS[92]
  );

  dfa_transition_state_to_state(
    states[92], states[93], STATE_TRANSITIONS[93]
  );

  dfa transition state to state(
    states[93], states[94], STATE_TRANSITIONS[94]
  );

  dfa_transition_state_to_state(
    states[94], states[95], STATE_TRANSITIONS[95]
  );

  dfa_transition_state_to_state(
    states[82], states[96], STATE_TRANSITIONS[96]
  );

  dfa_transition_state_to_state(
    states[96], states[97], STATE_TRANSITIONS[97]
  );

  dfa_transition_state_to_state(
    states[97], states[98], STATE_TRANSITIONS[98]
  );

  dfa_transition_state_to_state(
    states[98], states[99], STATE_TRANSITIONS[99]
  );

  dfa_transition_state_to_state(
    states[99], states[100], STATE_TRANSITIONS[100]
  );

  dfa_transition_state_to_state(
    states[96], states[101], STATE_TRANSITIONS[101]
  );
```

```
dfa_transition_state_to_state(
  states[101], states[102], STATE_TRANSITIONS[102]
);

dfa_transition_state_to_state(
  states[102], states[103], STATE_TRANSITIONS[103]
);

dfa_transition_state_to_state(
  states[103], states[104], STATE_TRANSITIONS[104]
);

dfa transition state to state(
  states[82], states[105], STATE_TRANSITIONS[105]
);

dfa_transition_state_to_state(
  states[105], states[106], STATE_TRANSITIONS[106]
);

dfa_transition_state_to_state(
  states[106], states[107], STATE_TRANSITIONS[107]
);

dfa_transition_state_to_state(
  states[107], states[108], STATE_TRANSITIONS[108]
);

dfa_transition_state_to_state(
  states[108], states[109], STATE_TRANSITIONS[109]
);

dfa_transition_state_to_state(
  states[110], states[111], STATE_TRANSITIONS[111]
);

dfa_transition_state_to_state(
  states[111], states[112], STATE_TRANSITIONS[112]
);

dfa_transition_state_to_state(
  states[112], states[113], STATE_TRANSITIONS[113]
);
```

```
dfa_transition_state_to_state(
  states[113], states[114], STATE_TRANSITIONS[114]
);

dfa_transition_state_to_state(
  states[114], states[115], STATE_TRANSITIONS[115]
);

dfa_transition_state_to_state(
  states[115], states[116], STATE_TRANSITIONS[116]
);

dfa transition state to state(
  states[117], states[118], STATE_TRANSITIONS[118]
);

dfa_transition_state_to_state(
  states[118], states[119], STATE_TRANSITIONS[119]
);

dfa_transition_state_to_state(
  states[119], states[120], STATE_TRANSITIONS[120]
);

dfa_transition_state_to_state(
  states[120], states[121], STATE_TRANSITIONS[121]
);

dfa_transition_state_to_state(
  states[118], states[122], STATE_TRANSITIONS[122]
);

dfa_transition_state_to_state(
  states[122], states[123], STATE_TRANSITIONS[123]
);

dfa_transition_state_to_state(
  states[123], states[124], STATE_TRANSITIONS[124]
);

dfa_transition_state_to_state(
  states[124], states[125], STATE_TRANSITIONS[125]
);
```

```
dfa_transition_state_to_state(
  states[125], states[126], STATE_TRANSITIONS[126]
);

dfa_transition_state_to_state(
  states[126], states[127], STATE_TRANSITIONS[127]
);

dfa_transition_state_to_state(
  states[128], states[129], STATE_TRANSITIONS[129]
);

dfa transition state to state(
  states[129], states[130], STATE_TRANSITIONS[130]
);

dfa_transition_state_to_state(
  states[130], states[131], STATE_TRANSITIONS[131]
);

dfa_transition_state_to_state(
  states[129], states[132], STATE_TRANSITIONS[132]
);

dfa_transition_state_to_state(
  states[132], states[133], STATE_TRANSITIONS[133]
);

dfa_transition_state_to_state(
  states[133], states[134], STATE_TRANSITIONS[134]
);

dfa_transition_state_to_state(
  states[134], states[135], STATE_TRANSITIONS[135]
);

dfa_transition_state_to_state(
  states[135], states[136], STATE_TRANSITIONS[136]
);

dfa_transition_state_to_state(
  states[136], states[137], STATE_TRANSITIONS[137]
);
```

```
dfa_transition_state_to_state(
  states[138], states[139], STATE_TRANSITIONS[139]
);

dfa_transition_state_to_state(
  states[139], states[140], STATE_TRANSITIONS[140]
);

dfa_transition_state_to_state(
  states[140], states[141], STATE_TRANSITIONS[141]
);

dfa transition state to state(
  states[141], states[142], STATE_TRANSITIONS[142]
);

epsilon = dfa_add_state_epsilon(
  dfa, states[5], PREFIX "state_5"
);

dfa_set_leaf(epsilon, AUTO);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[10], PREFIX "state_10"
);

dfa_set_leaf(epsilon, BREAK);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[14], PREFIX "state_14"
);

dfa_set_leaf(epsilon, CASE);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[17], PREFIX "state_17"
);

dfa_set_leaf(epsilon, CHAR);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);
```

```
epsilon = dfa_add_state_epsilon(
  dfa, states[21], PREFIX "state_21"
);

dfa_set_leaf(epsilon, CONST);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[26], PREFIX "state_26"
);

dfa_set_leaf(epsilon, CONTINUE);
dfa epsilon transition state to state(epsilon, source scope root);

epsilon = dfa_add_state_epsilon(
  dfa, states[33], PREFIX "state 33"
);

dfa_set_leaf(epsilon, DEFAULT);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[34], PREFIX "state_34"
);

dfa_set_leaf(epsilon, DO);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[38], PREFIX "state_38"
);

dfa_set_leaf(epsilon, DOUBLE);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[42], PREFIX "state_42"
);

dfa_set_leaf(epsilon, ELSE);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[45], PREFIX "state_45"
```

```
);

dfa_set_leaf(epsilon, ENUM);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[50], PREFIX "state_50"
);

dfa_set_leaf(epsilon, EXTERN);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa add state epsilon(
  dfa, states[55], PREFIX "state_55"
);

dfa_set_leaf(epsilon, FLOAT);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[57], PREFIX "state_57"
);

dfa_set_leaf(epsilon, FOR);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[61], PREFIX "state_61"
);

dfa_set_leaf(epsilon, GOTO);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[63], PREFIX "state_63"
);

dfa_set_leaf(epsilon, IF);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[65], PREFIX "state_65"
);
```

```
  dfa_set_leaf(epsilon, INT);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  epsilon = dfa_add_state_epsilon(
    dfa, states[69], PREFIX "state_69"
  );

  dfa_set_leaf(epsilon, LONG);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  epsilon = dfa_add_state_epsilon(
    dfa, states[77], PREFIX "state_77"
  );

  dfa_set_leaf(epsilon, REGISTER);
  dfa epsilon transition state to state(epsilon, source scope root);

  epsilon = dfa_add_state_epsilon(
    dfa, states[81], PREFIX "state_81"
  );

  dfa_set_leaf(epsilon, RETURN);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  epsilon = dfa_add_state_epsilon(
    dfa, states[86], PREFIX "state_86"
  );

  dfa_set_leaf(epsilon, SHORT);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  epsilon = dfa_add_state_epsilon(
    dfa, states[91], PREFIX "state_91"
  );

  dfa_set_leaf(epsilon, SIGNED);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  epsilon = dfa_add_state_epsilon(
    dfa, states[95], PREFIX "state_95"
  );

  dfa_set_leaf(epsilon, SIZEOF);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);
```

```
epsilon = dfa_add_state_epsilon(
  dfa, states[100], PREFIX "state_100"
);

dfa_set_leaf(epsilon, STATIC);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[104], PREFIX "state_104"
);

dfa set leaf(epsilon, STRUCT);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa add state epsilon(
  dfa, states[109], PREFIX "state_109"
);

dfa_set_leaf(epsilon, SWITCH);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[116], PREFIX "state_116"
);

dfa_set_leaf(epsilon, TYPEDEF);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[121], PREFIX "state_121"
);

dfa_set_leaf(epsilon, UNION);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
  dfa, states[127], PREFIX "state_127"
);

dfa_set_leaf(epsilon, UNSIGNED);
dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

epsilon = dfa_add_state_epsilon(
```

```
    dfa, states[131], PREFIX "state_131"
  );

  dfa_set_leaf(epsilon, VOID);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  epsilon = dfa_add_state_epsilon(
    dfa, states[137], PREFIX "state_137"
  );

  dfa_set_leaf(epsilon, VOLATILE);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  epsilon = dfa_add_state_epsilon(
    dfa, states[142], PREFIX "state_142"
  );

  dfa_set_leaf(epsilon, WHILE);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  for (i = 2; i < 113; i += 1) {
    list_push_back(nodes_lowercase, states[i]);
  }

  *node_type = states[113];

  for (i = 114; i < 143; i += 1) {
    list_push_back(nodes_lowercase, states[i]);
  }

  /* Clean-up */

  free(label);
  label = NULL;

  free(states);
  states = NULL;
}
```

8.2.5 mod_lexer_dfa_c90_literal

8.2.5.1 mod_lexer_dfa_c90_literal.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_c90_literal_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  List *nodes_uppercase
);
```

8.2.5.2 mod_lexer_dfa_c90_literal.c

```
#include <string.h>
#include <stdlib.h>

#include "lib_common.h"
#include "mod_lexer_dfa_c90_literal.h"
#include "mod_parser.tab.h"

#define PREFIX "literal_"

#define STATE_2_TRANSITION '\''
#define STATE_4_TRANSITION '\''
#define STATE_5_TRANSITION '\\'
#define STATE_7_TRANSITION 'x'
#define STATE_9_TRANSITION 'L'
#define STATE_13_TRANSITION '"'
#define STATE_15_TRANSITION '"'
#define STATE_16_TRANSITION '\\'
#define STATE_18_TRANSITION 'x'

#define STATE_3_TRANSITIONS "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijk" \
  "lmnopqrstuvwxyz0123456789!\"#%&()*+,-./:;<=>?[]^_{|}~ \t\v\f"

#define STATE_6_TRANSITIONS "'\"?\\abfnrtv"
```

```
#define STATE_8_TRANSITIONS "0123456789ABCDEFabcdef"

#define STATE_10_TRANSITIONS "01234567"

#define STATE_11_TRANSITIONS "01234567"

#define STATE_12_TRANSITIONS "01234567"

#define STATE_14_TRANSITIONS "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij" \
  "klmnopqrstuvwxyz0123456789!\'#%&()*+,-./:;<=>?[]^_{|}~ \t\v\f"

#define STATE 17 TRANSITIONS "'\"?\\abfnrtv"

#define STATE_19_TRANSITIONS "0123456789ABCDEFabcdef"

#define STATE_20_TRANSITIONS "01234567"

#define STATE_21_TRANSITIONS "01234567"

#define STATE_22_TRANSITIONS "01234567"

/*
 * lexer_dfa_c90_literal_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, source_scope_root
 * [param] IOP, nodes_uppercase
 */
void lexer_dfa_c90_literal_create(
  const DFA *dfa,
  DFAState *source_scope_root,
  DFAStateGroup *nodes_uppercase
) {
  DFAState *state_2   = dfa_add_state(dfa, PREFIX "state_2");
  DFAState *state_4   = dfa_add_state(dfa, PREFIX "state_4");
  DFAState *state_5   = dfa_add_state(dfa, PREFIX "state_5");
  DFAState *state_7   = dfa_add_state(dfa, PREFIX "state_7");
  DFAState *state_9   = dfa_add_state(dfa, PREFIX "state_9");
  DFAState *state_13  = dfa_add_state(dfa, PREFIX "state_13");
  DFAState *state_15  = dfa_add_state(dfa, PREFIX "state_15");
  DFAState *state_16  = dfa_add_state(dfa, PREFIX "state_16");
```

```
DFAState *state_18  = dfa_add_state(dfa, PREFIX "state_18");

DFAStateGroup *state_3 = dfa_add_state_group(
  dfa, strlen(STATE_3_TRANSITIONS), PREFIX "state_3"
);

DFAStateGroup *state_6 = dfa_add_state_group(
  dfa, strlen(STATE_6_TRANSITIONS), PREFIX "state_6"
);

DFAStateGroup *state_8 = dfa_add_state_group(
  dfa, strlen(STATE_8_TRANSITIONS), PREFIX "state_8"
);

DFAStateGroup *state_10 = dfa_add_state_group(
  dfa, strlen(STATE 10 TRANSITIONS), PREFIX "state 10"
);

DFAStateGroup *state_11 = dfa_add_state_group(
  dfa, strlen(STATE_11_TRANSITIONS), PREFIX "state_11"
);

DFAStateGroup *state_12 = dfa_add_state_group(
  dfa, strlen(STATE_12_TRANSITIONS), PREFIX "state_12"
);

DFAStateGroup *state_14 = dfa_add_state_group(
  dfa, strlen(STATE_14_TRANSITIONS), PREFIX "state_14"
);

DFAStateGroup *state_17 = dfa_add_state_group(
  dfa, strlen(STATE_17_TRANSITIONS), PREFIX "state_17"
);

DFAStateGroup *state_19 = dfa_add_state_group(
  dfa, strlen(STATE_19_TRANSITIONS), PREFIX "state_19"
);

DFAStateGroup *state_20 = dfa_add_state_group(
  dfa, strlen(STATE_20_TRANSITIONS), PREFIX "state_20"
);

DFAStateGroup *state_21 = dfa_add_state_group(
  dfa, strlen(STATE_21_TRANSITIONS), PREFIX "state_21"
```

```
);

DFAStateGroup *state_22 = dfa_add_state_group(
  dfa, strlen(STATE_22_TRANSITIONS), PREFIX "state_22"
);

int *state_3_transitions = dfa_string_to_transitions(
  STATE_3_TRANSITIONS
);

int *state_6_transitions = dfa_string_to_transitions(
  STATE_6_TRANSITIONS
);

int *state_8_transitions = dfa_string_to_transitions(
  STATE 8 TRANSITIONS
);

int *state_10_transitions = dfa_string_to_transitions(
  STATE_10_TRANSITIONS
);

int *state_11_transitions = dfa_string_to_transitions(
  STATE_11_TRANSITIONS
);

int *state_12_transitions = dfa_string_to_transitions(
  STATE_12_TRANSITIONS
);

int *state_14_transitions = dfa_string_to_transitions(
  STATE_14_TRANSITIONS
);

int *state_17_transitions = dfa_string_to_transitions(
  STATE_17_TRANSITIONS
);

int *state_19_transitions = dfa_string_to_transitions(
  STATE_19_TRANSITIONS
);

int *state_20_transitions = dfa_string_to_transitions(
  STATE_20_TRANSITIONS
```

```
);

int *state_21_transitions = dfa_string_to_transitions(
  STATE_21_TRANSITIONS
);

int *state_22_transitions = dfa_string_to_transitions(
  STATE_22_TRANSITIONS
);

/*** LEAVES ***/

dfa set leaf(state 4, CONSTANT);
dfa_set_leaf(state_15, STRING_LITERAL);

dfa epsilon transition state to state(state 4, source scope root);
dfa_epsilon_transition_state_to_state(state_15, source_scope_root);

/*** TRANSITION FROM SOURCE SCOPE ROOT ***/

dfa_transition_state_to_state(
  source_scope_root, state_2, STATE_2_TRANSITION
);

dfa_transition_state_to_state(
  source_scope_root, state_9, STATE_9_TRANSITION
);

dfa_transition_state_to_state(
  source_scope_root, state_13, STATE_13_TRANSITION
);

/*** TRANSITION FROM STATE 2 ***/

dfa_transition_state_to_state_group(
  state_2,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

dfa_transition_state_to_state(
  state_2, state_5, STATE_5_TRANSITION
);
```

```
/*** TRANSITION FROM STATE 3 ***/

dfa_transition_state_group_to_state_group(
  state_3,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

dfa_transition_state_group_to_state(
  state_3, state_4, STATE_4_TRANSITION
);

dfa_transition_state_group_to_state(
  state 3, state 5, STATE 5 TRANSITION
);

/*** TRANSITION FROM STATE 5 ***/

dfa_transition_state_to_state_group(
  state_5,
  state_6,
  strlen(STATE_6_TRANSITIONS),
  state_6_transitions
);

dfa_transition_state_to_state(
  state_5, state_7, STATE_7_TRANSITION
);

dfa_transition_state_to_state_group(
  state_5,
  state_10,
  strlen(STATE_10_TRANSITIONS),
  state_10_transitions
);

/*** TRANSITION FROM STATE 6 ***/

dfa_transition_state_group_to_state_group(
  state_6,
  state_3,
  strlen(STATE_3_TRANSITIONS),
```

```
  state_3_transitions
);

dfa_transition_state_group_to_state(
  state_6, state_4, STATE_4_TRANSITION
);

dfa_transition_state_group_to_state(
  state_6, state_5, STATE_5_TRANSITION
);

/*** TRANSITION FROM STATE 7 ***/

dfa_transition_state_to_state_group(
  state_7,
  state 8,
  strlen(STATE_8_TRANSITIONS),
  state_8_transitions
);

/*** TRANSITION FROM STATE 8 ***/

dfa_transition_state_group_to_state_group(
  state_8,
  state_8,
  strlen(STATE_8_TRANSITIONS),
  state_8_transitions
);

dfa_transition_state_group_to_state_group(
  state_8,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

dfa_transition_state_group_to_state(
  state_8, state_4, STATE_4_TRANSITION
);

dfa_transition_state_group_to_state(
  state_8, state_5, STATE_5_TRANSITION
);
```

```
/*** TRANSITION FROM STATE 9 ***/

dfa_transition_state_to_state(
  state_9, state_2, STATE_2_TRANSITION
);

dfa_transition_state_to_state(
  state_9, state_13, STATE_13_TRANSITION
);

/*** TRANSITION FROM STATE 10 ***/

dfa transition state group to state group(
  state_10,
  state_11,
  strlen(STATE 11 TRANSITIONS),
  state_11_transitions
);

dfa_transition_state_group_to_state_group(
  state_10,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

dfa_transition_state_group_to_state(
  state_10, state_4, STATE_4_TRANSITION
);

dfa_transition_state_group_to_state(
  state_10, state_5, STATE_5_TRANSITION
);

/*** TRANSITION FROM STATE 11 ***/

dfa_transition_state_group_to_state_group(
  state_11,
  state_12,
  strlen(STATE_12_TRANSITIONS),
  state_12_transitions
);

dfa_transition_state_group_to_state_group(
```

```
    state_11,
    state_3,
    strlen(STATE_3_TRANSITIONS),
    state_3_transitions
  );

  dfa_transition_state_group_to_state(
    state_11, state_4, STATE_4_TRANSITION
  );

  dfa_transition_state_group_to_state(
    state_11, state_5, STATE_5_TRANSITION
  );

  /*** TRANSITION FROM STATE 12 ***/

  dfa_transition_state_group_to_state_group(
    state_12,
    state_3,
    strlen(STATE_3_TRANSITIONS),
    state_3_transitions
  );

  dfa_transition_state_group_to_state(
    state_12, state_4, STATE_4_TRANSITION
  );

  dfa_transition_state_group_to_state(
    state_12, state_5, STATE_5_TRANSITION
  );

  /*** TRANSITION FROM STATE 13 ***/

  dfa_transition_state_to_state_group(
    state_13,
    state_14,
    strlen(STATE_14_TRANSITIONS),
    state_14_transitions
  );

  dfa_transition_state_to_state(
    state_13, state_15, STATE_15_TRANSITION
  );
```

```
  dfa_transition_state_to_state(
    state_13, state_16, STATE_16_TRANSITION
  );

  /*** TRANSITION FROM STATE 14 ***/

  dfa_transition_state_group_to_state_group(
    state_14,
    state_14,
    strlen(STATE_14_TRANSITIONS),
    state_14_transitions
  );

  dfa_transition_state_group_to_state(
    state_14, state_15, STATE_15_TRANSITION
  );

  dfa_transition_state_group_to_state(
    state_14, state_16, STATE_16_TRANSITION
  );

  /*** TRANSITION FROM STATE 16 ***/

  dfa_transition_state_to_state_group(
    state_16,
    state_17,
    strlen(STATE_17_TRANSITIONS),
    state_17_transitions
  );

  dfa_transition_state_to_state(
    state_16, state_18, STATE_18_TRANSITION
  );

  dfa_transition_state_to_state_group(
    state_16,
    state_20,
    strlen(STATE_20_TRANSITIONS),
    state_20_transitions
  );

  /*** TRANSITION FROM STATE 17 ***/

  dfa_transition_state_group_to_state_group(
```

```
  state_17,
  state_14,
  strlen(STATE_14_TRANSITIONS),
  state_14_transitions
);

dfa_transition_state_group_to_state(
  state_17, state_15, STATE_15_TRANSITION
);

dfa_transition_state_group_to_state(
  state_17, state_16, STATE_16_TRANSITION
);

/*** TRANSITION FROM STATE 18 ***/

dfa_transition_state_to_state_group(
  state_18,
  state_19,
  strlen(STATE_19_TRANSITIONS),
  state_19_transitions
);

/*** TRANSITION FROM STATE 19 ***/

dfa_transition_state_group_to_state_group(
  state_19,
  state_19,
  strlen(STATE_19_TRANSITIONS),
  state_19_transitions
);

dfa_transition_state_group_to_state_group(
  state_19,
  state_14,
  strlen(STATE_14_TRANSITIONS),
  state_14_transitions
);

dfa_transition_state_group_to_state(
  state_19, state_15, STATE_15_TRANSITION
);

dfa_transition_state_group_to_state(
```

```
  state_19, state_16, STATE_16_TRANSITION
);

/*** TRANSITION FROM STATE 20 ***/

dfa_transition_state_group_to_state_group(
  state_20,
  state_21,
  strlen(STATE_21_TRANSITIONS),
  state_21_transitions
);

dfa transition state group to state group(
  state_20,
  state_14,
  strlen(STATE 14 TRANSITIONS),
  state_14_transitions
);

dfa_transition_state_group_to_state(
  state_20, state_15, STATE_15_TRANSITION
);

dfa_transition_state_group_to_state(
  state_20, state_16, STATE_16_TRANSITION
);

/*** TRANSITION FROM STATE 21 ***/

dfa_transition_state_group_to_state_group(
  state_21,
  state_22,
  strlen(STATE_22_TRANSITIONS),
  state_22_transitions
);

dfa_transition_state_group_to_state_group(
  state_21,
  state_14,
  strlen(STATE_14_TRANSITIONS),
  state_14_transitions
);

dfa_transition_state_group_to_state(
```

```
  state_21, state_15, STATE_15_TRANSITION
);

dfa_transition_state_group_to_state(
  state_21, state_16, STATE_16_TRANSITION
);

/*** TRANSITION FROM STATE 22 ***/

dfa_transition_state_group_to_state_group(
  state_22,
  state_14,
  strlen(STATE 14 TRANSITIONS),
  state_14_transitions
);

dfa_transition_state_group_to_state(
  state_22, state_15, STATE_15_TRANSITION
);

dfa_transition_state_group_to_state(
  state_22, state_16, STATE_16_TRANSITION
);

list_push_back(nodes_uppercase, state_9);

/* Clean up */

free(state_3_transitions);
state_3_transitions = NULL;

free(state_6_transitions);
state_6_transitions = NULL;

free(state_8_transitions);
state_8_transitions = NULL;

free(state_10_transitions);
state_10_transitions = NULL;

free(state_11_transitions);
state_11_transitions = NULL;

free(state_12_transitions);
```

```
  state_12_transitions = NULL;

  free(state_14_transitions);
  state_14_transitions = NULL;

  free(state_17_transitions);
  state_17_transitions = NULL;

  free(state_19_transitions);
  state_19_transitions = NULL;

  free(state_20_transitions);
  state 20 transitions = NULL;

  free(state_21_transitions);
  state 21 transitions = NULL;

  free(state_22_transitions);
  state_22_transitions = NULL;

  list_free(state_3);
  list_free(state_6);
  list_free(state_8);
  list_free(state_10);
  list_free(state_11);
  list_free(state_12);
  list_free(state_14);
  list_free(state_17);
  list_free(state_19);
  list_free(state_20);
  list_free(state_21);
  list_free(state_22);
}
```

8.2.6 mod_lexer_dfa_c90_numeric

8.2.6.1 mod_lexer_dfa_c90_numeric.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_c90_numeric_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  GraphNode *node_dot
);
```

8.2.6.2 mod_lexer_dfa_c90_numeric.c

```
#include <stdlib.h>
#include <string.h>

#include "lib_common.h"
#include "mod_lexer_dfa_c90_numeric.h"
#include "mod_parser.tab.h"

#define PREFIX "numeric_"

#define STATE_4_TRANSITION '.'
#define STATE_10_TRANSITION '0'

#define STATE_2_TRANSITIONS "123456789"
#define STATE_3_TRANSITIONS "0123456789"
#define STATE_5_TRANSITIONS "0123456789"
#define STATE_6_TRANSITIONS "eE"
#define STATE_7_TRANSITIONS "+-"
#define STATE_8_TRANSITIONS "0123456789"
#define STATE_9_TRANSITIONS "fFlL"
#define STATE_11_TRANSITIONS "01234567"
#define STATE_12_TRANSITIONS "uU"
#define STATE 13 TRANSITIONS "lL"
#define STATE_14_TRANSITIONS "xX"
```

```
#define STATE_15_TRANSITIONS "0123456789abcdefABCDEF"
#define STATE_16_TRANSITIONS "lL"
#define STATE_17_TRANSITIONS "uU"

/*
 * lexer_dfa_c90_numeric_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, source_scope_root
 * [param] IOP, node_dot
 */
void lexer_dfa_c90_numeric_create(
  const DFA *dfa,
  DFAState *source scope root,
  DFAState *node_dot
) {
  DFAState *epsilon;

  DFAState *state_4 = dfa_add_state(dfa, PREFIX "state_4");
  DFAState *state_10 = dfa_add_state(dfa, PREFIX "state_10");

  DFAStateGroup *state_2 = dfa_add_state_group(
    dfa, strlen(STATE_2_TRANSITIONS), PREFIX "state_2"
  );

  DFAStateGroup *state_3 = dfa_add_state_group(
    dfa, strlen(STATE_3_TRANSITIONS), PREFIX "state_3"
  );

  DFAStateGroup *state_5 = dfa_add_state_group(
    dfa, strlen(STATE_5_TRANSITIONS), PREFIX "state_5"
  );

  DFAStateGroup *state_6 = dfa_add_state_group(
    dfa, strlen(STATE_6_TRANSITIONS), PREFIX "state_6"
  );

  DFAStateGroup *state_7 = dfa_add_state_group(
    dfa, strlen(STATE_7_TRANSITIONS), PREFIX "state_7"
  );

  DFAStateGroup *state_8 = dfa_add_state_group(
```

```
  dfa, strlen(STATE_8_TRANSITIONS), PREFIX "state_8"
);

DFAStateGroup *state_9 = dfa_add_state_group(
  dfa, strlen(STATE_9_TRANSITIONS), PREFIX "state_9"
);

DFAStateGroup *state_11 = dfa_add_state_group(
  dfa, strlen(STATE_11_TRANSITIONS), PREFIX "state_11"
);

DFAStateGroup *state_12 = dfa_add_state_group(
  dfa, strlen(STATE 12 TRANSITIONS), PREFIX "state 12"
);

DFAStateGroup *state 13 = dfa add state group(
  dfa, strlen(STATE_13_TRANSITIONS), PREFIX "state_13"
);

DFAStateGroup *state_14 = dfa_add_state_group(
  dfa, strlen(STATE_14_TRANSITIONS), PREFIX "state_14"
);

DFAStateGroup *state_15 = dfa_add_state_group(
  dfa, strlen(STATE_15_TRANSITIONS), PREFIX "state_15"
);

DFAStateGroup *state_16 = dfa_add_state_group(
  dfa, strlen(STATE_16_TRANSITIONS), PREFIX "state_16"
);

DFAStateGroup *state_17 = dfa_add_state_group(
  dfa, strlen(STATE_17_TRANSITIONS), PREFIX "state_17"
);

int *state_2_transitions = dfa_string_to_transitions(
  STATE_2_TRANSITIONS
);

int *state_3_transitions = dfa_string_to_transitions(
  STATE_3_TRANSITIONS
);

int *state_5_transitions = dfa_string_to_transitions(
```

```
  STATE_5_TRANSITIONS
);

int *state_6_transitions = dfa_string_to_transitions(
  STATE_6_TRANSITIONS
);

int *state_7_transitions = dfa_string_to_transitions(
  STATE_7_TRANSITIONS
);

int *state_8_transitions = dfa_string_to_transitions(
  STATE 8 TRANSITIONS
);

int *state 9 transitions = dfa string to transitions(
  STATE_9_TRANSITIONS
);

int *state_11_transitions = dfa_string_to_transitions(
  STATE_11_TRANSITIONS
);

int *state_12_transitions = dfa_string_to_transitions(
  STATE_12_TRANSITIONS
);

int *state_13_transitions = dfa_string_to_transitions(
  STATE_13_TRANSITIONS
);

int *state_14_transitions = dfa_string_to_transitions(
  STATE_14_TRANSITIONS
);

int *state_15_transitions = dfa_string_to_transitions(
  STATE_15_TRANSITIONS
);

int *state_16_transitions = dfa_string_to_transitions(
  STATE_16_TRANSITIONS
);

int *state_17_transitions = dfa_string_to_transitions(
```

```
    STATE_17_TRANSITIONS
  );

  /*** NODE 1 CHILDREN ***/

  dfa_transition_state_to_state_group(
    source_scope_root,
    state_2,
    strlen(STATE_2_TRANSITIONS),
    state_2_transitions
  );

  dfa transition state to state(
    source_scope_root, state_4, STATE_4_TRANSITION
  );

  dfa_transition_state_to_state(
    source_scope_root, state_10, STATE_10_TRANSITION
  );

  /*** NODE 2 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_2,
    state_3,
    strlen(STATE_3_TRANSITIONS),
    state_3_transitions
  );

  dfa_transition_state_group_to_state(
    state_2, state_4, STATE_4_TRANSITION
  );

  dfa_transition_state_group_to_state_group(
    state_2,
    state_12,
    strlen(STATE_12_TRANSITIONS),
    state_12_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_2,
    state_16,
    strlen(STATE_16_TRANSITIONS),
```

```
  state_16_transitions
);

epsilon = dfa_add_state_group_epsilon(
  dfa, state_2, PREFIX "state_2"
);

dfa_set_leaf(epsilon, CONSTANT);

dfa_epsilon_transition_state_to_state(
  epsilon,
  source_scope_root
);

/*** NODE 3 CHILDREN ***/

dfa_transition_state_group_to_state_group(
  state_3,
  state_3,
  strlen(STATE_3_TRANSITIONS),
  state_3_transitions
);

dfa_transition_state_group_to_state(
  state_3, state_4, STATE_4_TRANSITION
);

dfa_transition_state_group_to_state_group(
  state_3,
  state_12,
  strlen(STATE_12_TRANSITIONS),
  state_12_transitions
);

dfa_transition_state_group_to_state_group(
  state_3,
  state_16,
  strlen(STATE_16_TRANSITIONS),
  state_16_transitions
);

epsilon = dfa_add_state_group_epsilon(
  dfa, state_3, PREFIX "state_3"
);
```

```
dfa_set_leaf(epsilon, CONSTANT);

dfa_epsilon_transition_state_to_state(
  epsilon,
  source_scope_root
);

/*** NODE 4 CHILDREN ***/

dfa_transition_state_to_state_group(
  state_4,
  state 5,
  strlen(STATE_5_TRANSITIONS),
  state_5_transitions
);

/*** NODE 5 CHILDREN ***/

dfa_transition_state_group_to_state_group(
  state_5,
  state_5,
  strlen(STATE_5_TRANSITIONS),
  state_5_transitions
);

dfa_transition_state_group_to_state_group(
  state_5,
  state_6,
  strlen(STATE_6_TRANSITIONS),
  state_6_transitions
);

dfa_transition_state_group_to_state_group(
  state_5,
  state_9,
  strlen(STATE_9_TRANSITIONS),
  state_9_transitions
);

epsilon = dfa_add_state_group_epsilon(
  dfa, state_5, PREFIX "state_5"
);
```

```
  dfa_set_leaf(epsilon, CONSTANT);

  dfa_epsilon_transition_state_to_state(
    epsilon,
    source_scope_root
  );

  /*** NODE 6 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_6,
    state_7,
    strlen(STATE 7 TRANSITIONS),
    state_7_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_6,
    state_8,
    strlen(STATE_8_TRANSITIONS),
    state_8_transitions
  );

  /*** NODE 7 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_7,
    state_8,
    strlen(STATE_8_TRANSITIONS),
    state_8_transitions
  );

  /*** NODE 8 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_8,
    state_8,
    strlen(STATE_8_TRANSITIONS),
    state_8_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_8,
    state_9,
```

```
    strlen(STATE_9_TRANSITIONS),
    state_9_transitions
  );

  epsilon = dfa_add_state_group_epsilon(
    dfa, state_8, PREFIX "state_8"
  );

  dfa_set_leaf(epsilon, CONSTANT);

  dfa_epsilon_transition_state_to_state(
    epsilon,
    source scope root
  );

  /*** NODE 9 ***/

  dfa_set_leaves(state_9, CONSTANT);

  dfa_epsilon_transition_state_group_to_state(
    state_9, source_scope_root
  );

  /*** NODE 10 CHILDREN ***/

  dfa_transition_state_to_state(
    state_10, state_4, STATE_4_TRANSITION
  );

  dfa_transition_state_to_state_group(
    state_10,
    state_11,
    strlen(STATE_11_TRANSITIONS),
    state_11_transitions
  );

  dfa_transition_state_to_state_group(
    state_10,
    state_14,
    strlen(STATE_14_TRANSITIONS),
    state_14_transitions
  );

  dfa_transition_state_to_state_group(
```

```
    state_10,
    state_12,
    strlen(STATE_12_TRANSITIONS),
    state_12_transitions
  );

  dfa_transition_state_to_state_group(
    state_10,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );

  epsilon = dfa_add_state_epsilon(dfa, state_10, PREFIX "state_10");
  dfa_set_leaf(epsilon, CONSTANT);

  dfa_epsilon_transition_state_to_state(
    epsilon,
    source_scope_root
  );

  /*** NODE 11 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_11,
    state_11,
    strlen(STATE_11_TRANSITIONS),
    state_11_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_11,
    state_12,
    strlen(STATE_12_TRANSITIONS),
    state_12_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_11,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );
```

```
  epsilon = dfa_add_state_group_epsilon(
    dfa, state_11, PREFIX "state_11"
  );

  dfa_set_leaf(epsilon, CONSTANT);

  dfa_epsilon_transition_state_to_state(
    epsilon,
    source_scope_root
  );

  /*** NODE 12 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_12,
    state 13,
    strlen(STATE_13_TRANSITIONS),
    state_13_transitions
  );

  epsilon = dfa_add_state_group_epsilon(
    dfa, state_12, PREFIX "state_12"
  );

  dfa_set_leaf(epsilon, CONSTANT);

  dfa_epsilon_transition_state_to_state(
    epsilon,
    source_scope_root
  );

  /*** NODE 13 ***/

  dfa_set_leaves(state_13, CONSTANT);

  dfa_epsilon_transition_state_group_to_state(
    state_13, source_scope_root
  );

  /*** NODE 14 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_14,
    state_15,
```

```
    strlen(STATE_15_TRANSITIONS),
    state_15_transitions
  );

  /*** NODE 15 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_15,
    state_15,
    strlen(STATE_15_TRANSITIONS),
    state_15_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_15,
    state 12,
    strlen(STATE_12_TRANSITIONS),
    state_12_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_15,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );

  epsilon = dfa_add_state_group_epsilon(
    dfa, state_15, PREFIX "state_15"
  );

  dfa_set_leaf(epsilon, CONSTANT);

  dfa_epsilon_transition_state_to_state(
    epsilon,
    source_scope_root
  );

  /*** NODE 16 CHILDREN ***/

  dfa_transition_state_group_to_state_group(
    state_16,
    state_17,
    strlen(STATE_17_TRANSITIONS),
```

```
  state_17_transitions
);

epsilon = dfa_add_state_group_epsilon(
  dfa, state_16, PREFIX "state_16"
);

dfa_set_leaf(epsilon, CONSTANT);

dfa_epsilon_transition_state_to_state(
  epsilon,
  source_scope_root
);

/*** NODE 17 ***/

dfa_set_leaves(state_17, CONSTANT);

dfa_epsilon_transition_state_group_to_state(
  state_17, source_scope_root
);

/*** NODE DOT ***/
dfa_transition_state_to_state_group(
  node_dot,
  state_5,
  strlen(STATE_5_TRANSITIONS),
  state_5_transitions
);

/* Clean-up */

free(state_2_transitions);
state_2_transitions = NULL;

free(state_3_transitions);
state_3_transitions = NULL;

free(state_5_transitions);
state_5_transitions = NULL;

free(state_6_transitions);
state_6_transitions = NULL;
```

```
    free(state_7_transitions);
    state_7_transitions = NULL;

    free(state_8_transitions);
    state_8_transitions = NULL;

    free(state_9_transitions);
    state_9_transitions = NULL;

    free(state_11_transitions);
    state_11_transitions = NULL;

    free(state 12 transitions);
    state_12_transitions = NULL;

    free(state 13 transitions);
    state_13_transitions = NULL;

    free(state_14_transitions);
    state_14_transitions = NULL;

    free(state_15_transitions);
    state_15_transitions = NULL;

    free(state_16_transitions);
    state_16_transitions = NULL;

    free(state_17_transitions);
    state_17_transitions = NULL;

    list_free(state_2);
    list_free(state_3);
    list_free(state_5);
    list_free(state_6);
    list_free(state_7);
    list_free(state_8);
    list_free(state_9);
    list_free(state_11);
    list_free(state_12);
    list_free(state_13);
    list_free(state_14);
    list_free(state_15);
    list_free(state_16);
```

```
  list_free(state_17);
}
```

8.2.7 mod_lexer_dfa_c90_skip

8.2.7.1 mod_lexer_dfa_c90_skip.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_c90_skip_create(
  const DFA *dfa,
  GraphNode *source_scope_root
);
```

8.2.7.2 mod_lexer_dfa_c90_skip.c

```
#include <string.h>
#include <stdlib.h>

#include "lib_common.h"
#include "mod_lexer_dfa_c90_skip.h"

#define PREFIX "skip_"

#define STATE_3_TRANSITION '#'

#define STATE_2_TRANSITIONS " \n\t\v\r\f"
#define STATE_4_TRANSITIONS "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijk" \
  "lmnopqrstuvwxyz0123456789!\'\"\\#%&()*+,-./:;<=>?[]^_{|}~ \t\v\f"

/*
 * lexer_dfa_c90_skip_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, source_scope_root
 */
void lexer dfa c90 skip create(
  const DFA *dfa,
```

```
  GraphNode *source_scope_root
) {
  DFAStateGroup *state_2 = dfa_add_state_group(
    dfa, strlen(STATE_2_TRANSITIONS), PREFIX "state_2"
  );

  DFAStateGroup *state_4 = dfa_add_state_group(
    dfa, strlen(STATE_4_TRANSITIONS), PREFIX "state_4"
  );

  DFAState *state_3  = dfa_add_state(dfa, PREFIX "state_3");

  int *state_2_transitions = dfa_string_to_transitions(
    STATE_2_TRANSITIONS
  );

  int *state_4_transitions = dfa_string_to_transitions(
    STATE_4_TRANSITIONS
  );

  dfa_transition_state_to_state_group(
    source_scope_root,
    state_2,
    strlen(STATE_2_TRANSITIONS),
    state_2_transitions
  );

  dfa_epsilon_transition_state_group_to_state(
    state_2,
    source_scope_root
  );

  dfa_transition_state_to_state(
    source_scope_root,
    state_3,
    STATE_3_TRANSITION
  );

  dfa_transition_state_to_state_group(
    state_3,
    state_4,
    strlen(STATE_4_TRANSITIONS),
    state_4_transitions
  );
```

```
  dfa_transition_state_group_to_state_group(
    state_4,
    state_4,
    strlen(STATE_4_TRANSITIONS),
    state_4_transitions
  );

  dfa_epsilon_transition_state_group_to_state(
    state_4,
    source_scope_root
  );

  /* Clean-up */

  free(state 2 transitions);
  state_2_transitions = NULL;

  free(state_4_transitions);
  state_4_transitions = NULL;

  list_free(state_2);
  list_free(state_4);
}
```

8.2.8 mod_lexer_dfa_c90_symbol

8.2.8.1 mod_lexer_dfa_c90_symbol.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_c90_symbol_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  GraphNode *comment_scope_root,
  GraphNode **state_dot
);
```

8.2.8.2 mod_lexer_dfa_c90_symbol.c

```
#include <stdlib.h>
#include <assert.h>

#include "lib_common.h"
#include "mod_lexer_dfa_c90_symbol.h"
#include "mod_parser.tab.h"

#define PREFIX "symbol_"

#define STATE_TRANSITIONS "  [](){}--><=++=&&=*=~!=:?{};/=*%=<<==>>==" \
  "==^=||=,..."

/*
 * lexer_dfa_c90_symbol_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, source_scope_root
 * [param] IOP, comment_scope_root
 * [param] IOP, state dot
 */
```

```
void lexer_dfa_c90_symbol_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  GraphNode *comment_scope_root,
  GraphNode **state_dot
) {
  GraphNode *epsilon, **states;

  unsigned int i;

  char *label;

  states = malloc(sizeof(DFAState*) * 50);
  assert(states != NULL);

  label = malloc(sizeof(char) * 16);
  assert(label != NULL);

  for (i = 2; i < 50; i += 1) {
    sprintf(label, PREFIX "state_%d", i);
    states[i] = dfa_add_state(dfa, label);
  }

  dfa_transition_state_to_state(
    source_scope_root, states[2], STATE_TRANSITIONS[2]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[3], STATE_TRANSITIONS[3]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[4], STATE_TRANSITIONS[4]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[5], STATE_TRANSITIONS[5]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[6], STATE_TRANSITIONS[6]
  );

  dfa_transition_state_to_state(
```

```
    source_scope_root, states[10], STATE_TRANSITIONS[10]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[13], STATE_TRANSITIONS[13]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[16], STATE_TRANSITIONS[16]
  );

  dfa_transition_state_to_state(
    source scope root, states[18], STATE TRANSITIONS[18]
  );

  dfa transition state to state(
    source_scope_root, states[19], STATE_TRANSITIONS[19]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[21], STATE_TRANSITIONS[21]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[22], STATE_TRANSITIONS[22]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[23], STATE_TRANSITIONS[23]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[24], STATE_TRANSITIONS[24]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[25], STATE_TRANSITIONS[25]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[26], STATE_TRANSITIONS[26]
  );

  dfa_transition_state_to_state(
```

```
    source_scope_root, states[29], STATE_TRANSITIONS[29]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[31], STATE_TRANSITIONS[31]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[35], STATE_TRANSITIONS[35]
  );

  dfa_transition_state_to_state(
    source scope root, states[39], STATE TRANSITIONS[39]
  );

  dfa transition state to state(
    source_scope_root, states[41], STATE_TRANSITIONS[41]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[43], STATE_TRANSITIONS[43]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[46], STATE_TRANSITIONS[46]
  );

  dfa_transition_state_to_state(
    source_scope_root, states[47], STATE_TRANSITIONS[47]
  );

  dfa_transition_state_to_state(
    states[6], states[7], STATE_TRANSITIONS[7]
  );

  dfa_transition_state_to_state(
    states[6], states[8], STATE_TRANSITIONS[8]
  );

  dfa_transition_state_to_state(
    states[6], states[9], STATE_TRANSITIONS[9]
  );

  dfa_transition_state_to_state(
```

```
    states[10], states[11], STATE_TRANSITIONS[11]
  );

  dfa_transition_state_to_state(
    states[10], states[12], STATE_TRANSITIONS[12]
  );

  dfa_transition_state_to_state(
    states[13], states[14], STATE_TRANSITIONS[14]
  );

  dfa_transition_state_to_state(
    states[13], states[15], STATE TRANSITIONS[15]
  );

  dfa transition state to state(
    states[16], states[17], STATE_TRANSITIONS[17]
  );

  dfa_transition_state_to_state(
    states[19], states[20], STATE_TRANSITIONS[20]
  );

  dfa_transition_state_to_state(
    states[26], states[27], STATE_TRANSITIONS[27]
  );

  dfa_transition_state_to_state(
    states[26], states[28], STATE_TRANSITIONS[28]
  );

  dfa_transition_state_to_state(
    states[29], states[30], STATE_TRANSITIONS[30]
  );

  dfa_transition_state_to_state(
    states[31], states[32], STATE_TRANSITIONS[32]
  );

  dfa_transition_state_to_state(
    states[31], states[34], STATE_TRANSITIONS[34]
  );

  dfa_transition_state_to_state(
```

```
    states[32], states[33], STATE_TRANSITIONS[33]
  );

  dfa_transition_state_to_state(
    states[35], states[36], STATE_TRANSITIONS[36]
  );

  dfa_transition_state_to_state(
    states[35], states[38], STATE_TRANSITIONS[38]
  );

  dfa_transition_state_to_state(
    states[36], states[37], STATE TRANSITIONS[37]
  );

  dfa transition state to state(
    states[39], states[40], STATE_TRANSITIONS[40]
  );

  dfa_transition_state_to_state(
    states[41], states[42], STATE_TRANSITIONS[42]
  );

  dfa_transition_state_to_state(
    states[43], states[44], STATE_TRANSITIONS[44]
  );

  dfa_transition_state_to_state(
    states[43], states[45], STATE_TRANSITIONS[45]
  );

  dfa_transition_state_to_state(
    states[47], states[48], STATE_TRANSITIONS[48]
  );

  dfa_transition_state_to_state(
    states[48], states[49], STATE_TRANSITIONS[49]
  );

  dfa_set_leaf(states[2], SO_BRACKET);

  dfa_epsilon_transition_state_to_state(
    states[2], source_scope_root
  );
```

```
dfa_set_leaf(states[3], SC_BRACKET);

dfa_epsilon_transition_state_to_state(
  states[3], source_scope_root
);

dfa_set_leaf(states[4], RO_BRACKET);

dfa_epsilon_transition_state_to_state(
  states[4], source_scope_root
);

dfa_set_leaf(states[5], RC_BRACKET);

dfa epsilon transition state to state(
  states[5], source_scope_root
);

dfa_set_leaf(states[7], DEC_OP);

dfa_epsilon_transition_state_to_state(
  states[7], source_scope_root
);

dfa_set_leaf(states[8], PTR_OP);

dfa_epsilon_transition_state_to_state(
  states[8], source_scope_root
);

dfa_set_leaf(states[9], SUB_ASSIGN);

dfa_epsilon_transition_state_to_state(
  states[9], source_scope_root
);

dfa_set_leaf(states[11], INC_OP);

dfa_epsilon_transition_state_to_state(
  states[11], source_scope_root
);

dfa_set_leaf(states[12], ADD_ASSIGN);
```

```
dfa_epsilon_transition_state_to_state(
  states[12], source_scope_root
);

dfa_set_leaf(states[14], AND_OP);

dfa_epsilon_transition_state_to_state(
  states[14], source_scope_root
);

dfa_set_leaf(states[15], AND_ASSIGN);

dfa_epsilon_transition_state_to_state(
  states[15], source_scope_root
);

dfa_set_leaf(states[17], MUL_ASSIGN);

dfa_epsilon_transition_state_to_state(
  states[17], source_scope_root
);

dfa_set_leaf(states[18], BITWISE_NOT_OP);

dfa_epsilon_transition_state_to_state(
  states[18], source_scope_root
);

dfa_set_leaf(states[20], NE_OP);

dfa_epsilon_transition_state_to_state(
  states[20], source_scope_root
);

dfa_set_leaf(states[21], COLON);

dfa_epsilon_transition_state_to_state(
  states[21], source_scope_root
);

dfa_set_leaf(states[22], TERNARY_OP);

dfa_epsilon_transition_state_to_state(
```

```
    states[22], source_scope_root
  );

  dfa_set_leaf(states[23], CO_BRACKET);

  dfa_epsilon_transition_state_to_state(
    states[23], source_scope_root
  );

  dfa_set_leaf(states[24], CC_BRACKET);

  dfa_epsilon_transition_state_to_state(
    states[24], source scope root
  );

  dfa set leaf(states[25], SEMICOLON);

  dfa_epsilon_transition_state_to_state(
    states[25], source_scope_root
  );

  dfa_set_leaf(states[27], DIV_ASSIGN);

  dfa_epsilon_transition_state_to_state(
    states[27], source_scope_root
  );

  dfa_set_leaf(states[28], OPEN_COMMENT);

  dfa_epsilon_transition_state_to_state(
    states[28], comment_scope_root
  );

  dfa_set_leaf(states[30], MOD_ASSIGN);

  dfa_epsilon_transition_state_to_state(
    states[30], source_scope_root
  );

  dfa_set_leaf(states[33], LEFT_ASSIGN);

  dfa_epsilon_transition_state_to_state(
    states[33], source_scope_root
  );
```

```
dfa_set_leaf(states[34], LE_OP);

dfa_epsilon_transition_state_to_state(
  states[34], source_scope_root
);

dfa_set_leaf(states[37], RIGHT_ASSIGN);

dfa_epsilon_transition_state_to_state(
  states[37], source_scope_root
);

dfa_set_leaf(states[38], GE_OP);

dfa epsilon transition state to state(
  states[38], source_scope_root
);

dfa_set_leaf(states[40], EQ_OP);

dfa_epsilon_transition_state_to_state(
  states[40], source_scope_root
);

dfa_set_leaf(states[42], XOR_ASSIGN);

dfa_epsilon_transition_state_to_state(
  states[42], source_scope_root
);

dfa_set_leaf(states[44], OR_OP);

dfa_epsilon_transition_state_to_state(
  states[44], source_scope_root
);

dfa_set_leaf(states[45], OR_ASSIGN);

dfa_epsilon_transition_state_to_state(
  states[45], source_scope_root
);

dfa_set_leaf(states[46], COMMA);
```

```
dfa_epsilon_transition_state_to_state(
  states[46], source_scope_root
);

dfa_set_leaf(states[49], ELLIPSIS);

dfa_epsilon_transition_state_to_state(
  states[49], source_scope_root
);

epsilon = dfa_add_state_epsilon(
  dfa, states[6], PREFIX "state 6"
);

dfa set leaf(epsilon, SUB OP);

dfa_epsilon_transition_state_to_state(
  epsilon, source_scope_root
);

epsilon = dfa_add_state_epsilon(
  dfa, states[10], PREFIX "state_10"
);

dfa_set_leaf(epsilon, ADD_OP);

dfa_epsilon_transition_state_to_state(
  epsilon, source_scope_root
);

epsilon = dfa_add_state_epsilon(
  dfa, states[13], PREFIX "state_13"
);

dfa_set_leaf(epsilon, AMPERSAND);

dfa_epsilon_transition_state_to_state(
  epsilon, source_scope_root
);

epsilon = dfa_add_state_epsilon(
  dfa, states[16], PREFIX "state_16"
);
```

```
dfa_set_leaf(epsilon, ASTERISK);

dfa_epsilon_transition_state_to_state(
  epsilon, source_scope_root
);

epsilon = dfa_add_state_epsilon(
  dfa, states[19], PREFIX "state_19"
);

dfa_set_leaf(epsilon, LOGICAL_NOT_OP);

dfa_epsilon_transition_state_to_state(
  epsilon, source_scope_root
);

epsilon = dfa_add_state_epsilon(
  dfa, states[26], PREFIX "state_26"
);

dfa_set_leaf(epsilon, DIV_OP);

dfa_epsilon_transition_state_to_state(
  epsilon, source_scope_root
);

epsilon = dfa_add_state_epsilon(
  dfa, states[29], PREFIX "state_29"
);

dfa_set_leaf(epsilon, MOD_OP);

dfa_epsilon_transition_state_to_state(
  epsilon, source_scope_root
);

epsilon = dfa_add_state_epsilon(
  dfa, states[31], PREFIX "state_31"
);

dfa_set_leaf(epsilon, L_OP);

dfa_epsilon_transition_state_to_state(
```

```
    epsilon, source_scope_root
  );

  epsilon = dfa_add_state_epsilon(
    dfa, states[32], PREFIX "state_32"
  );

  dfa_set_leaf(epsilon, LEFT_OP);

  dfa_epsilon_transition_state_to_state(
    epsilon, source_scope_root
  );

  epsilon = dfa_add_state_epsilon(
    dfa, states[35], PREFIX "state_35"
  );

  dfa_set_leaf(epsilon, G_OP);

  dfa_epsilon_transition_state_to_state(
    epsilon, source_scope_root
  );

  epsilon = dfa_add_state_epsilon(
    dfa, states[36], PREFIX "state_36"
  );

  dfa_set_leaf(epsilon, RIGHT_OP);

  dfa_epsilon_transition_state_to_state(
    epsilon, source_scope_root
  );

  epsilon = dfa_add_state_epsilon(
    dfa, states[39], PREFIX "state_39"
  );

  dfa_set_leaf(epsilon, EQUAL);

  dfa_epsilon_transition_state_to_state(
    epsilon, source_scope_root
  );

  epsilon = dfa_add_state_epsilon(
```

```
    dfa, states[41], PREFIX "state_41"
  );

  dfa_set_leaf(epsilon, BITWISE_XOR_OP);

  dfa_epsilon_transition_state_to_state(
    epsilon, source_scope_root
  );

  epsilon = dfa_add_state_epsilon(
    dfa, states[43], PREFIX "state_43"
  );

  dfa_set_leaf(epsilon, BITWISE_OR_OP);

  dfa epsilon transition state to state(
    epsilon, source_scope_root
  );

  epsilon = dfa_add_state_epsilon(
    dfa, states[47], PREFIX "state_47"
  );

  dfa_set_leaf(epsilon, DOT_OP);

  dfa_epsilon_transition_state_to_state(
    epsilon, source_scope_root
  );

  *state_dot = states[47];

  /* Clean-up */

  free(label);
  label = NULL;

  free(states);
  states = NULL;
}
```

8.2.9 mod_lexer_dfa_c90_type_identifier

8.2.9.1 mod_lexer_dfa_c90_type_identifier.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_c90_type_identifier_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  GraphNode *node_type
);
```

8.2.9.2 mod_lexer_dfa_c90_type_identifier.c

```
#include <string.h>
#include <stdlib.h>

#include "lib_common.h"
#include "mod_lexer_dfa_c90_type_identifier.h"
#include "mod_parser.tab.h"

#define PREFIX "type_identifier_"

#define STATE_2_TRANSITION '_'

#define STATE_3_TRANSITIONS "_ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij" \
  "klmnopqrstuvwxyz"

#define STATE_4_TRANSITIONS "0123456789"

/*
 * lexer_dfa_c90_type_identifier_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, source_scope_root
```

```
 * [param] IOP, node_type
 */
void lexer_dfa_c90_type_identifier_create(
  const DFA *dfa,
  DFAState *source_scope_root,
  DFAState *node_type
) {
  DFAState *epsilon;

  DFAState *state_2 = dfa_add_state(dfa, PREFIX "state_2");

  DFAStateGroup *state_3 = dfa_add_state_group(
    dfa, strlen(STATE 3 TRANSITIONS), PREFIX "state 3"
  );

  DFAStateGroup *state 4 = dfa add state group(
    dfa, strlen(STATE_4_TRANSITIONS), PREFIX "state_4"
  );

  int *state_3_transitions = dfa_string_to_transitions(
    STATE_3_TRANSITIONS
  );

  int *state_4_transitions = dfa_string_to_transitions(
    STATE_4_TRANSITIONS
  );

  /*** TRANSITION FROM SOURCE SCOPE ROOT ***/

  dfa_transition_state_to_state(
    node_type, state_2, STATE_2_TRANSITION
  );

  epsilon = dfa_add_state_epsilon(
    dfa, node_type, PREFIX "state_type"
  );

  dfa_set_leaf(epsilon, IDENTIFIER);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  /*** TRANSITION FROM STATE 2 ***/

  dfa_transition_state_to_state_group(
    state_2,
```

```
    state_3,
    strlen(STATE_3_TRANSITIONS),
    state_3_transitions
  );

  dfa_transition_state_to_state_group(
    state_2,
    state_4,
    strlen(STATE_4_TRANSITIONS),
    state_4_transitions
  );

  epsilon = dfa add state epsilon(
    dfa, state_2, PREFIX "state_2"
  );

  dfa_set_leaf(epsilon, TYPE_IDENTIFIER);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  /*** TRANSITION FROM STATE 3 ***/

  dfa_transition_state_group_to_state_group(
    state_3,
    state_3,
    strlen(STATE_3_TRANSITIONS),
    state_3_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_3,
    state_4,
    strlen(STATE_4_TRANSITIONS),
    state_4_transitions
  );

  epsilon = dfa_add_state_group_epsilon(
    dfa, state_3, PREFIX "state_3"
  );

  dfa_set_leaf(epsilon, TYPE_IDENTIFIER);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  /*** TRANSITION FROM STATE 4 ***/
```

```
  dfa_transition_state_group_to_state_group(
    state_4,
    state_3,
    strlen(STATE_3_TRANSITIONS),
    state_3_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_4,
    state_4,
    strlen(STATE_4_TRANSITIONS),
    state_4_transitions
  );

  epsilon = dfa_add_state_group_epsilon(
    dfa, state 4, PREFIX "state 4"
  );

  dfa_set_leaf(epsilon, TYPE_IDENTIFIER);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  /* Clean-up */

  free(state_3_transitions);
  state_3_transitions = NULL;

  free(state_4_transitions);
  state_4_transitions = NULL;

  list_free(state_3);
  list_free(state_4);
}
```

8.2.10 mod_lexer_dfa_v_doc

8.2.10.1 mod_lexer_dfa_v_doc.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_v_doc_create(
  const DFA *dfa,
  GraphNode *comment_scope_root,
  GraphNode *doc_scope_root,
  GraphNode *doc_rule_scope_root
);
```

8.2.10.2 mod_lexer_dfa_v_doc.c

```
#include <stdlib.h>
#include <assert.h>

#include "lib_common.h"
#include "mod_lexer_dfa_v_doc.h"
#include "mod_parser.tab.h"

#define PREFIX "v_doc_"
#define STATE_TRANSITIONS "  [param]description]recursive]"

/*
 * lexer_dfa_v_doc_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, comment_scope_root
 * [param] IOP, doc_scope_root
 * [param] IOP, doc_rule_scope_root
 */
void lexer_dfa_v_doc_create(
  const DFA *dfa,
```

```
  DFAState *comment_scope_root,
  DFAState *doc_scope_root,
  DFAState *doc_rule_scope_root
) {
  DFAState **states;

  unsigned int i;

  char *label;

  states = malloc(sizeof(DFAState*) * 31);
  assert(states != NULL);

  label = malloc(sizeof(char) * 16);
  assert(label != NULL);

  for (i = 2; i < 31; i += 1) {
    sprintf(label, PREFIX "state_%d", i);
    states[i] = dfa_add_state(dfa, label);
  }

  dfa_transition_state_to_state(
    doc_scope_root, states[2], STATE_TRANSITIONS[2]
  );
  dfa_transition_state_to_state(
    states[2], states[3], STATE_TRANSITIONS[3]
  );

  dfa_transition_state_to_state(
    states[2], states[9], STATE_TRANSITIONS[9]
  );

  dfa_transition_state_to_state(
    states[2], states[21], STATE_TRANSITIONS[21]
  );
  dfa_transition_state_to_state(
    states[3], states[4], STATE_TRANSITIONS[4]
  );

  dfa_transition_state_to_state(
    states[4], states[5], STATE_TRANSITIONS[5]
  );

  dfa_transition_state_to_state(
```

```
    states[5], states[6], STATE_TRANSITIONS[6]
  );

  dfa_transition_state_to_state(
    states[6], states[7], STATE_TRANSITIONS[7]
  );

  dfa_transition_state_to_state(
    states[7], states[8], STATE_TRANSITIONS[8]
  );
  dfa_transition_state_to_state(
    states[9], states[10], STATE_TRANSITIONS[10]
  );

  dfa_transition_state_to_state(
    states[10], states[11], STATE TRANSITIONS[11]
  );

  dfa_transition_state_to_state(
    states[11], states[12], STATE_TRANSITIONS[12]
  );

  dfa_transition_state_to_state(
    states[12], states[13], STATE_TRANSITIONS[13]
  );

  dfa_transition_state_to_state(
    states[13], states[14], STATE_TRANSITIONS[14]
  );

  dfa_transition_state_to_state(
    states[14], states[15], STATE_TRANSITIONS[15]
  );

  dfa_transition_state_to_state(
    states[15], states[16], STATE_TRANSITIONS[16]
  );

  dfa_transition_state_to_state(
    states[16], states[17], STATE_TRANSITIONS[17]
  );

  dfa_transition_state_to_state(
    states[17], states[18], STATE_TRANSITIONS[18]
```

```
);

dfa_transition_state_to_state(
  states[18], states[19], STATE_TRANSITIONS[19]
);

dfa_transition_state_to_state(
  states[19], states[20], STATE_TRANSITIONS[20]
);
dfa_transition_state_to_state(
  states[21], states[22], STATE_TRANSITIONS[22]
);

dfa_transition_state_to_state(
  states[22], states[23], STATE_TRANSITIONS[23]
);

dfa_transition_state_to_state(
  states[23], states[24], STATE_TRANSITIONS[24]
);

dfa_transition_state_to_state(
  states[24], states[25], STATE_TRANSITIONS[25]
);

dfa_transition_state_to_state(
  states[25], states[26], STATE_TRANSITIONS[26]
);

dfa_transition_state_to_state(
  states[26], states[27], STATE_TRANSITIONS[27]
);

dfa_transition_state_to_state(
  states[27], states[28], STATE_TRANSITIONS[28]
);

dfa_transition_state_to_state(
  states[28], states[29], STATE_TRANSITIONS[29]
);

dfa_transition_state_to_state(
  states[29], states[30], STATE_TRANSITIONS[30]
);
```

```
dfa_set_leaf(states[8], V_PARAM);

dfa_set_leaf(states[20], V_DESCRIPTION);

dfa_set_leaf(states[30], V_RECURSIVE);

dfa_epsilon_transition_state_to_state(
  states[8], doc_rule_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[20], doc rule scope root
);

dfa epsilon transition state to state(
  states[30], comment_scope_root
);
dfa_epsilon_transition_state_to_state(
  states[2], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[3], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[4], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[5], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[6], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[7], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[9], comment_scope_root
```

```
);

dfa_epsilon_transition_state_to_state(
  states[10], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[11], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[12], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[13], comment scope root
);

dfa_epsilon_transition_state_to_state(
  states[14], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[15], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[16], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[17], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[18], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[19], comment_scope_root
);

dfa_epsilon_transition_state_to_state(
  states[21], comment_scope_root
```

```
  );

  dfa_epsilon_transition_state_to_state(
    states[22], comment_scope_root
  );

  dfa_epsilon_transition_state_to_state(
    states[23], comment_scope_root
  );

  dfa_epsilon_transition_state_to_state(
    states[24], comment_scope_root
  );

  dfa_epsilon_transition_state_to_state(
    states[25], comment scope root
  );

  dfa_epsilon_transition_state_to_state(
    states[26], comment_scope_root
  );

  dfa_epsilon_transition_state_to_state(
    states[27], comment_scope_root
  );

  dfa_epsilon_transition_state_to_state(
    states[28], comment_scope_root
  );

  dfa_epsilon_transition_state_to_state(
    states[29], comment_scope_root
  );

  /* Clean-up */

  free(label);
  label = NULL;

  free(states);
  states = NULL;
}
```

8.2.11 mod_lexer_dfa_v_doc_rule

8.2.11.1 mod_lexer_dfa_v_doc_rule.h

```
#include "lib_dfa.h"
#include "lib_graph.h"

void lexer_dfa_v_doc_rule_create(
  const DFA *dfa,
  GraphNode *source_scope_root,
  GraphNode *comment_scope_root,
  GraphNode *doc_rule_scope_root
);
```

8.2.11.2 mod_lexer_dfa_v_doc_rule.c

```
#include <string.h>
#include <stdlib.h>

#include "lib_common.h"
#include "mod_lexer_dfa_v_doc_rule.h"
#include "mod_parser.tab.h"

#define PREFIX "v_doc_rule_"

#define STATE_2_TRANSITION 'I'
#define STATE_4_TRANSITION 'O'
#define STATE_6_TRANSITION 'O'
#define STATE_8_TRANSITION ','
#define STATE_9_TRANSITION '+'
#define STATE_10_TRANSITION '-'
#define STATE_12_TRANSITION '*'
#define STATE_13_TRANSITION '/'
#define STATE_14_TRANSITION '\n'
#define STATE_18_TRANSITION '*'
#define STATE_19_TRANSITION '/'

#define STATE_3_TRANSITIONS "PFK"
```

```
#define STATE_5_TRANSITIONS "PF"

#define STATE_7_TRANSITIONS "PFVR"

#define STATE_11_TRANSITIONS "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij" \
  "klmnopqrstuvwxyz0123456789!\"#%&'()+,-./:;<=>?[\\]^_{|}~ \t\v\f"

#define STATE_15_TRANSITIONS "ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkl" \
  "mnopqrstuvwxyz_"

#define STATE_16_TRANSITIONS "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghij" \
  "klmnopqrstuvwxyz "

#define STATE_17_TRANSITIONS "0123456789"

#define STATE_20_TRANSITIONS "\n\r"

#define STATE_21_TRANSITIONS " \t"

/*
 * lexer_dfa_v_doc_rule_create
 *
 * Integrates the DFA of the Lexer.
 *
 * [param] IOP, dfa
 * [param] IOP, source_scope_root
 * [param] IOP, comment_scope_root
 * [param] IOP, doc_rule_scope_root
 */
void lexer_dfa_v_doc_rule_create(
  const DFA *dfa,
  DFAState *source_scope_root,
  DFAState *comment_scope_root,
  DFAState *doc_rule_scope_root
) {
  unsigned int i;

  ListNode *state_iterator;

  DFAState *epsilon;

  DFAState *state_2 = dfa_add_state(dfa, PREFIX "state_2");
  DFAState *state_4 = dfa_add_state(dfa, PREFIX "state_4");
```

```
  DFAState *state_6 = dfa_add_state(dfa, PREFIX "state_6");
  DFAState *state_8 = dfa_add_state(dfa, PREFIX "state_8");
  DFAState *state_9 = dfa_add_state(dfa, PREFIX "state_9");
  DFAState *state_10 = dfa_add_state(dfa, PREFIX "state_10");
  DFAState *state_12 = dfa_add_state(dfa, PREFIX "state_12");
  DFAState *state_13 = dfa_add_state(dfa, PREFIX "state_13");
  DFAState *state_14 = dfa_add_state(dfa, PREFIX "state_14");
  DFAState *state_18 = dfa_add_state(dfa, PREFIX "state_18");
  DFAState *state_19 = dfa_add_state(dfa, PREFIX "state_19");

  DFAStateGroup *state_3 = dfa_add_state_group(
    dfa, strlen(STATE_3_TRANSITIONS), PREFIX "state_3"
  );

  DFAStateGroup *state_5 = dfa_add_state_group(
    dfa, strlen(STATE 5 TRANSITIONS), PREFIX "state 5"
  );

  DFAStateGroup *state_7 = dfa_add_state_group(
    dfa, strlen(STATE_7_TRANSITIONS), PREFIX "state_7"
  );

  DFAStateGroup *state_11 = dfa_add_state_group(
    dfa, strlen(STATE_11_TRANSITIONS), PREFIX "state_11"
  );

  DFAStateGroup *state_15 = dfa_add_state_group(
    dfa, strlen(STATE_15_TRANSITIONS), PREFIX "state_15"
  );

  DFAStateGroup *state_16 = dfa_add_state_group(
    dfa, strlen(STATE_16_TRANSITIONS), PREFIX "state_16"
  );

  DFAStateGroup *state_17 = dfa_add_state_group(
    dfa, strlen(STATE_17_TRANSITIONS), PREFIX "state_17"
  );

  DFAStateGroup *state_20 = dfa_add_state_group(
    dfa, strlen(STATE_20_TRANSITIONS), PREFIX "state_20"
  );

  DFAStateGroup *state_21 = dfa_add_state_group(
    dfa, strlen(STATE_21_TRANSITIONS), PREFIX "state_21"
```

```
);

int *state_3_transitions = dfa_string_to_transitions(
  STATE_3_TRANSITIONS
);

int *state_5_transitions = dfa_string_to_transitions(
  STATE_5_TRANSITIONS
);

int *state_7_transitions = dfa_string_to_transitions(
  STATE_7_TRANSITIONS
);

int *state_11_transitions = dfa_string_to_transitions(
  STATE 11 TRANSITIONS
);

int *state_15_transitions = dfa_string_to_transitions(
  STATE_15_TRANSITIONS
);

int *state_16_transitions = dfa_string_to_transitions(
  STATE_16_TRANSITIONS
);

int *state_17_transitions = dfa_string_to_transitions(
  STATE_17_TRANSITIONS
);

int *state_20_transitions = dfa_string_to_transitions(
  STATE_20_TRANSITIONS
);

int *state_21_transitions = dfa_string_to_transitions(
  STATE_21_TRANSITIONS
);

/*** TRANSITIONS FORM DOC RULE SCOPE ROOT ***/

dfa_transition_state_to_state(
  doc_rule_scope_root, state_2, STATE_2_TRANSITION
);
```

```
dfa_transition_state_to_state(
  doc_rule_scope_root, state_6, STATE_6_TRANSITION
);

dfa_transition_state_to_state(
  doc_rule_scope_root, state_8, STATE_8_TRANSITION
);

dfa_transition_state_to_state(
  doc_rule_scope_root, state_9, STATE_9_TRANSITION
);

dfa transition state to state(
  doc_rule_scope_root, state_10, STATE_10_TRANSITION
);

dfa_transition_state_to_state_group(
  doc_rule_scope_root,
  state_15,
  strlen(STATE_15_TRANSITIONS),
  state_15_transitions
);

dfa_transition_state_to_state(
  doc_rule_scope_root, state_18, STATE_18_TRANSITION
);

dfa_transition_state_to_state_group(
  doc_rule_scope_root,
  state_20,
  strlen(STATE_20_TRANSITIONS),
  state_20_transitions
);

dfa_transition_state_to_state_group(
  doc_rule_scope_root,
  state_21,
  strlen(STATE_21_TRANSITIONS),
  state_21_transitions
);

/*** TRANSITIONS FROM STATE 2 ***/

dfa_transition_state_to_state_group(
```

```
    state_2,
    state_3,
    strlen(STATE_3_TRANSITIONS),
    state_3_transitions
  );

  dfa_transition_state_to_state(
    state_2, state_4, STATE_4_TRANSITION
  );

  /*** TRANSITIONS FROM STATE 3 ***/

  epsilon = dfa add state group epsilon(
    dfa, state_3, PREFIX "state_3"
  );

  dfa_set_leaf(epsilon, V_KEYWORD);

  dfa_epsilon_transition_state_to_state(
    epsilon, doc_rule_scope_root
  );

  /*** TRANSITIONS FROM STATE 4 ***/

  dfa_transition_state_to_state_group(
    state_4,
    state_5,
    strlen(STATE_5_TRANSITIONS),
    state_5_transitions
  );

  /*** TRANSITIONS FROM STATE 5 ***/

  epsilon = dfa_add_state_group_epsilon(
    dfa, state_5, PREFIX "state_5"
  );

  dfa_set_leaf(epsilon, V_KEYWORD);

  dfa_epsilon_transition_state_to_state(
    epsilon, doc_rule_scope_root
  );

  /*** TRANSITIONS FROM STATE 6 ***/
```

```
dfa_transition_state_to_state_group(
  state_6,
  state_7,
  strlen(STATE_7_TRANSITIONS),
  state_7_transitions
);

/*** TRANSITIONS FROM STATE 7 ***/

epsilon = dfa_add_state_group_epsilon(
  dfa, state_7, PREFIX "state_7"
);

dfa_set_leaf(epsilon, V_KEYWORD);

dfa_epsilon_transition_state_to_state(
  epsilon, doc_rule_scope_root
);

/*** TRANSITIONS FROM STATE 8 ***/

dfa_set_leaf(state_8, V_COMMA);

dfa_epsilon_transition_state_to_state(
  state_8, doc_rule_scope_root
);

/*** TRANSITIONS FROM STATE 9 ***/

dfa_set_leaf(state_9, V_PLUS);

dfa_epsilon_transition_state_to_state(
  state_9, doc_rule_scope_root
);

/*** TRANSITIONS FROM STATE 10 ***/

dfa_transition_state_to_state_group(
  state_10,
  state_11,
  strlen(STATE_11_TRANSITIONS),
  state_11_transitions
);
```

```
dfa_transition_state_to_state(
  state_10, state_12, STATE_12_TRANSITION
);

dfa_transition_state_to_state(
  state_10, state_14, STATE_14_TRANSITION
);

/*** TRANSITIONS FROM STATE 11 ***/

dfa_transition_state_group_to_state_group(
  state 11,
  state_11,
  strlen(STATE_11_TRANSITIONS),
  state 11 transitions
);

dfa_transition_state_group_to_state(
  state_11, state_12, STATE_12_TRANSITION
);

dfa_transition_state_group_to_state(
  state_11, state_14, STATE_14_TRANSITION
);

/*** TRANSITIONS FROM STATE 12 ***/

dfa_transition_state_to_state(
  state_12, state_12, STATE_12_TRANSITION
);

dfa_transition_state_to_state(
  state_12, state_13, STATE_13_TRANSITION
);

dfa_transition_state_to_state(
  state_12, state_14, STATE_14_TRANSITION
);

/*** TRANSITIONS FROM STATE 12 TO STATE 11 ***/

i = 0;
```

```
  for (
    state_iterator = list_get_iterator(state_11);
    state_iterator != NULL;
    state_iterator = list_iterate(state_iterator)
  ) {
    if (
      STATE_11_TRANSITIONS[i] != '*'
      && STATE_11_TRANSITIONS[i] != '\n'
      && STATE_11_TRANSITIONS[i] != '/') {
      dfa_transition_state_to_state(
        state_12, state_iterator->value, STATE_11_TRANSITIONS[i]
      );
    }

    i += 1;
  }

  /*** TRANSITIONS FROM STATE 13 ***/

  dfa_set_leaf(state_13, V_STRING);

  epsilon = dfa_add_state_epsilon(dfa, state_13, PREFIX "state_13");
  dfa_set_leaf(epsilon, CLOSE_COMMENT);
  dfa_epsilon_transition_state_to_state(epsilon, source_scope_root);

  /*** TRANSITIONS FROM STATE 14 ***/

  dfa_set_leaf(state_14, V_STRING);

  dfa_epsilon_transition_state_to_state(
    state_14, comment_scope_root
  );

  /*** TRANSITIONS FROM STATE 15 ***/

  dfa_transition_state_group_to_state_group(
    state_15,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_15,
```

```
    state_17,
    strlen(STATE_17_TRANSITIONS),
    state_17_transitions
  );

  epsilon = dfa_add_state_group_epsilon(
    dfa, state_15, PREFIX "state_15"
  );

  dfa_set_leaf(epsilon, V_IDENTIFIER);

  dfa_epsilon_transition_state_to_state(
    epsilon, doc rule scope root
  );

  /*** TRANSITIONS FROM STATE 16 ***/

  dfa_transition_state_group_to_state_group(
    state_16,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );

  dfa_transition_state_group_to_state_group(
    state_16,
    state_17,
    strlen(STATE_17_TRANSITIONS),
    state_17_transitions
  );

  epsilon = dfa_add_state_group_epsilon(
    dfa, state_16, PREFIX "state_16"
  );

  /*** TRANSITIONS TO STATE 16 EPSILON ***/

  dfa_transition_state_to_state_group(
    state_2,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );
```

```
  dfa_epsilon_transition_state_to_state(state_2, epsilon);

  dfa_transition_state_group_to_state_group(
    state_3,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );

  dfa_transition_state_to_state_group(
    state_4,
    state_16,
    strlen(STATE 16 TRANSITIONS),
    state_16_transitions
  );

  dfa_epsilon_transition_state_to_state(state_4, epsilon);

  dfa_transition_state_group_to_state_group(
    state_5,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );

  dfa_transition_state_to_state_group(
    state_6,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );

  dfa_epsilon_transition_state_to_state(state_6, epsilon);

  dfa_transition_state_group_to_state_group(
    state_7,
    state_16,
    strlen(STATE_16_TRANSITIONS),
    state_16_transitions
  );

  dfa_set_leaf(epsilon, V_IDENTIFIER);

  dfa_epsilon_transition_state_to_state(
```

```
  epsilon, doc_rule_scope_root
);

/*** TRANSITIONS FROM STATE 17 ***/

dfa_transition_state_group_to_state_group(
  state_17,
  state_16,
  strlen(STATE_16_TRANSITIONS),
  state_16_transitions
);

dfa transition state group to state group(
  state_17,
  state_17,
  strlen(STATE 17 TRANSITIONS),
  state_17_transitions
);

epsilon = dfa_add_state_group_epsilon(
  dfa, state_17, PREFIX "state_17"
);

dfa_set_leaf(epsilon, V_IDENTIFIER);

dfa_epsilon_transition_state_to_state(
  epsilon, doc_rule_scope_root
);

/*** TRANSITIONS FROM STATE 18 ***/

dfa_transition_state_to_state(
  state_18, state_19, STATE_19_TRANSITION
);

/*** TRANSITIONS FROM STATE 19 ***/

dfa_set_leaf(state_19, CLOSE_COMMENT);

dfa_epsilon_transition_state_to_state(
  state_19, source_scope_root
);

/*** TRANSITIONS FROM STATE 20 ***/
```

```
dfa_epsilon_transition_state_group_to_state(
  state_20, comment_scope_root
);

/*** TRANSITIONS FROM STATE 21 ***/

dfa_epsilon_transition_state_group_to_state(
  state_21, doc_rule_scope_root
);

/* Clean-up */

free(state_3_transitions);
state_3_transitions = NULL;

free(state_5_transitions);
state_5_transitions = NULL;

free(state_7_transitions);
state_7_transitions = NULL;

free(state_11_transitions);
state_11_transitions = NULL;

free(state_15_transitions);
state_15_transitions = NULL;

free(state_16_transitions);
state_16_transitions = NULL;

free(state_17_transitions);
state_17_transitions = NULL;

free(state_20_transitions);
state_20_transitions = NULL;

free(state_21_transitions);
state_21_transitions = NULL;

list_free(state_3);
list_free(state_5);
list_free(state_7);
list_free(state_11);
```

```
  list_free(state_15);
  list_free(state_16);
  list_free(state_17);
  list_free(state_20);
  list_free(state_21);
}
```

8.3 Parser

8.3.1.1 `mod_parser.h`

```
#include <stdio.h>

#include "lib_graph.h"
#include "mod_parser.tab.h"
#include "mod_lexer.h"

#define ABSTRACT_DECLARATOR_AST_NODE "ABSTRACT_DECLARATOR"
#define ADDITIVE_EXPRESSION_AST_NODE "ADDITIVE_EXPRESSION"
#define ARGUMENT_EXPRESSION_LIST_AST_NODE "ARGUMENT_EXPRESSION_LIST"
#define ARRAY_ACCESS_AST_NODE "ARRAY_ACCESS"
#define ASSIGNMENT_EXPRESSION_AST_NODE "ASSIGNMENT_EXPRESSION"
#define ASSIGN_OPERATOR_AST_NODE "ASSIGN_OPERATOR"
#define BITWISE_AND_EXPRESSION_AST_NODE "BITWISE_AND_EXPRESSION"
#define BITWISE_OR_EXPRESSION_AST_NODE "BITWISE_OR_EXPRESSION"
#define BITWISE_XOR_EXPRESSION_AST_NODE "BITWISE_XOR_EXPRESSION"
#define CAST_EXPRESSION_AST_NODE "CAST_EXPRESSION"
#define COMMENT_AST_NODE "COMMENT"
#define COMPOUND_STATEMENT_AST_NODE "COMPOUND_STATEMENT"
#define CONSTANT_AST_NODE "CONSTANT"
#define DECLARATION_AST_NODE "DECLARATION"
#define EXTERNAL_DECLARATION_AST_NODE "EXTERNAL_DECLARATION"
#define DECLARATION_LIST_AST_NODE "DECLARATION_LIST"
#define DECLARATION_SPECIFIERS_AST_NODE "DECLARATION_SPECIFIERS"
#define DECLARATOR_AST_NODE "DECLARATOR"
#define DECREMENT_AST_NODE "DECREMENT"
#define DIRECT_ABSTRACT_DECLARATOR_AST_NODE \
  "DIRECT_ABSTRACT_DECLARATOR"
#define DIRECT_DECLARATOR_AST_NODE "DIRECT_DECLARATOR"
#define ENUMERATOR_AST_NODE "ENUMERATOR"
#define ENUMERATOR_LIST_AST_NODE "ENUMERATOR_LIST"
#define ENUM_SPECIFIER_AST_NODE "ENUM_SPECIFIER"
#define EQUALITY_EXPRESSION_AST_NODE "EQUALITY_EXPRESSION"
#define EXPRESSION_AST_NODE "EXPRESSION"
#define EXPRESSION_STATEMENT_AST_NODE "EXPRESSION_STATEMENT"
```

```
#define FUNCTION_CALL_AST_NODE "FUNCTION_CALL"
#define FUNCTION_DEFINITION_AST_NODE "FUNCTION_DEFINITION"
#define IDENTIFIER_AST_NODE "IDENTIFIER"
#define IDENTIFIER_LIST_AST_NODE "IDENTIFIER_LIST"
#define INCREMENT_AST_NODE "INCREMENT"
#define INITIALIZER_AST_NODE "INITIALIZER"
#define INITIALIZER_LIST_AST_NODE "INITIALIZER_LIST"
#define INIT_DECLARATOR_AST_NODE "INIT_DECLARATOR"
#define INIT_DECLARATOR_LIST_AST_NODE "INIT_DECLARATOR_LIST"
#define ITERATION_STATEMENT_AST_NODE "ITERATION_STATEMENT"
#define JUMP_STATEMENT_GOTO_AST_NODE "JUMP_STATEMENT_GOTO"
#define JUMP_STATEMENT_CONTINUE_AST_NODE "JUMP_STATEMENT_CONTINUE"
#define JUMP STATEMENT BREAK AST NODE "JUMP STATEMENT BREAK"
#define JUMP_STATEMENT_RETURN_AST_NODE "JUMP_STATEMENT_RETURN"
#define JUMP_STATEMENT_RETURN_AST_NODE "JUMP_STATEMENT_RETURN"
#define LABELED STATEMENT AST NODE "LABELED STATEMENT"
#define LOGICAL_AND_EXPRESSION_AST_NODE "LOGICAL_AND_EXPRESSION"
#define LOGICAL_OR_EXPRESSION_AST_NODE "LOGICAL_OR_EXPRESSION"
#define MEMBER_ACCESS_AST_NODE "MEMBER_ACCESS"
#define MULTIPLICATIVE_EXPRESSION_AST_NODE \
  "MULTIPLICATIVE_EXPRESSION"
#define PARAMETER_DECLARATION_AST_NODE "PARAMETER_DECLARATION"
#define PARAMETER_LIST_AST_NODE "PARAMETER_LIST"
#define PARAMETER_TYPE_LIST_AST_NODE "PARAMETER_TYPE_LIST"
#define POINTER_AST_NODE "POINTER"
#define POINTER_ACCESS_AST_NODE "POINTER_ACCESS"
#define PROGRAM_AST_NODE "PROGRAM"
#define RELATIONAL_EXPRESSION_AST_NODE "RELATIONAL_EXPRESSION"
#define SELECTION_STATEMENT_AST_NODE "SELECTION_STATEMENT"
#define SHIFT_EXPRESSION_AST_NODE "SHIFT_EXPRESSION"
#define SPECIFIER_QUALIFIER_LIST_AST_NODE "SPECIFIER_QUALIFIER_LIST"
#define STATEMENT_AST_NODE "STATEMENT"
#define STATEMENT_LIST_AST_NODE "STATEMENT_LIST"
#define STORAGE_CLASS_SPECIFIER_AST_NODE "STORAGE_CLASS_SPECIFIER"
#define STRING_LITERAL_AST_NODE "STRING_LITERAL"
#define STRUCT_AST_NODE "STRUCT"
#define STRUCT_DECLARATION_AST_NODE "STRUCT_DECLARATION"
#define STRUCT_DECLARATION_LIST_AST_NODE "STRUCT_DECLARATION_LIST"
#define STRUCT_DECLARATOR_AST_NODE "STRUCT_DECLARATOR"
#define STRUCT_DECLARATOR_LIST_AST_NODE "STRUCT_DECLARATOR_LIST"
#define STRUCT_OR_UNION_SPECIFIER_AST_NODE \
  "STRUCT_OR_UNION_SPECIFIER"
#define TERNARY_EXPRESSION_AST_NODE "TERNARY_EXPRESSION"
#define TRANSLATION_UNIT_AST_NODE "TRANSLATION_UNIT"
```

```
#define TYPE_IDENTIFIER_AST_NODE "TYPE_IDENTIFIER"
#define TYPE_NAME_AST_NODE "TYPE_NAME"
#define TYPE_QUALIFIER_AST_NODE "TYPE_QUALIFIER"
#define TYPE_QUALIFIER_LIST_AST_NODE "TYPE_QUALIFIER_LIST"
#define TYPE_SPECIFIER_AST_NODE "TYPE_SPECIFIER"
#define UNARY_EXPRESSION_AST_NODE "UNARY_EXPRESSION"
#define UNARY_OPERATOR_AST_NODE "UNARY_OPERATOR"
#define UNION_AST_NODE "UNION"
#define V_IDENTIFIER_AST_NODE "V_IDENTIFIER"
#define V_KEYWORD_AST_NODE "V_KEYWORD"
#define V_KEYWORD_LIST_AST_NODE "V_KEYWORD_LIST"
#define V_RULE_AST_NODE "V_RULE"
#define V RULE DESCRIPTION AST NODE "V RULE DESCRIPTION"
#define V_RULE_LIST_AST_NODE "V_RULE_LIST"
#define V_RULE_PARAM_AST_NODE "V_RULE_PARAM"
#define V RULE RECURSIVE AST NODE "V RULE RECURSIVE"
#define V_STRING_AST_NODE "V_STRING"

Graph* parser_parse_file(Lexer *custom_lexer);
void ast_free(Graph *ast);
```

8.3.1.2 `mod_parser.y`

```
%{
  #include <assert.h>
  #include <stdio.h>
  #include <stdlib.h>
  #include <string.h>

  #include "lib_buffer.h"
  #include "lib_graph.h"
  #include "mod lexer.h"
  #include "mod_parser.h"

  extern void yyerror(char *s);
  extern int yylex(void);

  Graph *ast;
  Lexer *lexer;
  DFAMatch *match = NULL;
  Buffer *identifier;
```

```
  char *current_ast_node = PROGRAM_AST_NODE;
  char *value;
  char *type;
  int counter = 0;

  /**
   * parser_create_ast_node
   *
   * [param] IP, type
   * [param] IP, value
   */
  GraphNode* parser_create_ast_node(char *type, const char *value) {
    GraphNode *result;

    char *value_copy, *type_copy;

    counter += 1;
    current_ast_node = type;

    buffer_reset(identifier);
    buffer_append_string(identifier, current_ast_node);
    buffer_append_integer(identifier, counter);

    value_copy = malloc(sizeof(char) * (strlen(value) + 1));
    assert(value_copy != NULL);
    strcpy(value_copy, value);

    type_copy =
      malloc(sizeof(char) * (strlen(current_ast_node) + 1));
    assert(type_copy != NULL);
    strcpy(type_copy, current_ast_node);

    result = graph_add_node(ast, identifier->value);
    graph_node_set_field(result, "value", value_copy);
    graph_node_set_field(result, "type", type_copy);

    return result;
  }
%}

%locations
%error-verbose

%union {
```

```
  char *string_v;
  struct GraphNode *child_node;
}

%type <string_v> IDENTIFIER
%type <string_v> STRING_LITERAL
%type <string_v> TYPE_IDENTIFIER
%type <string_v> CONSTANT
%type <string_v> V_KEYWORD
%type <string_v> V_STRING
%type <string_v> V_IDENTIFIER

%type <child node> identifier
%type <child_node> constant
%type <child_node> string_literal
%type <child node> type identifier
%type <child_node> primary_expression
%type <child_node> postfix_expression
%type <child_node> argument_expression_list
%type <child_node> unary_expression
%type <child_node> unary_operator
%type <child_node> cast_expression
%type <child_node> multiplicative_expression
%type <child_node> additive_expression
%type <child_node> shift_expression
%type <child_node> relational_expression
%type <child_node> equality_expression
%type <child_node> and_expression
%type <child_node> exclusive_or_expression
%type <child_node> inclusive_or_expression
%type <child_node> logical_and_expression
%type <child_node> logical_or_expression
%type <child_node> conditional_expression
%type <child_node> assignment_expression
%type <child_node> assignment_operator
%type <child_node> expression
%type <child_node> constant_expression
%type <child_node> declaration
%type <child_node> declaration_specifiers
%type <child_node> init_declarator_list
%type <child_node> init_declarator
%type <child_node> storage_class_specifier
%type <child_node> type_specifier
%type <child_node> struct_or_union_specifier
```

```
%type <child_node> struct_or_union
%type <child_node> struct_declaration_list
%type <child_node> struct_declaration
%type <child_node> specifier_qualifier_list
%type <child_node> struct_declarator_list
%type <child_node> struct_declarator
%type <child_node> enum_specifier
%type <child_node> enumerator_list
%type <child_node> enumerator
%type <child_node> type_qualifier
%type <child_node> declarator
%type <child_node> direct_declarator
%type <child node> pointer
%type <child_node> type_qualifier_list
%type <child_node> parameter_type_list
%type <child node> parameter list
%type <child_node> parameter_declaration
%type <child_node> identifier_list
%type <child_node> type_name
%type <child_node> abstract_declarator
%type <child_node> direct_abstract_declarator
%type <child_node> initializer
%type <child_node> initializer_list
%type <child_node> statement
%type <child_node> labeled_statement
%type <child_node> compound_statement
%type <child_node> declaration_list
%type <child_node> statement_list
%type <child_node> expression_statement
%type <child_node> selection_statement
%type <child_node> iteration_statement
%type <child_node> jump_statement
%type <child_node> translation_unit
%type <child_node> external_declaration
%type <child_node> function_definition
%type <child_node> comment
%type <child_node> program
%type <child_node> v_identifier
%type <child_node> v_keyword
%type <child_node> v_keyword_list
%type <child_node> v_rule
%type <child_node> v_rule_description
%type <child_node> v_rule_list
%type <child_node> v_rule_param
```

```
%type <child_node> v_rule_recursive
%type <child_node> v_string

%token AUTO
%token BREAK
%token CASE
%token CHAR
%token CONST
%token CONTINUE
%token DEFAULT
%token DO
%token DOUBLE
%token ELSE
%token ENUM
%token EXTERN
%token FLOAT
%token FOR
%token GOTO
%token IF
%token INT
%token LONG
%token REGISTER
%token RETURN
%token SHORT
%token SIGNED
%token SIZEOF
%token STATIC
%token STRUCT
%token SWITCH
%token TYPEDEF
%token UNION
%token UNSIGNED
%token VOID
%token VOLATILE
%token WHILE
%token ELLIPSIS
%token RIGHT_ASSIGN
%token LEFT_ASSIGN
%token ADD_ASSIGN
%token SUB_ASSIGN
%token MUL_ASSIGN
%token DIV_ASSIGN
%token MOD_ASSIGN
%token AND_ASSIGN
```

```
%token XOR_ASSIGN
%token OR_ASSIGN
%token RIGHT_OP
%token LEFT_OP
%token INC_OP
%token DEC_OP
%token PTR_OP
%token AND_OP
%token OR_OP
%token LE_OP
%token GE_OP
%token EQ_OP
%token NE OP
%token DOT_OP
%token LOGICAL_NOT_OP
%token BITWISE NOT OP
%token SUB_OP
%token ADD_OP
%token DIV_OP
%token MOD_OP
%token L_OP
%token G_OP
%token BITWISE_XOR_OP
%token BITWISE_OR_OP
%token TERNARY_OP
%token SEMICOLON
%token CO_BRACKET
%token CC_BRACKET
%token COMMA
%token COLON
%token EQUAL
%token RO_BRACKET
%token RC_BRACKET
%token SO_BRACKET
%token SC_BRACKET
%token AMPERSAND
%token ASTERISK
%token IDENTIFIER
%token CONSTANT
%token STRING_LITERAL
%token TYPE_IDENTIFIER
%token OPEN_COMMENT
%token CLOSE_COMMENT
%token V_COMMA
```

```
%token V_DESCRIPTION
%token V_IDENTIFIER
%token V_KEYWORD
%token V_PARAM
%token V_PLUS
%token V_RECURSIVE
%token V_STRING

%start program

%nonassoc THEN
%nonassoc ELSE

%%

identifier
  : IDENTIFIER {
    $$ = parser_create_ast_node(IDENTIFIER_AST_NODE, $1);
  }
  ;

constant
  : CONSTANT {
    $$ = parser_create_ast_node(CONSTANT_AST_NODE, $1);
  }
  ;

string_literal
  : STRING_LITERAL {
    $$ = parser_create_ast_node(STRING_LITERAL_AST_NODE, $1);
  }
  ;

type_identifier
  : TYPE_IDENTIFIER {
    $$ = parser_create_ast_node(TYPE_IDENTIFIER_AST_NODE, $1);
  }
  ;

primary_expression
  : identifier { $$ = $1; }
  | constant { $$ = $1; }
  | string_literal { $$ = $1; }
  | RO_BRACKET expression RC_BRACKET { $$ = $2; }
```

```
  ;

postfix_expression
  : primary_expression { $$ = $1; }
  | postfix_expression SO_BRACKET expression SC_BRACKET {
    $$ = parser_create_ast_node(ARRAY_ACCESS_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | postfix_expression RO_BRACKET RC_BRACKET {
    $$ = parser_create_ast_node(FUNCTION_CALL_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | postfix_expression RO_BRACKET argument_expression_list
    RC_BRACKET {
    $$ = parser create ast node(FUNCTION CALL AST NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | postfix_expression DOT_OP identifier {
    $$ = parser_create_ast_node(MEMBER_ACCESS_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | postfix_expression PTR_OP identifier {
    $$ = parser_create_ast_node(POINTER_ACCESS_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | postfix_expression INC_OP {
    $$ = parser_create_ast_node(INCREMENT_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | postfix_expression DEC_OP {
    $$ = parser_create_ast_node(DECREMENT_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  ;

argument_expression_list
  : assignment_expression {
    $$ = parser_create_ast_node(
      ARGUMENT_EXPRESSION_LIST_AST_NODE,
      ""
```

```
    );

    graph_add_edge($$, $1);
  }
  | argument_expression_list COMMA assignment_expression {
    $$ = parser_create_ast_node(
      ARGUMENT_EXPRESSION_LIST_AST_NODE,
      ""
    );

    graph_copy_edges($$, $1);
    graph_add_edge($$, $3);
  }
  ;

unary expression
  : postfix_expression {
    $$ = parser_create_ast_node(UNARY_EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | INC_OP unary_expression {
    $$ = parser_create_ast_node(
      UNARY_EXPRESSION_AST_NODE,
      "INCREMENT"
    );

    graph_add_edge($$, $2);
  }
  | DEC_OP unary_expression {
    $$ = parser_create_ast_node(
      UNARY_EXPRESSION_AST_NODE,
      "DECREMENT"
    );

    graph_add_edge($$, $2);
  }
  | unary_operator cast_expression {
    $$ = parser_create_ast_node(UNARY_EXPRESSION_AST_NODE, "");

    /* These are reserse ordered because unary expressions have */
    /* right-to-left associativity.                             */

    graph_add_edge($$, $2);
    graph_add_edge($$, $1);
```

```
  }
  | SIZEOF unary_expression {
    $$ = parser_create_ast_node(UNARY_EXPRESSION_AST_NODE, "SIZEOF");
    graph_add_edge($$, $2);
  }
  | SIZEOF RO_BRACKET type_name RC_BRACKET {
    $$ = parser_create_ast_node(UNARY_EXPRESSION_AST_NODE, "SIZEOF");
    graph_add_edge($$, $3);
  }
  ;

unary_operator
  : AMPERSAND {
    $$ = parser_create_ast_node(
      UNARY_OPERATOR_AST_NODE, "ADDRESS_OF"
    );
  }
  | ASTERISK {
    $$ = parser_create_ast_node(
      UNARY_OPERATOR_AST_NODE, "DEREFERENCE"
    );
  }
  | ADD_OP {
    $$ = parser_create_ast_node(
      UNARY_OPERATOR_AST_NODE, "PLUS"
    );
  }
  | SUB_OP {
    $$ = parser_create_ast_node(
      UNARY_OPERATOR_AST_NODE, "MINUS"
    );
  }
  | BITWISE_NOT_OP {
    $$ = parser_create_ast_node(
      UNARY_OPERATOR_AST_NODE, "BITWISE_NOT"
    );
  }
  | LOGICAL_NOT_OP {
    $$ = parser_create_ast_node(
      UNARY_OPERATOR_AST_NODE, "LOGICAL_NOT"
    );
  }
  ;
```

```
cast_expression
  : unary_expression { $$ = $1; }
  | RO_BRACKET type_name RC_BRACKET cast_expression {
    $$ = parser_create_ast_node(CAST_EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $2);
    graph_add_edge($$, $4);
  }
  ;

multiplicative_expression
  : cast_expression { $$ = $1; }
  | multiplicative_expression ASTERISK cast_expression {
    $$ = parser create ast node(
      MULTIPLICATIVE_EXPRESSION_AST_NODE, "MULTIPLY"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | multiplicative_expression DIV_OP cast_expression {
    $$ = parser_create_ast_node(
      MULTIPLICATIVE_EXPRESSION_AST_NODE, "DIVIDE"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | multiplicative_expression MOD_OP cast_expression {
    $$ = parser_create_ast_node(
      MULTIPLICATIVE_EXPRESSION_AST_NODE, "MODULO"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

additive_expression
  : multiplicative_expression { $$ = $1; }
  | additive_expression ADD_OP multiplicative_expression {
    $$ = parser_create_ast_node(
      ADDITIVE_EXPRESSION_AST_NODE, "ADD"
    );
```

```
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | additive_expression SUB_OP multiplicative_expression {
    $$ = parser_create_ast_node(
      ADDITIVE_EXPRESSION_AST_NODE, "SUBTRACT"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

shift_expression
  : additive_expression { $$ = $1; }
  | shift expression LEFT OP additive expression {
    $$ = parser_create_ast_node(
      SHIFT_EXPRESSION_AST_NODE, "LEFT_SHIFT"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | shift_expression RIGHT_OP additive_expression {
    $$ = parser_create_ast_node(
      SHIFT_EXPRESSION_AST_NODE, "RIGHT_SHIFT"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

relational_expression
  : shift_expression { $$ = $1; }
  | relational_expression L_OP shift_expression {
    $$ = parser_create_ast_node(
      RELATIONAL_EXPRESSION_AST_NODE, "LESS_THAN"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | relational_expression G_OP shift_expression {
```

```
    $$ = parser_create_ast_node(
      RELATIONAL_EXPRESSION_AST_NODE, "GREATER_THAN"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | relational_expression LE_OP shift_expression {
    $$ = parser_create_ast_node(
      RELATIONAL_EXPRESSION_AST_NODE, "LESS_THAN_EQUAL"
    );

    graph add edge($$, $1);
    graph_add_edge($$, $3);
  }
  | relational expression GE OP shift expression {
    $$ = parser_create_ast_node(
      RELATIONAL_EXPRESSION_AST_NODE, "GREATER_THAN_EQUAL"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

equality_expression
  : relational_expression { $$ = $1; }
  | equality_expression EQ_OP relational_expression {
    $$ = parser_create_ast_node(
      EQUALITY_EXPRESSION_AST_NODE, "EQUAL"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | equality_expression NE_OP relational_expression {
    $$ = parser_create_ast_node(
      EQUALITY_EXPRESSION_AST_NODE, "NOT_EQUAL"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;
```

```
and_expression
  : equality_expression { $$ = $1; }
  | and_expression AMPERSAND equality_expression {
    $$ = parser_create_ast_node(BITWISE_AND_EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

exclusive_or_expression
  : and_expression { $$ = $1; }
  | exclusive or expression BITWISE XOR OP and expression {
    $$ = parser_create_ast_node(BITWISE_XOR_EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph add edge($$, $3);
  }
  ;

inclusive_or_expression
  : exclusive_or_expression { $$ = $1; }
  | inclusive_or_expression BITWISE_OR_OP exclusive_or_expression {
    $$ = parser_create_ast_node(BITWISE_OR_EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

logical_and_expression
  : inclusive_or_expression { $$ = $1; }
  | logical_and_expression AND_OP inclusive_or_expression {
    $$ = parser_create_ast_node(LOGICAL_AND_EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

logical_or_expression
  : logical_and_expression { $$ = $1; }
  | logical_or_expression OR_OP logical_and_expression {
    $$ = parser_create_ast_node(LOGICAL_OR_EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
```

```
  ;

conditional_expression
  : logical_or_expression { $$ = $1; }
  | logical_or_expression TERNARY_OP expression COLON
    conditional_expression {
    $$ = parser_create_ast_node(TERNARY_EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
    graph_add_edge($$, $5);
  }
  ;

assignment_expression
  : conditional_expression { $$ = $1; }
  | unary expression assignment operator assignment expression {
    $$ = parser_create_ast_node(ASSIGNMENT_EXPRESSION_AST_NODE, "");


    /* These are reserse ordered because assignment expressions */
    /* have right-to-left associativity.                        */

    graph_add_edge($$, $3);
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  ;

assignment_operator
  : EQUAL {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "SIMPLE_ASSIGN"
    );
  }
  | MUL_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "MUL_ASSIGN"
    );
  }
  | DIV_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "DIV_ASSIGN"
    );
  }
```

```
  | MOD_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "MOD_ASSIGN"
    );
  }
  | ADD_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "ADD_ASSIGN"
    );
  }
  | SUB_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN OPERATOR AST NODE, "SUB ASSIGN"
    );
  }
  | LEFT ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "LEFT_ASSIGN"
    );
  }
  | RIGHT_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "RIGHT_ASSIGN"
    );
  }
  | AND_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "AND_ASSIGN"
    );
  }
  | XOR_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "XOR_ASSIGN"
    );
  }
  | OR_ASSIGN {
    $$ = parser_create_ast_node(
      ASSIGN_OPERATOR_AST_NODE, "OR_ASSIGN"
    );
  }
  ;

expression
  : assignment_expression {
```

```
    $$ = parser_create_ast_node(EXPRESSION_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | expression COMMA assignment_expression {
    $$ = parser_create_ast_node(EXPRESSION_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $3);
  }
  ;

constant_expression
  : conditional_expression { $$ = $1; }
  ;

declaration
  : declaration specifiers SEMICOLON {
    $$ = parser_create_ast_node(DECLARATION_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | declaration_specifiers init_declarator_list SEMICOLON {
    $$ = parser_create_ast_node(DECLARATION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  ;

declaration_specifiers
  : storage_class_specifier {
    $$ = parser_create_ast_node(DECLARATION_SPECIFIERS_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | storage_class_specifier declaration_specifiers {
    $$ = parser_create_ast_node(DECLARATION_SPECIFIERS_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  | type_specifier {
    $$ = parser_create_ast_node(DECLARATION_SPECIFIERS_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | type_specifier declaration_specifiers {
    $$ = parser_create_ast_node(DECLARATION_SPECIFIERS_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
```

```
  }
  | type_qualifier {
    $$ = parser_create_ast_node(DECLARATION_SPECIFIERS_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | type_qualifier declaration_specifiers {
    $$ = parser_create_ast_node(DECLARATION_SPECIFIERS_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  ;

init_declarator_list
  : init_declarator {
    $$ = parser_create_ast_node(INIT_DECLARATOR_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | init_declarator_list COMMA init_declarator {
    $$ = parser_create_ast_node(INIT_DECLARATOR_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $3);
  }
  ;

init_declarator
  : declarator {
    $$ = parser_create_ast_node(INIT_DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | declarator EQUAL initializer {
    $$ = parser_create_ast_node(INIT_DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

storage_class_specifier
  : TYPEDEF {
    $$ = parser_create_ast_node(
      STORAGE_CLASS_SPECIFIER_AST_NODE, "TYPEDEF"
    );
  }
  | EXTERN {
    $$ = parser_create_ast_node(
```

```
      STORAGE_CLASS_SPECIFIER_AST_NODE, "EXTERN"
    );
  }
  | STATIC {
    $$ = parser_create_ast_node(
      STORAGE_CLASS_SPECIFIER_AST_NODE, "STATIC"
    );
  }
  | AUTO {
    $$ = parser_create_ast_node(
      STORAGE_CLASS_SPECIFIER_AST_NODE, "AUTO"
    );
  }
  | REGISTER {
    $$ = parser_create_ast_node(
      STORAGE CLASS SPECIFIER AST NODE, "REGISTER"
    );
  }
  ;

type_specifier
  : VOID {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "VOID");
  }
  | CHAR {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "CHAR");
  }
  | SHORT {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "SHORT");
  }
  | INT {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "INT");
  }
  | LONG {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "LONG");
  }
  | FLOAT {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "FLOAT");
  }
  | DOUBLE {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "DOUBLE");
  }
  | SIGNED {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "SIGNED");
```

```
  }
  | UNSIGNED {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "UNSIGNED");
  }
  | struct_or_union_specifier {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | enum_specifier {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | type identifier {
    $$ = parser_create_ast_node(TYPE_SPECIFIER_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  ;

struct_or_union_specifier
  : struct_or_union identifier CO_BRACKET struct_declaration_list
    CC_BRACKET {
    $$ = parser_create_ast_node(
      STRUCT_OR_UNION_SPECIFIER_AST_NODE, ""
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
    graph_add_edge($$, $4);
  }
  | struct_or_union type_identifier CO_BRACKET
    struct_declaration_list CC_BRACKET {
    $$ = parser_create_ast_node(
      STRUCT_OR_UNION_SPECIFIER_AST_NODE, ""
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
    graph_add_edge($$, $4);
  }
  | struct_or_union CO_BRACKET struct_declaration_list CC_BRACKET {
    $$ = parser_create_ast_node(
      STRUCT_OR_UNION_SPECIFIER_AST_NODE, ""
    );
```

```
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | struct_or_union identifier {
    $$ = parser_create_ast_node(
      STRUCT_OR_UNION_SPECIFIER_AST_NODE, ""
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  | struct_or_union type_identifier {
    $$ = parser create ast node(
      STRUCT_OR_UNION_SPECIFIER_AST_NODE, ""
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  ;

struct_or_union
  : STRUCT {
    $$ = parser_create_ast_node(STRUCT_AST_NODE, "");
  }
  | UNION {
    $$ = parser_create_ast_node(UNION_AST_NODE, "");
  }
  ;

struct_declaration_list
  : struct_declaration {
    $$ = parser_create_ast_node(
      STRUCT_DECLARATION_LIST_AST_NODE, ""
    );

    graph_add_edge($$, $1);
  }
  | struct_declaration_list struct_declaration {
    $$ = parser_create_ast_node(
      STRUCT_DECLARATION_LIST_AST_NODE, ""
    );

    graph_copy_edges($$, $1);
```

```
    graph_add_edge($$, $2);
  }
  ;

struct_declaration
  : specifier_qualifier_list struct_declarator_list SEMICOLON {
    $$ = parser_create_ast_node(
      STRUCT_DECLARATION_AST_NODE, ""
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  ;

specifier qualifier list
  : type_specifier specifier_qualifier_list {
    $$ = parser_create_ast_node(
      SPECIFIER_QUALIFIER_LIST_AST_NODE, ""
    );

    graph_copy_edges($$, $1);
    graph_add_edge($$, $2);
  }
  | type_specifier {
    $$ = parser_create_ast_node(
      SPECIFIER_QUALIFIER_LIST_AST_NODE, ""
    );

    graph_add_edge($$, $1);
  }
  | type_qualifier specifier_qualifier_list {
    $$ = parser_create_ast_node(
      SPECIFIER_QUALIFIER_LIST_AST_NODE, ""
    );

    graph_copy_edges($$, $1);
    graph_add_edge($$, $2);
  }
  | type_qualifier {
    $$ = parser_create_ast_node(
      SPECIFIER_QUALIFIER_LIST_AST_NODE, ""
    );
```

```
    graph_add_edge($$, $1);
  }
  ;

struct_declarator_list
  : struct_declarator {
    $$ = parser_create_ast_node(STRUCT_DECLARATOR_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | struct_declarator_list COMMA struct_declarator {
    $$ = parser_create_ast_node(STRUCT_DECLARATOR_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph add edge($$, $3);
  }
  ;

struct_declarator
  : declarator {
    $$ = parser_create_ast_node(STRUCT_DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | declarator COLON constant_expression {
    $$ = parser_create_ast_node(STRUCT_DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | COLON constant_expression {
    $$ = parser_create_ast_node(STRUCT_DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  ;

enum_specifier
  : ENUM CO_BRACKET enumerator_list CC_BRACKET {
    $$ = parser_create_ast_node(ENUM_SPECIFIER_AST_NODE, "");
    graph_add_edge($$, $3);
  }
  | ENUM identifier CO_BRACKET enumerator_list CC_BRACKET {
    $$ = parser_create_ast_node(ENUM_SPECIFIER_AST_NODE, "");
    graph_add_edge($$, $2);
    graph_add_edge($$, $4);
  }
  | ENUM identifier {
    $$ = parser_create_ast_node(ENUM_SPECIFIER_AST_NODE, "");
```

```
    graph_add_edge($$, $2);
  }
  ;

enumerator_list
  : enumerator {
    $$ = parser_create_ast_node(ENUMERATOR_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | enumerator_list COMMA enumerator {
    $$ = parser_create_ast_node(ENUMERATOR_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph add edge($$, $3);
  }
  ;

enumerator
  : identifier {
    $$ = parser_create_ast_node(ENUMERATOR_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | identifier EQUAL constant_expression {
    $$ = parser_create_ast_node(ENUMERATOR_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

type_qualifier
  : CONST {
    $$ = parser_create_ast_node(TYPE_QUALIFIER_AST_NODE, "CONST");
  }
  | VOLATILE {
    $$ = parser_create_ast_node(TYPE_QUALIFIER_AST_NODE, "VOLATILE");
  }
  ;

declarator
  : pointer direct_declarator {
    $$ = parser_create_ast_node(DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  | direct_declarator {
```

```
    $$ = parser_create_ast_node(DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  ;

direct_declarator
  : identifier {
    $$ = parser_create_ast_node(DIRECT_DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | RO_BRACKET declarator RC_BRACKET {
    $$ = parser_create_ast_node(DIRECT_DECLARATOR_AST_NODE, "");
    graph add edge($$, $2);
  }
  | direct_declarator SO_BRACKET constant_expression SC_BRACKET {
    $$ = parser create ast node(DIRECT DECLARATOR AST NODE, "ARRAY");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | direct_declarator SO_BRACKET SC_BRACKET {
    $$ = parser_create_ast_node(DIRECT_DECLARATOR_AST_NODE, "ARRAY");
    graph_add_edge($$, $1);
  }
  | direct_declarator RO_BRACKET parameter_type_list RC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_DECLARATOR_AST_NODE, "FUNCTION"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | direct_declarator RO_BRACKET identifier_list RC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_DECLARATOR_AST_NODE, "FUNCTION"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | direct_declarator RO_BRACKET RC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_DECLARATOR_AST_NODE, "FUNCTION"
    );
```

```
    graph_add_edge($$, $1);
  }
  ;

pointer
  : ASTERISK {
    $$ = parser_create_ast_node(POINTER_AST_NODE, "");
  }
  | ASTERISK type_qualifier_list {
    $$ = parser_create_ast_node(POINTER_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  | ASTERISK pointer {
    $$ = parser_create_ast_node(POINTER_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  | ASTERISK type_qualifier_list pointer {
    $$ = parser_create_ast_node(POINTER_AST_NODE, "");
    graph_add_edge($$, $2);
    graph_add_edge($$, $3);
  }
  ;

type_qualifier_list
  : type_qualifier {
    $$ = parser_create_ast_node(TYPE_QUALIFIER_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | type_qualifier_list type_qualifier {
    $$ = parser_create_ast_node(TYPE_QUALIFIER_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $2);
  }
  ;

parameter_type_list
  : parameter_list {
    $$ = parser_create_ast_node(PARAMETER_TYPE_LIST_AST_NODE, "");
    graph_add_edge($$, $1);}
  | parameter_list COMMA ELLIPSIS {
    $$ = parser_create_ast_node(
      PARAMETER_TYPE_LIST_AST_NODE, "VAR_ARG"
    );
```

```
    graph_add_edge($$, $1);
  }
  ;

parameter_list
  : parameter_declaration {
    $$ = parser_create_ast_node(PARAMETER_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | parameter_list COMMA parameter_declaration {
    $$ = parser_create_ast_node(PARAMETER_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph add edge($$, $3);
  }
  ;

parameter_declaration
  : declaration_specifiers declarator {
    $$ = parser_create_ast_node(PARAMETER_DECLARATION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  | declaration_specifiers abstract_declarator {
    $$ = parser_create_ast_node(PARAMETER_DECLARATION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  | declaration_specifiers {
    $$ = parser_create_ast_node(PARAMETER_DECLARATION_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  ;

identifier_list
  : identifier {
    $$ = parser_create_ast_node(IDENTIFIER_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | identifier_list COMMA identifier {
    $$ = parser_create_ast_node(IDENTIFIER_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $3);
  }
  ;
```

```
type_name
  : specifier_qualifier_list {
    $$ = parser_create_ast_node(TYPE_NAME_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | specifier_qualifier_list abstract_declarator {
    $$ = parser_create_ast_node(TYPE_NAME_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  ;

abstract_declarator
  : pointer {
    $$ = parser create ast node(ABSTRACT DECLARATOR AST NODE, "");
    graph_add_edge($$, $1);
  }
  | direct_abstract_declarator {
    $$ = parser_create_ast_node(ABSTRACT_DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | pointer direct_abstract_declarator {
    $$ = parser_create_ast_node(ABSTRACT_DECLARATOR_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  ;

direct_abstract_declarator
  : RO_BRACKET abstract_declarator RC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, ""
    );

    graph_add_edge($$, $2);
  }
  | SO_BRACKET SC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, "ARRAY"
    );

  }
  | SO_BRACKET constant_expression SC_BRACKET {
```

```
    $$ = parser_create_ast_node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, "ARRAY"
    );

    graph_add_edge($$, $2);
  }
  | direct_abstract_declarator SO_BRACKET SC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, "ARRAY"
    );

    graph_add_edge($$, $1);
  }
  | direct_abstract_declarator SO_BRACKET constant_expression
    SC_BRACKET {
    $$ = parser create ast node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, "ARRAY"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | RO_BRACKET RC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, "FUNCTION"
    );

  }
  | RO_BRACKET parameter_type_list RC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, "FUNCTION"
    );

    graph_add_edge($$, $2);
  }
  | direct_abstract_declarator RO_BRACKET RC_BRACKET {
    $$ = parser_create_ast_node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, "FUNCTION"
    );

    graph_add_edge($$, $1);
  }
  | direct_abstract_declarator RO_BRACKET parameter_type_list
    RC_BRACKET {
```

```
    $$ = parser_create_ast_node(
      DIRECT_ABSTRACT_DECLARATOR_AST_NODE, "FUNCTION"
    );

    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  ;

initializer
  : assignment_expression {
    $$ = parser_create_ast_node(INITIALIZER_AST_NODE, "");
    graph add edge($$, $1);
  }
  | CO_BRACKET initializer_list CC_BRACKET {
    $$ = parser create ast node(INITIALIZER AST NODE, "");
    graph_add_edge($$, $2);
  }
  | CO_BRACKET initializer_list COMMA CC_BRACKET {
    $$ = parser_create_ast_node(INITIALIZER_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  ;

initializer_list
  : initializer {
    $$ = parser_create_ast_node(INITIALIZER_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | initializer_list COMMA initializer {
    $$ = parser_create_ast_node(INITIALIZER_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $3);
  }
  ;

statement
  : labeled_statement {
    $$ = parser_create_ast_node(STATEMENT_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | compound_statement {
    $$ = parser_create_ast_node(STATEMENT_AST_NODE, "");
    graph_add_edge($$, $1);
```

```
  }
  | expression_statement {
    $$ = parser_create_ast_node(STATEMENT_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | selection_statement {
    $$ = parser_create_ast_node(STATEMENT_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | iteration_statement {
    $$ = parser_create_ast_node(STATEMENT_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | jump_statement {
    $$ = parser_create_ast_node(STATEMENT_AST_NODE, "");
    graph add edge($$, $1);
  }
  ;

labeled_statement
  : identifier COLON statement {
    $$ = parser_create_ast_node(LABELED_STATEMENT_AST_NODE, "LABEL");
    graph_add_edge($$, $1);
    graph_add_edge($$, $3);
  }
  | CASE constant_expression COLON statement {
    $$ = parser_create_ast_node(LABELED_STATEMENT_AST_NODE, "CASE");
    graph_add_edge($$, $2);
    graph_add_edge($$, $4);
  }
  | DEFAULT COLON statement {
    $$ = parser_create_ast_node(
      LABELED_STATEMENT_AST_NODE, "DEFAULT"
    );

    graph_add_edge($$, $3);
  }
  ;

compound_statement
  : CO_BRACKET CC_BRACKET {
    $$ = parser_create_ast_node(COMPOUND_STATEMENT_AST_NODE, "");
  }
  | CO_BRACKET statement_list CC_BRACKET {
```

```
    $$ = parser_create_ast_node(COMPOUND_STATEMENT_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  | CO_BRACKET declaration_list CC_BRACKET {
    $$ = parser_create_ast_node(COMPOUND_STATEMENT_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  | CO_BRACKET declaration_list statement_list CC_BRACKET {
    $$ = parser_create_ast_node(COMPOUND_STATEMENT_AST_NODE, "");
    graph_add_edge($$, $2);
    graph_add_edge($$, $3);
  }
  ;

declaration_list
  : declaration {
    $$ = parser_create_ast_node(DECLARATION_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | declaration_list declaration {
    $$ = parser_create_ast_node(DECLARATION_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $2);
  }
  ;

statement_list
  : statement {
    $$ = parser_create_ast_node(STATEMENT_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | statement_list statement {
    $$ = parser_create_ast_node(STATEMENT_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $2);
  }
  ;

expression_statement
  : SEMICOLON {
    $$ = parser_create_ast_node(
      EXPRESSION_STATEMENT_AST_NODE, "EMPTY"
    );
  }
```

```
  | expression SEMICOLON {
    $$ = parser_create_ast_node(EXPRESSION_STATEMENT_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  ;

selection_statement
  : IF RO_BRACKET expression RC_BRACKET statement %prec THEN {
    $$ = parser_create_ast_node(SELECTION_STATEMENT_AST_NODE, "IF");
    graph_add_edge($$, $3);
    graph_add_edge($$, $5);
  }
  | IF RO BRACKET expression RC BRACKET statement ELSE statement {
    $$ = parser_create_ast_node(
      SELECTION_STATEMENT_AST_NODE, "IF_ELSE"
    );

    graph_add_edge($$, $3);
    graph_add_edge($$, $5);
    graph_add_edge($$, $7);
  }
  | SWITCH RO_BRACKET expression RC_BRACKET statement {
    $$ = parser_create_ast_node(
      SELECTION_STATEMENT_AST_NODE, "SWITCH"
    );

    graph_add_edge($$, $3);
    graph_add_edge($$, $5);
  }
  ;

iteration_statement
  : WHILE RO_BRACKET expression RC_BRACKET statement {
    $$ = parser_create_ast_node(
      ITERATION_STATEMENT_AST_NODE, "WHILE"
    );

    graph_add_edge($$, $3);
    graph_add_edge($$, $5);
  }
  | DO statement WHILE RO_BRACKET expression RC_BRACKET SEMICOLON {
    $$ = parser_create_ast_node(
      ITERATION_STATEMENT_AST_NODE, "DO_WHILE"
    );
```

```
    graph_add_edge($$, $2);
    graph_add_edge($$, $5);
  }
  | FOR RO_BRACKET expression_statement expression_statement
    RC_BRACKET statement {
    $$ = parser_create_ast_node(ITERATION_STATEMENT_AST_NODE, "FOR");
    graph_add_edge($$, $3);
    graph_add_edge($$, $4);
    graph_add_edge($$, $6);
  }
  | FOR RO_BRACKET expression_statement expression_statement
    expression RC BRACKET statement {
    $$ = parser_create_ast_node(ITERATION_STATEMENT_AST_NODE, "FOR");
    graph_add_edge($$, $3);
    graph add edge($$, $4);
    graph_add_edge($$, $5);
    graph_add_edge($$, $7);
  }
  ;

jump_statement
  : GOTO identifier SEMICOLON {
    $$ = parser_create_ast_node(
      JUMP_STATEMENT_GOTO_AST_NODE, ""
    );

    graph_add_edge($$, $2);
  }
  | CONTINUE SEMICOLON {
    $$ = parser_create_ast_node(
      JUMP_STATEMENT_CONTINUE_AST_NODE, ""
    );

  }
  | BREAK SEMICOLON {
    $$ = parser_create_ast_node(
      JUMP_STATEMENT_BREAK_AST_NODE, ""
    );

  }
  | RETURN SEMICOLON {
    $$ = parser_create_ast_node(
      JUMP_STATEMENT_RETURN_AST_NODE, ""
```

```
    );

  }
  | RETURN expression SEMICOLON {
    $$ = parser_create_ast_node(
      JUMP_STATEMENT_RETURN_AST_NODE, ""
    );

    graph_add_edge($$, $2);
  }
  ;

translation unit
  : external_declaration {
    $$ = parser_create_ast_node(TRANSLATION_UNIT_AST_NODE, "");
    graph add edge($$, $1);
  }
  | translation_unit external_declaration {
    $$ = parser_create_ast_node(TRANSLATION_UNIT_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $2);
  }
  ;

external_declaration
  : function_definition {
    $$ = parser_create_ast_node(EXTERNAL_DECLARATION_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | declaration {
    $$ = parser_create_ast_node(EXTERNAL_DECLARATION_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | comment {
    $$ = parser_create_ast_node(EXTERNAL_DECLARATION_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  ;

function_definition
  : declaration_specifiers declarator declaration_list
    compound_statement {
    $$ = parser_create_ast_node(FUNCTION_DEFINITION_AST_NODE, "");
    graph_add_edge($$, $1);
```

```
    graph_add_edge($$, $2);
    graph_add_edge($$, $3);
    graph_add_edge($$, $4);
  }
  | declaration_specifiers declarator compound_statement {
    $$ = parser_create_ast_node(FUNCTION_DEFINITION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
    graph_add_edge($$, $3);
  }
  | declarator declaration_list compound_statement {
    $$ = parser_create_ast_node(FUNCTION_DEFINITION_AST_NODE, "");
    graph add edge($$, $1);
    graph_add_edge($$, $2);
    graph_add_edge($$, $3);
  }
  | declarator compound_statement {
    $$ = parser_create_ast_node(FUNCTION_DEFINITION_AST_NODE, "");
    graph_add_edge($$, $1);
    graph_add_edge($$, $2);
  }
  ;

program
  : translation_unit {
    counter += 1;
    current_ast_node = PROGRAM_AST_NODE;

    type = malloc(sizeof(char) * (strlen(current_ast_node) + 1));
    assert(type != NULL);
    strcpy(type, current_ast_node);

    $$ = graph_add_node(ast, PROGRAM_AST_NODE);
    graph_add_edge($$, $1);
    graph_node_set_field($$, "type", type);
  }
  ;

comment
  : OPEN_COMMENT v_rule_list CLOSE_COMMENT {
    $$ = parser_create_ast_node(COMMENT_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  | OPEN_COMMENT CLOSE_COMMENT {
```

```
    $$ = parser_create_ast_node(COMMENT_AST_NODE, "");
  }
  ;

v_rule_list
  : v_rule {
    $$ = parser_create_ast_node(V_RULE_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | v_rule_list v_rule {
    $$ = parser_create_ast_node(V_RULE_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph add edge($$, $2);
  }
  ;

v_rule
  : v_rule_param {
    $$ = parser_create_ast_node(V_RULE_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | v_rule_recursive {
    $$ = parser_create_ast_node(V_RULE_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | v_rule_description {
    $$ = parser_create_ast_node(V_RULE_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  ;

v_rule_param
  : V_PARAM v_keyword_list V_COMMA v_identifier v_string {
    $$ = parser_create_ast_node(V_RULE_PARAM_AST_NODE, "");
    graph_add_edge($$, $2);
    graph_add_edge($$, $4);
    graph_add_edge($$, $5);
  }
  | V_PARAM v_keyword_list V_COMMA v_identifier {
    $$ = parser_create_ast_node(V_RULE_PARAM_AST_NODE, "");
    graph_add_edge($$, $2);
    graph_add_edge($$, $4);
  }
  | V_PARAM v_identifier v_string {
```

```
    $$ = parser_create_ast_node(V_RULE_PARAM_AST_NODE, "");
    graph_add_edge($$, $2);
    graph_add_edge($$, $3);
  }
  | V_PARAM v_identifier {
    $$ = parser_create_ast_node(V_RULE_PARAM_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  | V_PARAM v_keyword_list v_string {
    $$ = parser_create_ast_node(V_RULE_PARAM_AST_NODE, "");
    graph_add_edge($$, $2);
    graph_add_edge($$, $3);
  }
  | V_PARAM v_keyword_list {
    $$ = parser_create_ast_node(V_RULE_PARAM_AST_NODE, "");
    graph add edge($$, $2);
  }
  ;

v_rule_recursive
  : V_RECURSIVE {
    $$ = parser_create_ast_node(V_RULE_RECURSIVE_AST_NODE, "");
  }
  ;

v_rule_description
  : V_DESCRIPTION v_string {
    $$ = parser_create_ast_node(V_RULE_DESCRIPTION_AST_NODE, "");
    graph_add_edge($$, $2);
  }
  ;

v_keyword_list
  : v_keyword {
    $$ = parser_create_ast_node(V_KEYWORD_LIST_AST_NODE, "");
    graph_add_edge($$, $1);
  }
  | v_keyword_list V_PLUS v_keyword {
    $$ = parser_create_ast_node(V_KEYWORD_LIST_AST_NODE, "");
    graph_copy_edges($$, $1);
    graph_add_edge($$, $3);
  }
  ;
```

```
v_identifier
  : V_IDENTIFIER {
    $$ = parser_create_ast_node(V_IDENTIFIER_AST_NODE, $1);
  }
  ;

v_string
  : V_STRING {
    $$ = parser_create_ast_node(V_STRING_AST_NODE, $1);
  }
  ;

v keyword
  : V_KEYWORD {
    $$ = parser_create_ast_node(V_KEYWORD_AST_NODE, $1);
  }
  ;

%%

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/**
 * yyerror
 */
void yyerror(char *message) {
  fprintf(
    stderr,
    "%s at line %d, column %d: \"%s\"\n",
    message,
    yylloc.first_line,
    yylloc.first_column,
    yylval.string_v
  );

  exit(-1);
}

/**
 * ignore_deep_comments
 */
void ignore_deep_comments(void) {
```

```
  while (
    strcmp(current_ast_node, PROGRAM_AST_NODE) != 0 &&
    strcmp(current_ast_node, TRANSLATION_UNIT_AST_NODE) != 0 &&
    match->type == OPEN_COMMENT
  ) {
    while (match->type != CLOSE_COMMENT) {
      dfa_match_free(match);
      match = lexer_get_next_token(lexer);
    }

    dfa_match_free(match);
    match = lexer_get_next_token(lexer);
  }
}

/**
 * yylex
 */
int yylex(void) {
  if (match != NULL) {
    dfa_match_free(match);
  }

  match = lexer_get_next_token(lexer);

  ignore_deep_comments();

  if (match != NULL) {
    yylval.string_v = match->value;

    yylloc.first_line = match->line;
    yylloc.last_line = match->line;
    yylloc.first_column = match->column;
    yylloc.last_column = match->column + (int) strlen(match->value);

    return match->type;
  }

  return YYEOF;
}

/**
 * parser_parse_file
 *
```

```
 * [param] IP, custom_lexer
 */
Graph* parser_parse_file(Lexer *custom_lexer) {
  lexer = custom_lexer;

  ast = graph_create();

  identifier = buffer_create();

  yyparse();

  buffer_free(identifier);

  return ast;
}

/********** CLEAN-UP FUNCTIONS **********/

/**
 * ast_node_free
 *
 * [param] IOP, ast_node
 */
void ast_node_free(GraphNode *ast_node) {
  HashMapIterator *iterator;

  for (
    iterator = hashmap_get_iterator(ast_node->fields);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    free(iterator->value);
    iterator->value = NULL;
  }
}

/**
 * ast_free
 *
 * [param] IOP, ast
 */
void ast_free(Graph *ast) {
  HashMapIterator *iterator;
```

```
  for (
    iterator = hashmap_get_iterator(ast->nodes);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    ast_node_free(iterator->value);
  }

  graph_free(ast);
}
```

8.4 Analyzer

8.4.1 `mod_analyzer`

8.4.1.1 `mod_analyzer.h`

```
#include "lib_graph.h"

void analyzer_analyze(const Graph *ast);
```

8.4.1.2 `mod_analyzer.c`

```
#include "mod_analyzer.h"
#include "mod_analyzer_report_program.h"

/**
 * analyzer_analyze
 *
 * [param] IP, ast
 */
void analyzer_analyze(const Graph *ast) {
  AnalyzerReportProgram *report_program =
    analyzer_report_program_create(ast);

  /* Clean-up */

  analyzer_report_program_free(report_program);
}
```

8.4.2 mod_analyzer_common

8.4.2.1 mod_analyzer_common.h

```
#include "lib_graph.h"

List* analyzer_get_identifier_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
);

bool analyzer_check_array_access(
  const Graph *ast,
  const GraphNode *ast_node
);

bool analyzer_check_member_access(
  const Graph *ast,
  const GraphNode *ast_node
);

bool analyzer_check_pointer_access(
  const Graph *ast,
  const GraphNode *ast node
);

char* analyzer_get_ast_node_type(
  const GraphNode *ast_node
);

char* analyzer_get_ast_node_value(
  const GraphNode *ast_node
);

char* analyzer_get_identifier_ast_node_value(
  const Graph *ast,
  const GraphNode *node
);
```

8.4.2.2 `mod_analyzer_common.c`

```
#include <string.h>

#include "mod_analyzer_common.h"
#include "mod_parser.h"

/*
 * analyzer_get_identifier_ast_node_list
 *
 * AST Exploration function. Retrieves the list of first-level
 * identifiers and function identifiers among the subtree of an AST
 * node using DFS preorder.
 *
 * [param] IP, ast
 * [param] IP, ast node
 * [param] OR - list of AST nodes
 */
List* analyzer_get_identifier_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
) {
  ListNode *iterator;

  GraphNode *term_ast_node;

  char *term_type;

  /* Retrieving the list of identifiers and calls to function*/

  List *roots = list_create(1, ast_node);

  List *labels = list_create(
    2, IDENTIFIER_AST_NODE, FUNCTION_CALL_AST_NODE
  );

  List *ast_node_list = graph_dfs_preorder_find(
    ast, roots, "type", labels
  );

  List *result = list_create_empty();

  for (
```

```
    iterator = list_get_iterator(ast_node_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    term_ast_node = iterator->value;
    term_type = analyzer_get_ast_node_type(term_ast_node);

    if (strcmp(term_type, IDENTIFIER_AST_NODE) == 0) {
      list_push_back(result, term_ast_node);
    }
  }

  /* Clean-up */

  list_free(roots);
  list free(labels);
  list_free(ast_node_list);

  return result;
}

/*
 * analyzer_check_array_access
 *
 * Detects array accesses.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - resulting boolean
 */
bool analyzer_check_array_access(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *roots   = list_create(1, ast_node);
  List *labels  = list_create(1, ARRAY_ACCESS_AST_NODE);

  GraphNode *identifier_ast_node = graph_dfs_preorder_find_first(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
```

```
  list_free(labels);

  return identifier_ast_node != NULL;
}

/*
 * analyzer_check_member_access
 *
 * Detects member accesses.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - resulting boolean
 */
bool analyzer_check_member_access(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *roots = list_create(1, ast_node);
  List *labels = list_create(1, MEMBER_ACCESS_AST_NODE);

  GraphNode *identifier_ast_node = graph_dfs_preorder_find_first(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return identifier_ast_node != NULL;
}

/*
 * analyzer_check_pointer_access
 *
 * Detects pointer accesses.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - resulting boolean
 */
bool analyzer_check_pointer_access(
  const Graph *ast,
```

```
  const GraphNode *ast_node
) {
  List *roots = list_create(1, ast_node);
  List *labels = list_create(1, POINTER_ACCESS_AST_NODE);

  GraphNode *identifier_ast_node = graph_dfs_preorder_find_first(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list free(labels);

  return identifier_ast_node != NULL;
}

/*
 * analyzer_get_ast_node_type
 *
 * Returns the value of the type field of and ASTNode.
 *
 * [param] IP, ast_node
 * [param] OR - value of the type field
 */
char* analyzer_get_ast_node_type(const GraphNode *ast_node) {
  return graph_node_get_field(ast_node, "type");
}

/*
 * analyzer_get_ast_node_value
 *
 * Returns the value of the value field of and ASTNode.
 *
 * [param] IP, ast_node
 * [param] OR - value of the value field
 */
char* analyzer_get_ast_node_value(const GraphNode *ast_node) {
  return graph_node_get_field(ast_node, "value");
}

/*
 * analyzer_get_identifier_ast_node_value
 *
```

```
 * Returns the value of the first identifier found among the subtree
 * of an ASTNode.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - the value of the identifier
 */
char* analyzer_get_identifier_ast_node_value(
  const Graph *ast,
  const GraphNode *node
) {
  List *roots   = list_create(1, node);
  List *labels  = list create(1, IDENTIFIER AST NODE);

  GraphNode *identifier_ast_node = graph_dfs_preorder_find_first(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return graph_node_get_field(identifier_ast_node, "value");
}
```

8.4.3 mod_analyzer_destination_map

8.4.3.1 mod_analyzer_destination_map.h

```
#include "lib_hashmap.h"

#define DESTINATION_IP "IP"
#define DESTINATION_IF "IF"
#define DESTINATION_IK "IK"
#define DESTINATION_OP "OP"
#define DESTINATION_OF "OF"
#define DESTINATION_OV "OV"
#define DESTINATION_OR "OR"
#define DESTINATION_IOP "IOP"
#define DESTINATION_IOF "IOF"

HashMap* analyzer_destination_map_create(void);
int* analyzer_destination_flag_create(void);
void analyzer_destination_map_free(HashMap *destination_map);
```

8.4.3.2 mod_analyzer_destination_map.c

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "lib_hashmap.h"
#include "mod_analyzer_destination_map.h"

/********** LIBRARY FUNCTIONS **********/

/*
 * analyzer_destination_map_create
 *
 * Creates a new destination map.
 *
 * [param] OR - the newly created destination map.
 */
```

```
HashMap *analyzer_destination_map_create(void) {
  HashMap *destination_map = hashmap_create();

  hashmap_set(
    destination_map,
    DESTINATION_IP,
    analyzer_destination_flag_create()
  );

  hashmap_set(
    destination_map,
    DESTINATION_IF,
    analyzer destination flag create()
  );

  hashmap set(
    destination_map,
    DESTINATION_IK,
    analyzer_destination_flag_create()
  );

  hashmap_set(
    destination_map,
    DESTINATION_OP,
    analyzer_destination_flag_create()
  );

  hashmap_set(
    destination_map,
    DESTINATION_OF,
    analyzer_destination_flag_create()
  );

  hashmap_set(
    destination_map,
    DESTINATION_OV,
    analyzer_destination_flag_create()
  );

  return destination_map;
}

/*
 * analyzer_destination_flag_create
```

```
 *
 * Creates a pointer to a boolean value.
 *
 * [param] OR - pointer to the boolean value.
 */
int *analyzer_destination_flag_create(void) {
  int *destination_flag = malloc(sizeof(int));
  assert(destination_flag != NULL);

  *destination_flag = 0;

  return destination_flag;
}

/*
 * analyzer destination map free
 *
 * Frees the memory allocated by a destination map.
 *
 * [param] IOP, destination_map
 */
void analyzer_destination_map_free(HashMap *destination_map) {
  HashMapIterator *iterator;

  for (
    iterator = hashmap_get_iterator(destination_map);
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    free(iterator->value);
    iterator->value = NULL;
  }

  hashmap_free(destination_map);
}
```

8.4.4 mod_analyzer_report_documentation

8.4.4.1 mod_analyzer_report_documentation.h

```
#include "lib_hashmap.h"
#include "lib_graph.h"
#include "mod_analyzer_report_identifier.h"

#ifndef MOD_ANALYZER_REPORT_DOCUMENTATION
#define MOD_ANALYZER_REPORT_DOCUMENTATION

typedef struct AnalyzerReportDocumentation {
  HashMap* parameters;
  bool destination_or;
} AnalyzerReportDocumentation;

#endif

AnalyzerReportDocumentation* analyzer_report_documentation_create(
  const Graph *ast,
  const GraphNode *ast_node
);

void analyzer report documentation free(
  AnalyzerReportDocumentation *analyzer_report_documentation
);
```

8.4.4.2 mod_analyzer_report_documentation.c

```
#include <stdlib.h>
#include <string.h>

#include "mod_analyzer_common.h"
#include "mod_analyzer_destination_map.h"
#include "mod_analyzer_report_documentation.h"
#include "mod_parser.h"

/********** RESERVED FUNCTION DECLARATIONS **********/
```

```
GraphNode *ard_get_identifier_ast_node(
  const Graph *ast,
  const GraphNode *node
);

List *ard_get_keyword_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
);

List *ard_get_rule_param_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
);

void ard_parameter_register_rule(
  const Graph *ast,
  const GraphNode *ast_node,
  AnalyzerReportDocumentation *report_documentation
);

void ard_parameter_register_rule_named(
  const List *keyword_ast_node_list,
  const GraphNode *identifier_ast_node,
  AnalyzerReportDocumentation *report_documentation
);

void ard_parameter_register_rule_unnamed(
  const List *keyword_ast_node_list,
  AnalyzerReportDocumentation *report_documentation
);

/********** LIBRARY FUNCTIONS **********/

/*
 * analyzer_report_documentation_create
 *
 * Creates a new AnalyzerReportDocumentation.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - the newly created AnalyzerReportDocumentation.
 */
```

```
AnalyzerReportDocumentation *analyzer_report_documentation_create(
  const Graph *ast,
  const GraphNode *ast_node
) {
  ListNode *iterator;

  List *param_ast_node_list = ard_get_rule_param_ast_node_list(
    ast, ast_node
  );

  AnalyzerReportDocumentation *report_documentation = malloc(
    sizeof(AnalyzerReportDocumentation)
  );

  report_documentation->parameters = hashmap_create();
  report documentation->destination or = false;

  for (
    iterator = list_get_iterator(param_ast_node_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    ard_parameter_register_rule(
      ast, iterator->value, report_documentation
    );
  }

  /* Clean-up */

  list_free(param_ast_node_list);

  return report_documentation;
}

/*
 * analyzer_report_documentation_free
 *
 * Frees the memory allocated for the AnalyzerReportDocumentation.
 *
 * [param] IOP, analyzer_report_documentation
 */
void analyzer_report_documentation_free(
  AnalyzerReportDocumentation *analyzer_report_documentation
) {
```

```
  HashMapIterator *iterator;

  AnalyzerReportIdentifier *current;

  for (
    iterator = hashmap_get_iterator(
      analyzer_report_documentation->parameters
    );
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    current = iterator->value;

    analyzer_destination_map_free(current->destination_map);

    free(current);
    current = NULL;
  }

  hashmap_free(analyzer_report_documentation->parameters);

  free(analyzer_report_documentation);
  analyzer_report_documentation = NULL;
}

/********** RESERVED FUNCTIONS **********/

/*
 * ard_get_identifier_ast_node
 *
 * AST Exploration function. Retrieves the first v_keyword among the
 * subtree of an AST node using DFS preorder.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - the AST node
 */
GraphNode *ard_get_identifier_ast_node(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *roots   = list_create(1, ast_node);
  List *labels  = list_create(1, V_IDENTIFIER_AST_NODE);
```

```
  GraphNode *identifier_ast_node = graph_dfs_preorder_find_first(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return identifier_ast_node;
}

/*
 * ard_get_keyword_ast_node_list
 *
 * AST Exploration function. Retrieves the list of v keywords among
 * the subtree of an AST node using DFS postorder.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - list of AST nodes
 */
List *ard_get_keyword_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *roots   = list_create(1, ast_node);
  List *labels  = list_create(1, V_KEYWORD_AST_NODE);

  List *result = graph_dfs_postorder_find_all(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return result;
}

/*
 * ard_get_rule_param_ast_node_list
 *
```

```
 * AST Exploration function. Retrieves the list of v_rule_params
 * among the subtree of an AST node using DFS postorder.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - list of AST nodes
 */
List *ard_get_rule_param_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *roots   = list_create(1, ast_node);
  List *labels  = list create(1, V RULE PARAM AST NODE);

  List *result = graph_dfs_postorder_find_all(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return result;
}

/*
 * ard_parameter_register_rule
 *
 * Updates an AnalyzerReportDocumentation registering a documentation
 * rule.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] IOP, report_documentation
 */
void ard_parameter_register_rule(
  const Graph *ast,
  const GraphNode *ast_node,
  AnalyzerReportDocumentation *report_documentation
) {
  GraphNode *identifier_ast_node = ard_get_identifier_ast_node(
    ast, ast_node
  );
```

```
  List *keyword_ast_node_list = ard_get_keyword_ast_node_list(
    ast, ast_node
  );

  if (identifier_ast_node != NULL) {
    ard_parameter_register_rule_named(
      keyword_ast_node_list,
      identifier_ast_node,
      report_documentation
    );
  }
  else {
    ard_parameter_register_rule_unnamed(
      keyword_ast_node_list, report_documentation
    );
  }

  /* Clean-up */

  list_free(keyword_ast_node_list);
}

/*
 * ard_parameter_register_rule_named
 *
 * Updates an AnalyzerReportDocumentation registering a documentation
 * named rule.
 *
 * [param] IP, keyword_ast_node_list
 * [param] IP, identifier_ast_node
 * [param] IOP, report_documentation
 */
void ard_parameter_register_rule_named(
  const List *keyword_ast_node_list,
  const GraphNode *identifier_ast_node,
  AnalyzerReportDocumentation *report_documentation
) {
  ListNode *iterator;

  bool *destination_flag;

  char *keyword_value;
```

```
char *identifier_value = analyzer_get_ast_node_value(
  identifier_ast_node
);

AnalyzerReportIdentifier *report_parameter = hashmap_get(
  report_documentation->parameters, identifier_value
);

if (report_parameter == NULL) {
  report_parameter = analyzer_report_identifier_create();

  hashmap_set(
    report documentation->parameters,
    identifier_value,
    report_parameter
  );
}

for (
  iterator = list_get_iterator(keyword_ast_node_list);
  iterator != NULL;
  iterator = list_iterate(iterator)
) {
  keyword_value = analyzer_get_ast_node_value(iterator->value);

  if (strcmp(keyword_value, DESTINATION_IP) == 0) {
    destination_flag = hashmap_get(
      report_parameter->destination_map, DESTINATION_IP
    );

    *destination_flag = 1;
  }
  else if (strcmp(keyword_value, DESTINATION_IF) == 0) {
    destination_flag = hashmap_get(
      report_parameter->destination_map, DESTINATION_IF
    );

    *destination_flag = 1;
  }
  else if (strcmp(keyword_value, DESTINATION_IK) == 0) {
    destination_flag = hashmap_get(
      report_parameter->destination_map, DESTINATION_IK
    );
```

```
      *destination_flag = 1;
    }
    else if (strcmp(keyword_value, DESTINATION_OV) == 0) {
      destination_flag = hashmap_get(
        report_parameter->destination_map, DESTINATION_OV
      );

      *destination_flag = 1;
    }
    else if (strcmp(keyword_value, DESTINATION_OF) == 0) {
      destination_flag = hashmap_get(
        report_parameter->destination_map, DESTINATION_OF
      );

      *destination_flag = 1;
    }
    else if (strcmp(keyword_value, DESTINATION_OP) == 0) {
      destination_flag = hashmap_get(
        report_parameter->destination_map, DESTINATION_OP
      );

      *destination_flag = 1;
    }
    else if (strcmp(keyword_value, DESTINATION_IOP) == 0) {
      destination_flag = hashmap_get(
        report_parameter->destination_map, DESTINATION_IP
      );

      *destination_flag = 1;

      destination_flag = hashmap_get(
        report_parameter->destination_map, DESTINATION_OP
      );

      *destination_flag = 1;
    }
    else if (strcmp(keyword_value, DESTINATION_IOF) == 0) {
      destination_flag = hashmap_get(
        report_parameter->destination_map, DESTINATION_IF
      );

      *destination_flag = 1;

      destination_flag = hashmap_get(
```

```
        report_parameter->destination_map, DESTINATION_OF
      );

      *destination_flag = 1;
    }
    else if (strcmp(keyword_value, DESTINATION_OR) == 0) {
      report_documentation->destination_or = true;
    }
  }
}

/*
 * ard parameter register rule unnamed
 *
 * Updates an AnalyzerReportDocumentation registering a documentation
 * unnamed rule.
 *
 * [param] IP, keyword_ast_node_list
 * [param] IOP, report_documentation
 */
void ard_parameter_register_rule_unnamed(
  const List *keyword_ast_node_list,
  AnalyzerReportDocumentation *report_documentation
) {
  ListNode *iterator;

  char *keyword_value;

  for (
    iterator = list_get_iterator(keyword_ast_node_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    keyword_value = analyzer_get_ast_node_value(iterator->value);

    if (strcmp(keyword_value, DESTINATION_OR) == 0) {
      report_documentation->destination_or = true;
    }
  }
}
```

8.4.5 mod_analyzer_report_function

8.4.5.1 mod_analyzer_report_function.h

```
#include "lib_graph.h"
#include "mod_analyzer_report_identifier.h"
#include "mod_analyzer_report_parameter.h"
#include "mod_analyzer_report_program.h"

#ifndef MOD_ANALYZER_REPORT_FUNCTION
#define MOD_ANALYZER_REPORT_FUNCTION

typedef struct AnalyzerReportFunction {
  Graph *identifier_dependencies;
  HashMap *identifier_reports;
  AnalyzerReportParameterList *parameter_list_report;
  bool process_documentation;
} AnalyzerReportFunction;

#endif

AnalyzerReportFunction *analyzer_report_function_compute(
  const Graph *ast,
  const GraphNode *function ast node,
  const AnalyzerReportProgram *report_program
);

AnalyzerReportFunction *analyzer_report_function_create(
  int parameters_count
);

void analyzer_report_function_add_identifier_dependency(
  const AnalyzerReportFunction *report_function,
  const char *parent_identifier,
  const char *child_identifier
);

void analyzer_report_function_free(
  AnalyzerReportFunction *report_function
);
```

```
void analyzer_report_function_register_identifier(
  const AnalyzerReportFunction *report_function,
  const char *identifier
);

void analyzer_report_function_reset_identifier(
  const AnalyzerReportFunction *report_function,
  const char *identifier
);
```

8.4.5.2 `mod_analyzer_report_function.c`

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "mod_analyzer_common.h"
#include "mod_analyzer_destination_map.h"
#include "mod_analyzer_report_function.h"
#include "mod_analyzer_report_function_operation_assignment.h"
#include "mod_analyzer_report_function_operation_call.h"
#include "mod_analyzer_report_identifier.h"
#include "mod_analyzer_report_parameter.h"
#include "mod_parser.h"

GraphNode *arf_get_compound_statement_ast_node(
  const Graph *ast,
  const GraphNode *ast node
);

List *arf get identifiers ast node list(
  const Graph *ast,
  const GraphNode *ast_node
);

List *arf_get_operations_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
);
```

```
List *arf_get_parameters_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
);

bool arf_check_non_void_return_type(
  const Graph *ast,
  const GraphNode *ast_node
);

void arf_parameter_list_compute(
  const AnalyzerReportFunction *report_function
);

void arf_parameter_list_report_initialize(
  const Graph *ast,
  const GraphNode *function_ast_node,
  const List *parameter_ast_node_list,
  const AnalyzerReportFunction *report_function
);

/********** LIBRARY FUNCTIONS **********/

/*
 * analyzer_report_function_compute
 *
 * Given a function_declaration AST node, computes an
 * AnalyzerReportFunction.
 *
 * [param] IP, ast
 * [param] IP, function_ast_node
 * [param] IOP, report_program
 * [param] OR - the newly created AnalyzerReportFunction
 */
AnalyzerReportFunction *analyzer_report_function_compute(
  const Graph *ast,
  const GraphNode *function_ast_node,
  const AnalyzerReportProgram *report_program
) {
  List *operation_ast_node_list, *identifiers_ast_node_list;

  ListNode *iterator;

  GraphNode *operation_ast_node;
```

```
char *operation_ast_node_type, *identifier;

/* Retrieving the list of the parameters of the function */

List *parameter_ast_node_list = arf_get_parameters_ast_node_list(
  ast, function_ast_node
);

/* Creating the report of the function */

AnalyzerReportFunction *report_function =
  analyzer report function create(
    (int) parameter_ast_node_list->length
  );

/* Initializing the parameter list report */

arf_parameter_list_report_initialize(
  ast,
  function_ast_node,
  parameter_ast_node_list,
  report_function
);

/* Retrieving and registering each identifier */

identifiers_ast_node_list = arf_get_identifiers_ast_node_list(
  ast, function_ast_node
);

for (
  iterator = list_get_iterator(identifiers_ast_node_list);
  iterator != NULL;
  iterator = list_iterate(iterator)
) {
  identifier = analyzer_get_identifier_ast_node_value(
    ast, iterator->value
  );

  analyzer_report_function_register_identifier(
    report_function, identifier
  );
}
```

```
arf_parameter_list_compute(report_function);

/* Retrieving the ordered list of operations */

operation_ast_node_list = arf_get_operations_ast_node_list(
  ast, function_ast_node
);

for (
  iterator = list_get_iterator(operation_ast_node_list);
  iterator != NULL;
  iterator = list iterate(iterator)
) {
  operation_ast_node = iterator->value;

  operation_ast_node_type = analyzer_get_ast_node_type(
    operation_ast_node
  );

  if (
    strcmp(operation_ast_node_type, INIT_DECLARATOR_AST_NODE) == 0
  ) {
    analyzer_report_function_operation_declaration_compute(
      ast, operation_ast_node, report_function
    );
  }
  else if (
    strcmp(operation_ast_node_type, ASSIGNMENT_EXPRESSION_AST_NODE)
      == 0
  ) {
    analyzer_report_function_operation_assignment_compute(
      ast, operation_ast_node, report_function
    );
  }
  else if (
    strcmp(operation_ast_node_type, FUNCTION_CALL_AST_NODE) == 0
  ) {
    analyzer_report_function_operation_call_compute(
      ast, operation_ast_node, report_program, report_function
    );
  }

  arf_parameter_list_compute(report_function);
```

```
  }

  /* Clean-up */

  list_free(identifiers_ast_node_list);
  list_free(parameter_ast_node_list);
  list_free(operation_ast_node_list);

  return report_function;
}

/*
 * analyzer report function create
 *
 * Creates an AnalyzerReportFunction.
 *
 * [param] IP, parameters_count
 * [param] OR - the newly created AnalyzerReportFunction
 */
AnalyzerReportFunction *analyzer_report_function_create(
  int parameters_count
) {
  AnalyzerReportFunction *report_function = malloc(
    sizeof(AnalyzerReportFunction)
  );
  assert(report_function != NULL);

  report_function->identifier_dependencies = graph_create();

  report_function->identifier_reports = hashmap_create();

  report_function->process_documentation = true;

  report_function->parameter_list_report =
    analyzer_report_parameter_list_create(parameters_count);

  return report_function;
}

/*
 * analyzer_report_function_add_identifier_dependency
 *
 * Updates the identifier dependency graph with a new dependency.
 *
```

```
 * [param] IOP, report_function
 * [param] IP, parent_identifier
 * [param] IP, child_identifier
 */
void analyzer_report_function_add_identifier_dependency(
  const AnalyzerReportFunction *report_function,
  const char *parent_identifier,
  const char *child_identifier
) {
  GraphNode *parent_dependency_node, *child_dependency_node;

  parent_dependency_node = hashmap_get(
    report function->identifier dependencies->nodes,
    parent_identifier
  );

  child_dependency_node = hashmap_get(
    report_function->identifier_dependencies->nodes,
    child_identifier
  );

  graph_add_edge(parent_dependency_node, child_dependency_node);
}

/*
 * analyzer_report_function_free
 *
 * Frees the memory allocated by an AnalyzerReportFunction.
 *
 * [param] IOP, report_function
 */
void analyzer_report_function_free(
  AnalyzerReportFunction *report_function
) {
  HashMapIterator *iterator;

  for (
    iterator = hashmap_get_iterator(
      report_function->identifier_reports
    );
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    analyzer_report_identifier_free(iterator->value);
```

```
  }

  analyzer_report_parameter_list_free(
    report_function->parameter_list_report
  );

  graph_free(report_function->identifier_dependencies);

  hashmap_free(report_function->identifier_reports);

  free(report_function);
  report_function = NULL;
}

/*
 * analyzer report function register identifier
 *
 * Updates the identifier dependency graph, and the map of the
 * reports of the identifier, creating the node corresponding to a
 * new identifier.
 *
 * [param] IOP, report_function
 * [param] IP, identifier
 */
void analyzer_report_function_register_identifier(
  const AnalyzerReportFunction *report_function,
  const char *identifier
) {
  GraphNode *dependency_node = hashmap_get(
    report_function->identifier_dependencies->nodes, identifier
  );

  AnalyzerReportIdentifier *report_identifier = hashmap_get(
    report_function->identifier_reports, identifier
  );

  int *destination_flag;

  if (dependency_node == NULL) {
    graph_add_node(
      report_function->identifier_dependencies, identifier
    );
  }
```

```
  if (report_identifier == NULL) {
    report_identifier = analyzer_report_identifier_create();

    hashmap_set(
      report_function->identifier_reports,
      identifier,
      report_identifier
    );
  }

  /* Setting the IP destination flag to true by default, since  */
  /* every identifier used in the function can be considered an */
  /* input one.                                                 */

  destination_flag = hashmap_get(
    report identifier->destination map, DESTINATION IP
  );

  *destination_flag = 1;
}

/*
 * analyzer_report_function_reset_identifier
 *
 * In the identifier dependency graph, clears the dependencies of
 * the node related to an identifier.
 * In the map of the reports of the identifiers, resets each
 * destination.
 *
 * [param] IOP, report_function
 * [param] IP, identifier
 */
void analyzer_report_function_reset_identifier(
  const AnalyzerReportFunction *report_function,
  const char *identifier
) {
  GraphNode *dependency_node;

  AnalyzerReportIdentifier *report_identifier;

  HashMapIterator *iterator;

  int *destination_flag;
```

```
  /* Resetting the dependencies */

  dependency_node = hashmap_get(
    report_function->identifier_dependencies->nodes, identifier
  );

  graph_remove_edges(dependency_node);

  /* Resetting the destination map */

  report_identifier = hashmap_get(
    report_function->identifier_reports, identifier
  );

  for (
    iterator = hashmap get iterator(
      report_identifier->destination_map
    );
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    destination_flag = iterator->value;
    *destination_flag = 0;
  }
}

/********** RESERVED FUNCTIONS **********/

/*
 * arf_get_compound_statement_ast_node
 *
 * AST Exploration function. Retrieves the first compound_statement
 * among the subtree of an AST node using DFS postorder.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - the ast node
 */
GraphNode *arf_get_compound_statement_ast_node(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *roots   = list_create(1, ast_node);
  List *labels  = list_create(1, COMPOUND_STATEMENT_AST_NODE);
```

```
  GraphNode *result = graph_dfs_preorder_find_first(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return result;
}

/*
 * arf_get_identifiers_ast_node_list
 *
 * AST Exploration function. Retrieves the list of identifiers among
 * the subtree of an AST node using DFS postorder.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - list of AST nodes
 */
List *arf_get_identifiers_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
) {
  GraphNode *compound_statement_ast_node =
    arf_get_compound_statement_ast_node(ast, ast_node);

  List *roots   = list_create(1, compound_statement_ast_node);

  List *labels  = list_create(1, IDENTIFIER_AST_NODE);

  List *result = graph_dfs_postorder_find_all(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return result;
```

```
}

/*
 * arf_get_operations_ast_node_list
 *
 * AST Exploration function. Retrieves the list of operations among
 * the subtree of an AST node using DFS postorder.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - list of AST nodes
 */
List *arf get operations ast node list(
  const Graph *ast,
  const GraphNode *ast_node
) {
  GraphNode *compound_statement_ast_node =
    arf_get_compound_statement_ast_node(ast, ast_node);

  List *roots = list_create(1, compound_statement_ast_node);

  List *labels = list_create(
    3,
    INIT_DECLARATOR_AST_NODE,
    ASSIGNMENT_EXPRESSION_AST_NODE,
    FUNCTION_CALL_AST_NODE
  );

  List *result = graph_dfs_postorder_find_all(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return result;
}

/*
 * arf_get_parameters_ast_node_list
 *
 * AST Exploration function. Retrieves the list of
```

```
 * parameter_declarators among the subtree of an AST node using DFS
 * postorder. Checks whether the function parameter list is void.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - list of AST nodes
 */
List *arf_get_parameters_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *parameter_declaration_roots;
  List *parameter declaration labels;
  List *parameter_declaration_ast_node_list;

  GraphNode *parameter declaration child 1 ast node,
    *parameter_declaration_child_2_ast_node,
    *parameter_declaration_child_3_ast_node;

  char *parameter_declaration_child_1_type,
    *parameter_declaration_child_2_type,
    *parameter_declaration_child_3_type,
    *parameter_declaration_child_3_value;

  parameter_declaration_roots = list_create(1, ast_node);

  parameter_declaration_labels = list_create(
    1, PARAMETER_DECLARATION_AST_NODE
  );

  parameter_declaration_ast_node_list = graph_dfs_postorder_find_all(
    ast,
    parameter_declaration_roots,
    "type",
    parameter_declaration_labels
  );

  list_free(parameter_declaration_roots);
  list_free(parameter_declaration_labels);

  /* Checking whether the parameter declaration list is void */

  /* Checking whether there is 1 child only, of type */
  /* PARAMETER_DECLARATION                           */
```

```
if (parameter_declaration_ast_node_list->length != 1) {
  return parameter_declaration_ast_node_list;
}

parameter_declaration_child_1_ast_node = list_get(
  parameter_declaration_ast_node_list, 0
)->value;

parameter_declaration_child_1_type = analyzer_get_ast_node_type(
  parameter_declaration_child_1_ast_node
);

if (
  parameter_declaration_child_1_type
  && strcmp(
    parameter_declaration_child_1_type,
    PARAMETER_DECLARATION_AST_NODE
  ) != 0
) {
  return parameter_declaration_ast_node_list;
}

/* Checking whether there is 1 child only, of type */
/* DECLARATION_SPECIFIERS                          */

if (
  parameter_declaration_child_1_ast_node->children_edges
                                        ->length != 1
) {
  return parameter_declaration_ast_node_list;
}

parameter_declaration_child_2_ast_node = ((GraphEdge*) list_get(
  parameter_declaration_child_1_ast_node->children_edges, 0
)->value)->child_node;

parameter_declaration_child_2_type = analyzer_get_ast_node_type(
  parameter_declaration_child_2_ast_node
);

if (
  parameter_declaration_child_2_type
  && strcmp(
```

```
    parameter_declaration_child_2_type,
    DECLARATION_SPECIFIERS_AST_NODE
  ) != 0
) {
  return parameter_declaration_ast_node_list;
}

/* Checking whether there is 1 child only, of type */
/* TYPE_SPECIFIER, and value VOID                  */

if (
  parameter_declaration_child_2_ast_node->children_edges
                                        ->length != 1
) {
  return parameter_declaration_ast_node_list;
}

parameter_declaration_child_3_ast_node = ((GraphEdge*) list_get(
  parameter_declaration_child_2_ast_node->children_edges, 0
)->value)->child_node;

parameter_declaration_child_3_type = analyzer_get_ast_node_type(
  parameter_declaration_child_3_ast_node
);

parameter_declaration_child_3_value = analyzer_get_ast_node_value(
  parameter_declaration_child_3_ast_node
);

if (
  (
    parameter_declaration_child_3_type
    && strcmp(
      parameter_declaration_child_3_type, TYPE_SPECIFIER_AST_NODE
    ) != 0
  )
  || (
    parameter_declaration_child_3_value
    && strcmp(parameter_declaration_child_3_value, "VOID") != 0
  )
) {
  return parameter_declaration_ast_node_list;
}
```

```
  list_free(parameter_declaration_ast_node_list);
  parameter_declaration_ast_node_list = list_create_empty();

  return parameter_declaration_ast_node_list;
}

/*
 * arf_check_non_void_return_type
 *
 * Checks whether the return type of a function is non-void.
 *
 * [param] ast
 * [param] ast node
 * [param] OR - the resulting boolean value
 */
bool arf check non void return type(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *declaration_roots, *declarator_roots;
  List *declaration_labels, *declarator_labels;

  GraphNode *declaration_ast_node, *declaration_child_1_ast_node;
  GraphNode *declarator_ast_node, *declarator_child_1_ast_node;

  char *declaration_child_1_type, *declaration_child_1_value;
  char *declarator_child_1_type;

  declaration_roots = list_create(1, ast_node);

  declaration_labels = list_create(
    1, DECLARATION_SPECIFIERS_AST_NODE
  );

  declaration_ast_node = graph_dfs_preorder_find_first(
    ast, declaration_roots, "type", declaration_labels
  );

  list_free(declaration_roots);
  list_free(declaration_labels);

  declarator_roots = list_create(1, ast_node);

  declarator_labels = list_create(1, DECLARATOR_AST_NODE);
```

```
declarator_ast_node = graph_dfs_preorder_find_first(
  ast, declarator_roots, "type", declarator_labels
);

list_free(declarator_roots);
list_free(declarator_labels);

/* Checking whether the DECLARATION has 1 child only, of type */
/* TYPE_SPECIFIER and value different from VOID               */

if (declaration_ast_node->children_edges->length != 1) {
  return true;
}

declaration child 1 ast node = ((GraphEdge*) list get(
  declaration_ast_node->children_edges, 0
)->value)->child_node;

declaration_child_1_type = analyzer_get_ast_node_type(
  declaration_child_1_ast_node
);

declaration_child_1_value = analyzer_get_ast_node_value(
  declaration_child_1_ast_node
);

if (
  (
    declaration_child_1_type
    && strcmp(
      declaration_child_1_type, TYPE_SPECIFIER_AST_NODE
    ) != 0
  )
  || declaration_child_1_value == NULL
  || strcmp(declaration_child_1_value, "VOID") != 0
) {
  return true;
}

/* Checking whether the DECLARATOR the first child of */
/* type POINTER                                       */

if (list_is_empty(declarator_ast_node->children_edges)) {
```

```
    return true;
  }

  declarator_child_1_ast_node = ((GraphEdge*) list_get(
    declarator_ast_node->children_edges, 0
  )->value)->child_node;

  declarator_child_1_type = analyzer_get_ast_node_type(
    declarator_child_1_ast_node
  );

  if (
    declarator child 1 type
    && strcmp(declarator_child_1_type, POINTER_AST_NODE) == 0
  ) {
    return true;
  }

  return false;
}

/*
 * arf_parameter_list_compute
 *
 * Computes the parameters list of the function.
 *
 * [param] IOP, report_function
 */
void arf_parameter_list_compute(
  const AnalyzerReportFunction *report_function
) {
  HashMapIterator *iterator_a, *iterator_b;

  AnalyzerReportIdentifier *report_identifier;

  AnalyzerReportParameter *report_parameter;

  int *destination_flag;

  for (
    iterator_a = hashmap_get_iterator(
      report_function->identifier_reports
    );
    iterator_a != NULL;
```

```
    iterator_a = hashmap_iterate(iterator_a)
  ) {
    report_identifier = iterator_a->value;

    report_parameter = hashmap_get(
      report_function->parameter_list_report->parameter_reports_map,
      iterator_a->key
    );

    if (report_parameter != NULL) {
      for (
        iterator_b = hashmap_get_iterator(
          report identifier->destination map
        );
        iterator_b != NULL;
        iterator b = hashmap iterate(iterator b)
      ) {
        if (*((int *) iterator_b->value) == 1) {
          destination_flag = hashmap_get(
            report_parameter->destination_map, iterator_b->key
          );
          *destination_flag = 1;
        }
      }
    }
  }
}

/*
 * arf_parameter_list_report_initialize
 *
 * Initializes the report of the parameters list of a function.
 *
 * [param] IP, ast
 * [param] IP, function_ast_node
 * [param] IP, parameter_ast_node_list
 * [param] IOP, report_function
 */
void arf_parameter_list_report_initialize(
  const Graph *ast,
  const GraphNode *function_ast_node,
  const List *parameter_ast_node_list,
  const AnalyzerReportFunction *report_function
) {
```

```
  ListNode *iterator;

  char *parameter_identifier;

  int parameter_index = 0;

  if (arf_check_non_void_return_type(ast, function_ast_node)) {
    report_function->parameter_list_report->destination_or = true;
  }

  for(
    iterator = list_get_iterator(parameter_ast_node_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    parameter identifier = analyzer get identifier ast node value(
      ast, iterator->value
    );

    hashmap_set(
      report_function->parameter_list_report->parameter_reports_map,
      parameter_identifier,
      report_function->parameter_list_report
                     ->parameter_reports[parameter_index]
    );

    parameter_index += 1;
  }
}
```

8.4.6 mod_analyzer_report_function_operation_assignment

8.4.6.1 mod_analyzer_report_function_operation_assignment.h

```
#include "mod_analyzer_report_function.h"

void analyzer_report_function_operation_assignment_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportFunction *report_function
);

void analyzer_report_function_operation_declaration_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportFunction *report_function
);
```

8.4.6.2 mod_analyzer_report_function_operation_assignment.c

```
#include <string.h>

#include "mod_analyzer_common.h"
#include "mod_analyzer_destination_map.h"
#include "mod_analyzer_report_function_operation_assignment.h"
#include "mod_analyzer_report_identifier.h"
#include "mod_parser.h"

/********** RESERVED FUNCTION DECLARATIONS **********/

void arfoa_compute_left_right(
  const Graph *ast,
  const GraphNode *left_side_ast_node,
  const GraphNode *right_side_ast_node,
  const AnalyzerReportFunction *report_function,
  bool reset_left_side_identifier
);
```

```
void arfoa_compute_left_right_heredity(
  const AnalyzerReportFunction *report_function,
  const AnalyzerReportIdentifier *report_left_side_identifier,
  const char *right_side_identifier
);

/********** LIBRARY FUNCTIONS **********/

/*
 * analyzer_report_function_operation_assignment_compute
 *
 * Updates an AnalyzerReportFunction after and assignment operation.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] IOP, report function
 */
void analyzer_report_function_operation_assignment_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportFunction *report_function
) {
  GraphNode *right_side_ast_node = ((GraphEdge*) list_get(
    ast_node->children_edges, 0
  )->value)->child_node;

  GraphNode *left_side_ast_node = ((GraphEdge*) list_get(
    ast_node->children_edges, 1
  )->value)->child_node;

  GraphNode *operator_ast_node = ((GraphEdge*) list_get(
    ast_node->children_edges, 2
  )->value)->child_node;

  char *operator_value = analyzer_get_ast_node_value(
    operator_ast_node
  );

  bool array_access   = analyzer_check_array_access(
    ast, left_side_ast_node
  );

  bool member_access  = analyzer_check_member_access(
    ast, left_side_ast_node
```

```
  );

  bool pointer_access = analyzer_check_pointer_access(
    ast, left_side_ast_node
  );

  bool simple_assign = strcmp(operator_value, "SIMPLE_ASSIGN") == 0;

  arfoa_compute_left_right(
    ast,
    left_side_ast_node,
    right_side_ast_node,
    report function,
    !array_access
    && !member_access
    && !pointer access
    && simple_assign
  );
}


/*
 * analyzer_report_function_operation_declaration_compute
 *
 * Updates an AnalyzerReportFunction after a declaration operation.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] IOP, report_function
 */
void analyzer_report_function_operation_declaration_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportFunction *report_function
) {
  List *left_side_roots   = list_create(1, ast_node);
  List *left_side_labels  = list_create(1, DECLARATOR_AST_NODE);
  List *right_side_roots  = list_create(1, ast_node);
  List *right_side_labels = list_create(1, INITIALIZER_AST_NODE);

  GraphNode *left_side_ast_node   = graph_dfs_preorder_find_first(
    ast, left_side_roots, "type", left_side_labels);
  GraphNode *right_side_ast_node  = graph_dfs_preorder_find_first(
    ast, right_side_roots, "type", right_side_labels);
```

```
  arfoa_compute_left_right(
    ast,
    left_side_ast_node,
    right_side_ast_node,
    report_function,
    true
  );

  /* Clean-up */

  list_free(left_side_roots);
  list free(left side labels);
  list_free(right_side_roots);
  list_free(right_side_labels);
}

/********** RESERVED FUNCTIONS **********/

/*
 * arfoa_compute_left_right
 *
 * Computes a left-right expression.
 *
 * [param] IP, ast
 * [param] IP, left_side_ast_node
 * [param] IP, right_side_ast_node
 * [param] IOP, report_function
 * [param] IP, reset_left_side_identifier
 */
void arfoa_compute_left_right(
  const Graph *ast,
  const GraphNode *left_side_ast_node,
  const GraphNode *right_side_ast_node,
  const AnalyzerReportFunction *report_function,
  bool reset_left_side_identifier
) {
  List *right_side_terms_ast_node_list;

  ListNode *iterator;

  AnalyzerReportIdentifier *left_side_report_identifier;

  char *left_side_identifier, *right_side_term_identifier;
```

```
int *destination_flag;

/* Retrieving the left side identifier */

left_side_identifier = analyzer_get_identifier_ast_node_value(
  ast, left_side_ast_node
);

/* Retrieving the report of the left side identifier */

left_side_report_identifier = hashmap_get(
  report function->identifier reports, left side identifier
);

/* Flushing the left side dependencies if needed */

if (reset_left_side_identifier) {
  analyzer_report_function_reset_identifier(
    report_function, left_side_identifier
  );
}

/* Adding destination OP to the left side identifier */

destination_flag = hashmap_get(
  left_side_report_identifier->destination_map, DESTINATION_OP
);

*destination_flag = 1;

/* Retrieving the right side list of terms */

if (right_side_ast_node == NULL) {
  return;
}

right_side_terms_ast_node_list =
  analyzer_get_identifier_ast_node_list(ast, right_side_ast_node);

for (
  iterator = list_get_iterator(right_side_terms_ast_node_list);
  iterator != NULL;
  iterator = list_iterate(iterator)
```

```
  ) {
    right_side_term_identifier =
      analyzer_get_identifier_ast_node_value(ast, iterator->value);

    /* Creating a dependency edge between the left side identifier */
    /* and the one of the current right side term                  */

    analyzer_report_function_add_identifier_dependency(
      report_function,
      left_side_identifier,
      right_side_term_identifier
    );

    /* Computing the heredity */

    arfoa compute left right heredity(
      report_function,
      left_side_report_identifier,
      right_side_term_identifier
    );
  }

  /* Clean-up */

  list_free(right_side_terms_ast_node_list);
}

/*
 * arfoa_compute_left_right_heredity
 *
 * Computes the destination heredity in a left-right expression.
 *
 * [param] IOP, report_function
 * [param] IOP, left_side_report_identifier
 * [param] IP, right_side_identifier
 */
void arfoa_compute_left_right_heredity(
  const AnalyzerReportFunction *report_function,
  const AnalyzerReportIdentifier *report_left_side_identifier,
  const char *right_side_identifier
) {
  AnalyzerReportIdentifier *right_side_report_identifier;

  right_side_report_identifier = hashmap_get(
```

```
    report_function->identifier_reports, right_side_identifier
  );

  analyzer_report_identifier_compute_heredity(
    report_left_side_identifier, right_side_report_identifier
  );
}
```

8.4.7 mod_analyzer_report_function_operation_call

8.4.7.1 mod_analyzer_report_function_operation_call.h

```
#include "mod_analyzer_report_function.h"
#include "mod_analyzer_report_program.h"

void analyzer_report_function_operation_call_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportProgram *report_program,
  const AnalyzerReportFunction *report_function
);
```

8.4.7.2 mod_analyzer_report_function_operation_call.c

```
#include <string.h>

#include "mod_analyzer_common.h"
#include "mod_analyzer_destination_map.h"
#include "mod_analyzer_report_function.h"
#include "mod_analyzer_report_function_operation_call.h"
#include "mod_analyzer_report_program.h"
#include "mod_parser.h"

/********** RESERVED FUNCTION DECLARATIONS **********/

GraphNode *arfoc_get_argument_list_ast_node(
  const Graph *ast,
  const GraphNode *ast_node
);

List *arfoc_call_argument_get_term_dependencies(
  const AnalyzerReportFunction *report_function,
  const char *term_identifier
);

List *arfoc_get_argument_unary_ast_node_list(
```

```
  const Graph *ast,
  const GraphNode *ast_node
);

void arfoc_operation_call_argument_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportFunction *report_function,
  const AnalyzerReportParameter *report_parameter
);

void arfoc_operation_call_argument_heredity_compute(
  const AnalyzerReportFunction *report function,
  const AnalyzerReportParameter *report_parameter,
  const char *term_identifier,
  bool reset term identifier
);

/********** LIBRARY FUNCTIONS **********/

/*
 * analyzer_report_function_operation_call_compute
 *
 * Computes a function call operation.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] IOP, report_program
 * [param] IOP, report_function
 */
void analyzer_report_function_operation_call_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportProgram *report_program,
  const AnalyzerReportFunction *report_function
) {
  char *call_identifier;

  AnalyzerReportFunction *report_function_call;

  AnalyzerReportParameter *report_parameter;

  GraphNode *argument_list_ast_node, *argument_ast_node;
```

```
  ListNode *iterator;

  int argument_index = 0;

  /* Retrieving the identifier of the called function */

  call_identifier = analyzer_get_identifier_ast_node_value(
    ast, ast_node
  );

  /* Retrieving the report of the called function */

  report function call = hashmap get(
    report_program->function_reports, call_identifier
  );

  if (report_function_call == NULL) {
    return;
  }

  /* Retrieving the list of arguments of the function */

  argument_list_ast_node = arfoc_get_argument_list_ast_node(
    ast, ast_node
  );

  if (argument_list_ast_node == NULL) {
    return;
  }

  for (
    iterator = list_get_iterator(
      argument_list_ast_node->children_edges
    );
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    argument_ast_node = ((GraphEdge*) iterator->value)->child_node;

    report_parameter =
      argument_index < report_function_call->parameter_list_report
                                            ->parameters_count
      ? report_function_call->parameter_list_report
                            ->parameter_reports[argument_index]
```

```
      : report_function_call->parameter_list_report
                            ->vararg_report;

    arfoc_operation_call_argument_compute(
      ast,
      argument_ast_node,
      report_function,
      report_parameter
    );

    argument_index += 1;
  }
}

/********** RESERVED FUNCTIONS **********/

/*
 * arfoc_get_argument_list_ast_node
 *
 * AST Exploration function. Retrieves the first
 * argument_expression_list among the subtree of an AST node using
 * DFS postorder.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - the AST node
 */
GraphNode *arfoc_get_argument_list_ast_node(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *roots   = list_create(1, ast_node);
  List *labels  = list_create(1, ARGUMENT_EXPRESSION_LIST_AST_NODE);

  GraphNode *result = graph_dfs_preorder_find_first(
    ast, roots, "type", labels
  );

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return result;
```

```
}

/*
 * arfoc_call_argument_get_term_dependencies
 *
 * Retrieves the graph node in the identifier dependency graph
 * related to a function term.
 *
 * [param] IP, report_function
 * [param] IP, term_identifier
 * [param] OR
 */
List *arfoc call argument get term dependencies(
  const AnalyzerReportFunction *report_function,
  const char *term_identifier
) {
  List *term_dependencies;

  List *term_dependency_graph_node = hashmap_get(
    report_function->identifier_dependencies->nodes,
    term_identifier
  );

  List *roots = list_create(1, term_dependency_graph_node);

  term_dependencies = graph_bfs(
    report_function->identifier_dependencies,
    roots
  );

  /* Clear-up */

  list_free(roots);

  return term_dependencies;
}

/*
 * arfoc_get_argument_unary_ast_node_list
 *
 * AST Exploration function. Retrieves the list of the top-level
 * unary_expressions among the subtree of an AST node using DFS
 * preorder.
 *
```

```
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] OR - the list of AST nodes
 */
List *arfoc_get_argument_unary_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
) {
  ListNode *iterator;

  List *identifier_ast_node_list;

  List *result = list create empty();

  List *roots   = list_create(1, ast_node);
  List *labels  = list create(1, UNARY EXPRESSION AST NODE);

  List *unary_ast_node_list = graph_dfs_preorder_find(
    ast, roots, "type", labels
  );

  for (
    iterator = list_get_iterator(unary_ast_node_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    identifier_ast_node_list = analyzer_get_identifier_ast_node_list(
      ast, iterator->value
    );

    if (identifier_ast_node_list->length > 0) {
      list_push_back(result, iterator->value);
    }

    /* Clean-up */

    list_free(identifier_ast_node_list);
  }

  /* Clean-up */

  list_free(roots);
  list_free(labels);
  list_free(unary_ast_node_list);
```

```
  return result;
}

/*
 * arfoc_operation_call_argument_compute
 *
 * Computes an argument in a function call operation.
 *
 * [param] IP, ast
 * [param] IP, ast_node
 * [param] IP, report_function
 * [param] IP, report parameter
 */
void arfoc_operation_call_argument_compute(
  const Graph *ast,
  const GraphNode *ast_node,
  const AnalyzerReportFunction *report_function,
  const AnalyzerReportParameter *report_parameter
) {
  ListNode *iterator;

  char *term_identifier;

  bool array_access, member_access, pointer_access, *assignation;

  /* Retrieving the list of identifiers within the argument */

  List *term_ast_node_list = arfoc_get_argument_unary_ast_node_list(
    ast, ast_node);

  for (
    iterator = list_get_iterator(term_ast_node_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    term_identifier = analyzer_get_identifier_ast_node_value(
      ast, iterator->value
    );

    array_access = analyzer_check_array_access(
      ast, iterator->value
    );
```

```
    member_access = analyzer_check_member_access(
      ast, iterator->value
    );

    pointer_access = analyzer_check_pointer_access(
      ast, iterator->value
    );

    assignation = hashmap_get(
      report_parameter->destination_map, DESTINATION_OP
    );

    arfoc operation call argument heredity compute(
      report_function,
      report_parameter,
      term identifier,
      !array_access
      && !member_access
      && !pointer_access
      && *assignation
    );
  }

  /* Clean-up */

  list_free(term_ast_node_list);
}

/*
 * arfoc_operation_call_argument_heredity_compute
 *
 * Computes the heredity of an argument in a function call operation.
 *
 * [param] IOP, report_function
 * [param] IOP, report_parameter
 * [param] IP, term_identifier
 * [param] IP, reset_term_identifier
 */
void arfoc_operation_call_argument_heredity_compute(
  const AnalyzerReportFunction *report_function,
  const AnalyzerReportParameter *report_parameter,
  const char *term_identifier,
  bool reset_term_identifier
) {
```

```
  ListNode *iterator;

  List *term_dependencies;

  AnalyzerReportIdentifier *report_identifier;

  int *term_destination_flag,
    *parameter_destination_flag_op,
    *parameter_destination_flag_ov,
    *parameter_destination_flag_of,
    *parameter_destination_flag_ik,
    *parameter_destination_flag_if;

  /* Retrieving the list of identifiers depending on the current */
  /* one.                                                        */

  if (reset_term_identifier) {
    analyzer_report_function_reset_identifier(
      report_function, term_identifier
    );
  }

  term_dependencies = arfoc_call_argument_get_term_dependencies(
    report_function,
    term_identifier
  );

  /* Computing the destination flag OP */

  parameter_destination_flag_op = hashmap_get(
    report_parameter->destination_map, DESTINATION_OP
  );

  parameter_destination_flag_ov = hashmap_get(
    report_parameter->destination_map, DESTINATION_OV
  );

  parameter_destination_flag_of = hashmap_get(
    report_parameter->destination_map, DESTINATION_OF
  );

  parameter_destination_flag_ik = hashmap_get(
    report_parameter->destination_map, DESTINATION_IK
  );
```

```
parameter_destination_flag_if = hashmap_get(
  report_parameter->destination_map, DESTINATION_IF
);

for (
  iterator = list_get_iterator(term_dependencies);
  iterator != NULL;
  iterator = list_iterate(iterator)
) {
  report_identifier = hashmap_get(
    report_function->identifier_reports,
    ((GraphNode*) iterator->value)->identifier
  );

  if (
    parameter_destination_flag_op
    && *parameter_destination_flag_op == 1
  ) {
    term_destination_flag = hashmap_get(
      report_identifier->destination_map, DESTINATION_OP
    );

    *term_destination_flag = 1;
  }

  if (
    parameter_destination_flag_ov
    && *parameter_destination_flag_ov == 1
  ) {
    term_destination_flag = hashmap_get(
      report_identifier->destination_map, DESTINATION_OV
    );

    *term_destination_flag = 1;
  }

  if (
    parameter_destination_flag_of
    && *parameter_destination_flag_of == 1
  ) {
    term_destination_flag = hashmap_get(
      report_identifier->destination_map, DESTINATION_OF
    );
```

```
      *term_destination_flag = 1;
    }

    if (
      parameter_destination_flag_ik
      && *parameter_destination_flag_ik == 1
    ) {
      term_destination_flag = hashmap_get(
        report_identifier->destination_map, DESTINATION_IK
      );

      *term destination flag = 1;
    }

    if (
      parameter_destination_flag_if
      && *parameter_destination_flag_if == 1
    ) {
      term_destination_flag = hashmap_get(
        report_identifier->destination_map, DESTINATION_IF
      );

      *term_destination_flag = 1;
    }
  }

  /* Clean-up */

  list_free(term_dependencies);
}
```

8.4.8 mod_analyzer_report_identifier

8.4.8.1 mod_analyzer_report_identifier.h

```
#include "lib_hashmap.h"

#ifndef MOD_ANALYZER_REPORT_IDENTIFIER
#define MOD_ANALYZER_REPORT_IDENTIFIER

typedef struct AnalyzerReportIdentifier {
  HashMap *destination_map;
} AnalyzerReportIdentifier;

#endif

AnalyzerReportIdentifier* analyzer_report_identifier_create();

void analyzer_report_identifier_compute_heredity(
  const AnalyzerReportIdentifier *parent,
  const AnalyzerReportIdentifier *child
);

void analyzer_report_identifier_free(
  AnalyzerReportIdentifier *report_identifier
);
```

8.4.8.2 mod_analyzer_report_identifier.c

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#include "mod_analyzer_destination_map.h"
#include "mod_analyzer_report_identifier.h"

/********** LIBRARY FUNCTIONS **********/

/*
```

```
 * analyzer_report_identifier_create
 *
 * Creates a new AnalyzerReportIdentifier.
 *
 * [param] OR - the newly created IdentifierReport
 */
AnalyzerReportIdentifier *analyzer_report_identifier_create(void) {
  AnalyzerReportIdentifier *report_identifier = malloc(
    sizeof(AnalyzerReportIdentifier)
  );

  assert(report_identifier != NULL);

  report_identifier->destination_map =
    analyzer_destination_map_create();

  return report_identifier;
}

/*
 * analyzer_report_identifier_compute_heredity
 *
 * Computes the heredity of an identifier. The IK and IF destinations
 * only are inherited from parent to child.
 *
 * [param] IOP, parent
 * [param] IOP, child
 */
void analyzer_report_identifier_compute_heredity(
  const AnalyzerReportIdentifier *parent,
  const AnalyzerReportIdentifier *child
) {
  int *parent_destination, *child_destination;

  parent_destination = hashmap_get(
    child->destination_map, DESTINATION_IK
  );

  if (parent_destination != NULL && *parent_destination == 1) {
    child_destination = hashmap_get(
      parent->destination_map, DESTINATION_IK
    );

    *child_destination = 1;
```

```
  }

  parent_destination = hashmap_get(
    child->destination_map, DESTINATION_IF
  );

  if (parent_destination != NULL && *parent_destination) {
    child_destination = hashmap_get(
      parent->destination_map, DESTINATION_IF
    );

    *child_destination = 1;
  }
}

/*
 * analyzer_report_identifier_free
 *
 * Frees the memory allocated by an AnalyzerReportIdentifier.
 *
 * [param] IOP, report_identifier
 */
void analyzer_report_identifier_free(
  AnalyzerReportIdentifier *report_identifier
) {
  analyzer_destination_map_free(report_identifier->destination_map);

  free(report_identifier);
  report_identifier = NULL;
}
```

8.4.9 mod_analyzer_report_parameter

8.4.9.1 mod_analyzer_report_parameter.h

```
#include "lib_hashmap.h"

#ifndef MOD_ANALYZER_REPORT_PARAMETER
#define MOD_ANALYZER_REPORT_PARAMETER

typedef struct AnalyzerReportParameter {
  HashMap *destination_map;
} AnalyzerReportParameter;

typedef struct AnalyzerReportParameterList {
  int parameters_count;
  AnalyzerReportParameter **parameter_reports;
  AnalyzerReportParameter *vararg_report;
  HashMap *parameter_reports_map;
  bool destination_or;
} AnalyzerReportParameterList;

#endif

AnalyzerReportParameter* analyzer report parameter create();

AnalyzerReportParameterList* analyzer_report_parameter_list_create(
  int parameters_count
);

void analyzer_report_parameter_list_free(
  AnalyzerReportParameterList *report_parameter_list
);
```

8.4.9.2 mod_analyzer_report_parameter.c

```
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
```

```
#include "mod_analyzer_common.h"
#include "mod_analyzer_destination_map.h"
#include "mod_analyzer_report_function.h"
#include "mod_analyzer_report_parameter.h"

/********** RESERVED FUNCTION DECLARATIONS **********/

void arp_free(AnalyzerReportParameter *report_parameter);

/********** LIBRARY FUNCTIONS **********/

/*
 * analyzer_report_parameter_create
 *
 * Creates a new AnalyzerReportParameter.
 *
 * [param] OR - the newly created AnalyzerReportParameter
 */
AnalyzerReportParameter *analyzer_report_parameter_create(void) {
  AnalyzerReportParameter *report_parameter;

  report_parameter = malloc(sizeof(AnalyzerReportParameter));
  assert(report_parameter != NULL);

  report_parameter->destination_map =
    analyzer_destination_map_create();

  return report_parameter;
}

/*
 * analyzer_report_parameter_list_create
 *
 * Creates a new AnalyzerReportParameterList.
 *
 * [param] IP, parameters_count
 * [param] OR - the newly created AnalyzerReportParameter.
 */
AnalyzerReportParameterList *analyzer_report_parameter_list_create(
  int parameters_count
) {
  int i;
```

```
  AnalyzerReportParameterList *report_parameter_list =
    malloc(sizeof(AnalyzerReportParameterList));

  assert(report_parameter_list != NULL);

  report_parameter_list->parameters_count = parameters_count;
  report_parameter_list->parameter_reports_map = hashmap_create();

  report_parameter_list->parameter_reports = malloc(
    parameters_count * sizeof(AnalyzerReportParameter)
  );

  assert(report parameter list->parameter reports != NULL);

  report_parameter_list->destination_or = false;

  for (i = 0; i < parameters_count; i += 1) {
    report_parameter_list->parameter_reports[i] =
      analyzer_report_parameter_create();
  }

  report_parameter_list->vararg_report =
    analyzer_report_parameter_create();

  return report_parameter_list;
}

/*
 * analyzer_report_parameter_list_free
 *
 * Frees the memory allocated by an AnalyzerReportParameterList.
 *
 * [param] IOP, report_parameter_list
 */
void analyzer_report_parameter_list_free(
  AnalyzerReportParameterList *report_parameter_list
) {
  int index;

  for (
    index = 0;
    index < report_parameter_list->parameters_count;
    index += 1
  ) {
```

```
    arp_free(
      report_parameter_list->parameter_reports[index]
    );
  }

  arp_free(
    report_parameter_list->vararg_report
  );

  hashmap_free(report_parameter_list->parameter_reports_map);

  free(report_parameter_list->parameter_reports);
  report parameter list->parameter reports = NULL;

  free(report_parameter_list);
  report parameter list = NULL;
}

/********** RESERVED FUNCTIONS **********/

/*
 * analyzer_report_parameter_free
 *
 * Frees the memory allocated by an AnalyzerReportParameter.
 *
 * [param] IOP, report_parameter
 */
void arp_free(
  AnalyzerReportParameter *report_parameter
) {
  analyzer_destination_map_free(report_parameter->destination_map);

  free(report_parameter);
  report_parameter = NULL;
}
```

8.4.10 mod_analyzer_report_program

8.4.10.1 mod_analyzer_report_program.h

```
#include "lib_graph.h"

#ifndef MOD_ANALYZER_REPORT_PROGRAM
#define MOD_ANALYZER_REPORT_PROGRAM

typedef struct AnalyzerReportProgram {
  Graph *function_dependencies;
  HashMap *function_reports;
  HashMap *documentation_reports;
} AnalyzerReportProgram;

#endif

AnalyzerReportProgram *analyzer_report_program_create(
  const Graph *ast
);

void analyzer_report_program_free(
  AnalyzerReportProgram *report_program
);
```

8.4.10.2 mod_analyzer_report_program.c

```
#include <stdlib.h>
#include <assert.h>

#include "mod_analyzer_common.h"
#include "mod_analyzer_destination_map.h"
#include "mod_analyzer_report_documentation.h"
#include "mod_analyzer_report_function.h"
#include "mod_analyzer_report_program.h"
#include "mod_parser.h"

#define AST_NODE_IDENTIFIER_KEY "AST_NODE_IDENTIFIER"
```

```
#define ERROR_MESSAGE_NO_DOC_FOUND \
  "[WARNING] Function '%s': no documentation found.\n"

#define ERROR_MESSAGE_DESTINATION_UNNECESSARY \
  "[WARNING] Function '%s': destination %s for parameter '%s' " \
  "unnecessary.\n"

#define ERROR_MESSAGE_DESTINATION_NOT_FOUND \
  "[WARNING] Function '%s': destination %s for parameter '%s' not " \
  "found.\n"

#define ERROR MESSAGE DESTINATION OR UNNECESSARY \
  "[WARNING] Function '%s': destination OR unnecessary.\n"

#define ERROR MESSAGE DESTINATION OR NOT FOUND \
  "[WARNING] Function '%s': destination OR not found.\n"

#define ERROR_MESSAGE_PARAMETER_UNKNOWN \
  "[WARNING] Function '%s': documentation found for unknown " \
  "parameter with identifier '%s'.\n"

#define ERROR_MESSAGE_PARAMETER_NOT_FOUND \
  "[WARNING] Function '%s': no documentation found for parameter " \
  "with identifier '%s'.\n"

#define ERROR_MESSAGE_RECURSIVE_CALL_CHAIN \
  "[ERROR] Recursive call chains detected.\n"

#define SUCCESS_MESSAGE \
  "[LOG] Success, no warnings found.\n"

/********** RESERVED FUNCTION DECLARATIONS **********/

/* AST EXPLORATION FUNCTIONS */

List *arp_get_function_ast_node_list(const Graph *ast);

List *arp_get_function_ast_node_list_ordered(
  const AnalyzerReportProgram *report_program
);

List *arp_get_function_call_ast_node_list(
  const Graph *ast,
```

```
  const GraphNode *ast_node
);

/* EVALUATION OF THE REPORTS OF FUNCTIONS AND DOCUMENTATIONS */

void arp_evaluate_documentation_reports(
  const AnalyzerReportProgram *report_program
);

bool arp_evaluate_documentation_report(
  const AnalyzerReportDocumentation *report_documentation,
  const AnalyzerReportFunction *report_function,
  const char *function identifier
);

bool arp evaluate documentation report destinations(
  const HashMap *documentation_param_destination_map,
  const HashMap *function_param_destination_map,
  const char *function_identifier,
  const char *parameter_identifier
);

bool arp_evaluate_documentation_report_destination(
  const HashMap *documentation_parameter_destination_map,
  const HashMap *function_parameter_destination_map,
  const char *function_identifier,
  const char *parameter_identifier,
  const char *destination
);

/* EVALUATION OF THE DEPENDENCIES OF THE FUNCTIONS */

void arp_function_dependencies_compute(
  const Graph *ast,
  const AnalyzerReportProgram *report_program
);

void arp_function_dependency_compute(
  const Graph *ast,
  const GraphNode *function_ast_node,
  const AnalyzerReportProgram *report_program
);

void arp_function_dependency_register(
```

```
  const Graph *ast,
  const GraphNode *function_ast_node,
  const AnalyzerReportProgram *report_program
);

void arp_function_check_recursive_call_chain_presence(
  const AnalyzerReportProgram *report_program
);

/* REGISTERING FUNCTIONS */

void arp_register_defaults(
  const AnalyzerReportProgram *report program
);

void arp register documentation(
  const Graph *ast,
  const AnalyzerReportProgram *report_program
);

/********** LIBRARY FUNCTIONS **********/

/**
 * analyzer_report_program_create
 *
 * Creates a new AnalyzerReportProgram.
 *
 * [param] IP, ast
 * [param] OR - the newly created AnalyzerReportProgram
 */
AnalyzerReportProgram* analyzer_report_program_create(
  const Graph *ast
) {
  List *function_dependency_ordered_list;

  ListNode *iterator;

  char *function_identifier, *function_ast_node_identifier;

  GraphNode *function_ast_node;

  AnalyzerReportProgram *report_program
    = malloc(sizeof(AnalyzerReportProgram));
```

```
  assert(report_program != NULL);

  report_program->function_dependencies = graph_create();
  report_program->function_reports = hashmap_create();
  report_program->documentation_reports = hashmap_create();

  /* Registering the default function reports */

  arp_register_defaults(report_program);

  /* Computing the dependencies of the functions */

  arp function dependencies compute(ast, report program);

  /* Checking whether the program has recursive call chains */

  arp_function_check_recursive_call_chain_presence(report_program);

  /* Ordering the list of the function definitions */

  function_dependency_ordered_list =
    arp_get_function_ast_node_list_ordered(report_program);

  /* Creating a ReportFunction for each function definition */

  for (
    iterator = list_get_iterator(function_dependency_ordered_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    function_identifier = ((GraphNode*) iterator->value)->identifier;

    function_ast_node_identifier = graph_node_get_field(
      iterator->value, AST_NODE_IDENTIFIER_KEY
    );

    function_ast_node = hashmap_get(
      ast->nodes, function_ast_node_identifier
    );

    printf("[LOG] Evaluating function %s.\n", function_identifier);

    hashmap_set(
      report_program->function_reports,
```

```
      function_identifier,
      analyzer_report_function_compute(
        ast, function_ast_node, report_program
      )
    );
  }

  /* Registering the documentation */

  arp_register_documentation(ast, report_program);

  /* Evaluating the documentation */

  arp_evaluate_documentation_reports(report_program);

  /* */

  /* Clean-up */

  list_free(function_dependency_ordered_list);

  return report_program;
}

/**
 * analyzer_report_program_free
 *
 * Frees the memory associated to the AnalyzerReportProgram.
 *
 * [param] IOP, report_program
 */
void analyzer_report_program_free(
  AnalyzerReportProgram *report_program
) {
  HashMapIterator *iterator;

  for (
    iterator = hashmap_get_iterator(
      report_program->function_reports
    );
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    analyzer_report_function_free(iterator->value);
```

```
  }

  for (
    iterator = hashmap_get_iterator(
      report_program->documentation_reports
    );
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    analyzer_report_documentation_free(iterator->value);
  }

  graph free(report program->function dependencies);
  hashmap_free(report_program->function_reports);
  hashmap_free(report_program->documentation_reports);

  free(report_program);
  report_program = NULL;
}

/********** RESERVED FUNCTIONS **********/

/* AST EXPLORATION FUNCTIONS */

/*
 * arp_get_function_ast_node_list
 *
 * AST Exploration function. Retrieves the list of the top-level
 * function_definitions among the subtree of an AST node using DFS
 * preorder.
 *
 * [param] IP, ast
 * [param] OR - the list of AST nodes
 */
List *arp_get_function_ast_node_list(const Graph *ast) {
  List *roots   = list_create(
    1, hashmap_get(ast->nodes, PROGRAM_AST_NODE)
  );

  List *labels  = list_create(
    1, FUNCTION_DEFINITION_AST_NODE
  );

  List *result = graph_dfs_preorder_find(ast, roots, "type", labels);
```

```
  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return result;
}

/*
 * arp_get_function_ast_node_list_ordered
 *
 * Computes the list of AST nodes of functions in the correct order
 * for the evaluation.
 *
 * [param] IP, report program
 * [param] OR - list of AST nodes
 */
List *arp_get_function_ast_node_list_ordered(
  const AnalyzerReportProgram *report_program
) {
  HashMapIterator *iterator;

  GraphNode *root;

  List *result, *roots;

  if (report_program->function_dependencies->nodes->length == 0) {
    printf("[WARNING] No functions have been defined.\n");
    exit(-1);
  }

  /* At this point we are dealing with a directed acyclic and */
  /* eventually not connected graph                           */

  iterator = hashmap_get_iterator(
    report_program->function_dependencies->nodes
  );

  root = iterator->value;

  roots = list_create(1, root);

  result = graph_dfs_postorder(
```

```
    report_program->function_dependencies, roots
  );

  /* Clean-up */

  list_free(roots);

  free(iterator);
  iterator = NULL;

  return result;
}

/*
 * arp_get_function_call_ast_node_list
 *
 * AST Exploration function. Retrieves the list of function_calls
 * among the subtree of an AST node using DFS postorder.
 *
 * [param] IP, ast
 * [param] IP. ast_node
 * [param] OR - the list of AST nodes
 */
List *arp_get_function_call_ast_node_list(
  const Graph *ast,
  const GraphNode *ast_node
) {
  List *roots   = list_create(1, ast_node);
  List *labels  = list_create(1, FUNCTION_CALL_AST_NODE);

  List *result = graph_dfs_postorder_find_all(ast, roots, "type",
labels);

  /* Clean-up */

  list_free(roots);
  list_free(labels);

  return result;
}

/* EVALUATION OF THE REPORTS OF FUNCTIONS AND DOCUMENTATIONS */

/*
```

```
 * arp_evaluate_documentation_reports
 *
 * Evaluates whether the AnalyzerReportDocuments are correct.
 *
 * [param] IP, report_program
 */
void arp_evaluate_documentation_reports(
  const AnalyzerReportProgram *report_program
) {
  HashMapIterator *iterator;

  char *function_identifier;

  AnalyzerReportFunction *report_function;

  AnalyzerReportDocumentation *report documentation;

  bool warning_found = false;

  /* Iterating over each function report */

  for (
    iterator = hashmap_get_iterator(
      report_program->function_reports
    );
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    function_identifier = iterator->key;

    report_function = iterator->value;

    report_documentation = hashmap_get(
      report_program->documentation_reports, function_identifier
    );

    if (report_function->process_documentation) {
      warning_found = arp_evaluate_documentation_report(
        report_documentation,
        report_function,
        function_identifier
      ) || warning_found;
    }
  }
```

```
  if (!warning_found) {
    printf(SUCCESS_MESSAGE);
  }
}

/*
 * arp_evaluate_documentation_report
 *
 * Evaluates whether an AnalyzerReportDocumentation is correct.
 *
 * [param] IP, report_documentation
 * [param] IP, report function
 * [param] IP, function_identifier
 * [param] OR - the resulting boolean value
 */
bool arp_evaluate_documentation_report(
  const AnalyzerReportDocumentation *report_documentation,
  const AnalyzerReportFunction *report_function,
  const char *function_identifier
) {
  HashMapIterator *iterator;

  AnalyzerReportIdentifier  *function_param_report,
                            *documentation_param_report;

  HashMap *evaluation_map = hashmap_create();

  bool warning_found = false;

  /* Checking whether no documentation has been found, but no */
  /* warning has to be returned, since the function does not  */
  /* have parameters nor return type.                         */

  if (
    report_documentation == NULL
    && report_function->parameter_list_report->parameter_reports_map
                      ->length == 0
    && report_function->parameter_list_report
                      ->destination_or == false
  ) {
    hashmap_free(evaluation_map);

    return false;
```

```
  }
  else if (report_documentation == NULL) {
    printf(ERROR_MESSAGE_NO_DOC_FOUND, function_identifier);

    hashmap_free(evaluation_map);

    return true;
  }

  /* Checking whether the documentation found is empty. */

  if (
    (
      report_documentation->parameters->length == 0
      && report_documentation->destination_or == false
    )
    && (
      report_function->parameter_list_report
                     ->parameter_reports_map->length > 0
      || report_function->parameter_list_report
                        ->destination_or == true
    )
  ) {
    printf(ERROR_MESSAGE_NO_DOC_FOUND, function_identifier);

    hashmap_free(evaluation_map);

    return true;
  }

  /* Evaluating each parameter found in the documentation */

  for (
    iterator = hashmap_get_iterator(
      report_documentation->parameters
    );
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    documentation_param_report = iterator->value;

    function_param_report = hashmap_get(
      report_function->parameter_list_report->parameter_reports_map,
      iterator->key
```

```
    );

    hashmap_set(evaluation_map, iterator->key, "");

    if (function_param_report == NULL) {
      warning_found = true;

      printf(
        ERROR_MESSAGE_PARAMETER_UNKNOWN,
        function_identifier,
        iterator->key
      );
    }
    else {
      warning_found = arp_evaluate_documentation_report_destinations(
        documentation param report->destination map,
        function_param_report->destination_map,
        function_identifier,
        iterator->key
      ) || warning_found;
    }
  }

  /* Iterating over each function parameter not already evaluated */

  for (
    iterator = hashmap_get_iterator(
      report_function->parameter_list_report->parameter_reports_map
    );
    iterator != NULL;
    iterator = hashmap_iterate(iterator)
  ) {
    function_param_report = iterator->value;

    documentation_param_report = hashmap_get(
      report_documentation->parameters,
      iterator->key
    );

    if (!hashmap_get(evaluation_map, iterator->key)) {
      if (documentation_param_report == NULL) {
        warning_found = true;

        printf(
```

```
          ERROR_MESSAGE_PARAMETER_NOT_FOUND,
          function_identifier,
          iterator->key
        );
      }
      else {
        warning_found =
          arp_evaluate_documentation_report_destinations(
            documentation_param_report->destination_map,
            function_param_report->destination_map,
            function_identifier,
            iterator->key
          ) || warning found;
      }
    }
  }

  /* Evaluating the OR destination */

  if (
    report_documentation->destination_or
    && !report_function->parameter_list_report->destination_or
  ) {
    warning_found = true;

    printf(
      ERROR_MESSAGE_DESTINATION_OR_UNNECESSARY,
      function_identifier
    );
  }
  else if (
    !report_documentation->destination_or
    && report_function->parameter_list_report->destination_or
  ) {
    warning_found = true;

    printf(
      ERROR_MESSAGE_DESTINATION_OR_NOT_FOUND,
      function_identifier
    );
  }

  /* Clean-up */
```

```
  hashmap_free(evaluation_map);

  return warning_found;
}

/*
 * arp_evaluate_documentation_report_destinations
 *
 * Evaluates whether the destinations of an
 * AnalyzerReportDocumentation are correct.
 *
 * [param] IP, documentation_param_destination_map
 * [param] IP, function param destination map
 * [param] IP, function_identifier
 * [param] IP, parameter_identifier
 * [param] OR - the resulting boolean value
 */
bool arp_evaluate_documentation_report_destinations(
  const HashMap *documentation_param_destination_map,
  const HashMap *function_param_destination_map,
  const char *function_identifier,
  const char *parameter_identifier
) {
  /* Destination IP */

  bool warning_found = arp_evaluate_documentation_report_destination(
    documentation_param_destination_map,
    function_param_destination_map,
    function_identifier,
    parameter_identifier,
    DESTINATION_IP
  );

  /* Destination IK */

  warning_found = arp_evaluate_documentation_report_destination(
    documentation_param_destination_map,
    function_param_destination_map,
    function_identifier,
    parameter_identifier,
    DESTINATION_IK
  ) || warning_found;

  /* Destination IF */
```

```
  warning_found = arp_evaluate_documentation_report_destination(
    documentation_param_destination_map,
    function_param_destination_map,
    function_identifier,
    parameter_identifier,
    DESTINATION_IF
  ) || warning_found;

  /* Destination OP */

  warning_found = arp_evaluate_documentation_report_destination(
    documentation param destination map,
    function_param_destination_map,
    function_identifier,
    parameter identifier,
    DESTINATION_OP
  ) || warning_found;

  /* Destination OF */

  warning_found = arp_evaluate_documentation_report_destination(
    documentation_param_destination_map,
    function_param_destination_map,
    function_identifier,
    parameter_identifier,
    DESTINATION_OF
  ) || warning_found;

  /* Destination OV */

  warning_found = arp_evaluate_documentation_report_destination(
    documentation_param_destination_map,
    function_param_destination_map,
    function_identifier,
    parameter_identifier,
    DESTINATION_OV
  ) || warning_found;

  return warning_found;
}

/*
 * arp_evaluate_documentation_report_destination
```

```
 *
 * Evaluates whether the destination of an
 * AnalyzerReportDocumentation are correct.
 *
 * [param] IP, documentation_parameter_destination_map
 * [param] IP, function_parameter_destination_map
 * [param] IP, function_identifier
 * [param] IP, parameter_identifier
 * [param] IP, destination
 * [param] OR - the resulting boolean value
 */
bool arp_evaluate_documentation_report_destination(
  const HashMap *documentation parameter destination map,
  const HashMap *function_parameter_destination_map,
  const char *function_identifier,
  const char *parameter identifier,
  const char *destination
) {
  bool warning_found = false;

  bool *documentation_destination_flag = hashmap_get(
    documentation_parameter_destination_map, destination
  );

  bool *function_destination_flag = hashmap_get(
    function_parameter_destination_map, destination
  );

  if (
    *documentation_destination_flag
    && !*function_destination_flag
  ) {
    warning_found = true;

    printf(
      ERROR_MESSAGE_DESTINATION_UNNECESSARY,
      function_identifier,
      destination,
      parameter_identifier
    );
  }

  if (
    !*documentation_destination_flag
```

```
    && *function_destination_flag
  ) {
    warning_found = true;

    printf(
      ERROR_MESSAGE_DESTINATION_NOT_FOUND,
      function_identifier,
      destination,
      parameter_identifier
    );
  }

  return warning found;
}

/* EVALUATION OF THE DEPENDENCIES OF THE FUNCTIONS */

/*
 * arp_function_dependencies_compute
 *
 * Computes the function dependencies graph.
 *
 * [param] IP, ast
 * [param] IOP, report_program
 */
void arp_function_dependencies_compute(
  const Graph *ast,
  const AnalyzerReportProgram *report_program
) {
  ListNode *iterator;

  List *function_ast_node_list;

  /* Retrieving the list of function definitions */

  function_ast_node_list = arp_get_function_ast_node_list(ast);

  /* Creating a node in the function dependencies graph for each */
  /* function definition                                         */

  for (
    iterator = list_get_iterator(function_ast_node_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
```

```
  ) {
    arp_function_dependency_register(
      ast,
      iterator->value,
      report_program
    );
  }

  /* Computing the edges between each node in the function */
  /* dependencies graph                                    */

  for (
    iterator = list get iterator(function ast node list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    arp_function_dependency_compute(
      ast,
      iterator->value,
      report_program
    );
  }

  /* Clean-up */

  list_free(function_ast_node_list);
}

/*
 * arp_function_dependency_compute
 *
 * Computes the dependencies of a function.
 *
 * [param] IP, ast
 * [param] IP, function_ast_node
 * [param] IOP, report_program
 */
void arp_function_dependency_compute(
  const Graph *ast,
  const GraphNode *function_ast_node,
  const AnalyzerReportProgram *report_program
) {
  ListNode *iterator;
```

```
List *function_call_ast_node_list;

char *identifier;

GraphNode *dependency_parent_node, *dependency_child_node;

/* Retrieving the identifier of the function definition. */

identifier = analyzer_get_identifier_ast_node_value(
  ast, function_ast_node
);

/* Retrieving the node in the dependencies graph related to the */
/* current function.                                            */

dependency parent node = hashmap get(
  report_program->function_dependencies->nodes, identifier
);

/* Retrieving the called functions within the body of the */
/* function definition */

function_call_ast_node_list = arp_get_function_call_ast_node_list(
  ast, function_ast_node
);

for (
  iterator = list_get_iterator(function_call_ast_node_list);
  iterator != NULL;
  iterator = list_iterate(iterator)
) {
  /* Retrieving the identifier of the called function */

  identifier = analyzer_get_identifier_ast_node_value(
    ast, iterator->value
  );

  /* Retrieving the node in the dependencies graph related to */
  /* the called function                                      */

  dependency_child_node = hashmap_get(
    report_program->function_dependencies->nodes, identifier
  );
```

```
    if (dependency_child_node != NULL) {
      graph_add_edge(dependency_parent_node, dependency_child_node);
    }
  }

  /* Clean-up */

  list_free(function_call_ast_node_list);
}

/*
 * arp_function_dependency_register
 *
 * This function gets the identifier of the function and creates the
 * corresponding node in the dependencies graph.
 *
 * [param] IP, ast
 * [param] IP, function_ast_node
 * [param] IOP, report_program
 */
void arp_function_dependency_register(
  const Graph *ast,
  const GraphNode *function_ast_node,
  const AnalyzerReportProgram *report_program
) {
  GraphNode *dependency_node;

  /* Retrieving the identifier of the function definition */

  char *identifier_value = analyzer_get_identifier_ast_node_value(
    ast, function_ast_node
  );

  dependency_node = graph_add_node(
    report_program->function_dependencies, identifier_value
  );

  graph_node_set_field(
    dependency_node,
    AST_NODE_IDENTIFIER_KEY,
    function_ast_node->identifier
  );
}
```

```
/*
 * arp_function_check_recursive_call_chain_presence
 *
 * Checks the presence of recursive call chains.
 *
 * [param] IP, report_program
 */
void arp_function_check_recursive_call_chain_presence(
  const AnalyzerReportProgram *report_program
) {
  bool self_loops_detected = false;
  bool elementary_loops_detected = false;

  ListNode *iterator_a, *iterator_b;

  /* Evaluating whether the function dependency graph has any */
  /* cycles, that would show the presence of at least one     */
  /* recursive call chain in the source code.                 */

  List *scc_list = graph_get_scc(
    report_program->function_dependencies
  );

  List *self_loops_list = graph_get_self_loops(
    report_program->function_dependencies
  );

  self_loops_detected = self_loops_list->length > 0;

  elementary_loops_detected = scc_list->length
    != report_program->function_dependencies->nodes->length;

  if (self_loops_detected || elementary_loops_detected) {
    printf(ERROR_MESSAGE_RECURSIVE_CALL_CHAIN);
  }

  if (self_loops_detected) {
    for (
      iterator_a = list_get_iterator(self_loops_list);
      iterator_a != NULL;
      iterator_a = list_iterate(iterator_a)
    ) {
      printf(
        "Self loops: %s\n",
```

```
      ((GraphNode *) iterator_a->value)->identifier
    );
  }
}

if (elementary_loops_detected) {
  for (
    iterator_a = list_get_iterator(scc_list);
    iterator_a != NULL;
    iterator_a = list_iterate(iterator_a)
  ) {
    if (((List*) iterator_a->value)->length > 1) {
      printf("Call chain:");

      for (
        iterator_b = list_get_iterator(iterator_a->value);
        iterator_b != NULL;
        iterator_b = list_iterate(iterator_b)
      ) {
        printf(
          " %s",
          ((GraphNode *) iterator_b->value)->identifier
        );
      }

      printf("\n\n");
    }
  }
}

/* Clean-up */

for (
  iterator_a = list_get_iterator(scc_list);
  iterator_a != NULL;
  iterator_a = list_iterate(iterator_a)
) {
  list_free(iterator_a->value);
}

list_free(scc_list);
list_free(self_loops_list);

if (self_loops_detected || elementary_loops_detected) {
```

```
    exit(-1);
  }
}

/* REGISTERING FUNCTIONS */

/*
 * arp_register_defaults
 *
 * Registers the AnalyzerReportFunctions of printf, fprintf, scanf
 * and fscanf.
 *
 * [param] IOP, report program
 */
void arp_register_defaults(
  const AnalyzerReportProgram *report program
) {
  AnalyzerReportFunction *report_function;

  int *destination_flag;

  /* printf */

  report_function = analyzer_report_function_create(1);

  report_function->process_documentation = false;

  destination_flag = hashmap_get(
    report_function->parameter_list_report->vararg_report
                   ->destination_map,
    DESTINATION_OV
  );

  *destination_flag = 1;

  hashmap_set(
    report_program->function_reports,
    "printf",
    report_function
  );

  /* fprintf */

  report_function = analyzer_report_function_create(2);
```

```
report_function->process_documentation = false;

destination_flag = hashmap_get(
  report_function->parameter_list_report->vararg_report
                ->destination_map,
  DESTINATION_OF
);

*destination_flag = 1;

hashmap_set(
  report program->function reports,
  "fprintf",
  report_function
);

/* scanf */

report_function = analyzer_report_function_create(1);

report_function->process_documentation = false;

destination_flag = hashmap_get(
  report_function->parameter_list_report->vararg_report
                ->destination_map,
  DESTINATION_OP
);

*destination_flag = 1;

destination_flag = hashmap_get(
  report_function->parameter_list_report->vararg_report
                ->destination_map,
  DESTINATION_IK
);

*destination_flag = 1;

hashmap_set(
  report_program->function_reports,
  "scanf",
  report_function
);
```

```
  /* fscanf */

  report_function = analyzer_report_function_create(2);

  report_function->process_documentation = false;

  destination_flag = hashmap_get(
    report_function->parameter_list_report->vararg_report
                   ->destination_map,
    DESTINATION_OP
  );

  *destination_flag = 1;

  destination flag = hashmap get(
    report_function->parameter_list_report->vararg_report
                   ->destination_map,
    DESTINATION_IF
  );

  *destination_flag = 1;

  hashmap_set(
    report_program->function_reports,
    "fscanf", report_function
  );
}

/*
 * analyzer_report_program_register_documentation
 *
 * Registers each AnalyzerReportDocumentation in a map, using the
 * corresponding function identifier as key.
 *
 * [param] IP, ast
 * [param] IOP, report_program
 */
void arp_register_documentation(
  const Graph *ast,
  const AnalyzerReportProgram *report_program
) {
  ListNode *iterator;
```

```
  GraphNode *previous, *function_ast_node, *comment_ast_node;

  List *external_labels, *external_roots;

  List *function_labels, *function_roots;

  List *comment_labels, *comment_roots;

  List *external_node_list;

  AnalyzerReportDocumentation *documentation_report;

  char *function identifier;

  /* Retrieving the ordered list of external declarations */

  external_labels = list_create(1, EXTERNAL_DECLARATION_AST_NODE);

  external_roots = list_create(
    1, hashmap_get(ast->nodes, PROGRAM_AST_NODE)
  );

  external_node_list = graph_dfs_preorder_find(
    ast, external_roots, "type", external_labels
  );

  /* Iterating over the list of external declarations */

  function_labels = list_create(1, FUNCTION_DEFINITION_AST_NODE);
  comment_labels  = list_create(1, COMMENT_AST_NODE);

  for (
    iterator = list_get_iterator(external_node_list);
    iterator != NULL;
    iterator = list_iterate(iterator)
  ) {
    /* Checking whether the current external declaration contains */
    /* a function definition.                                     */

    function_roots = list_create(1, iterator->value);

    function_ast_node = graph_dfs_preorder_find_first(
      ast, function_roots, "type", function_labels
    );
```

```
  previous = iterator->prev->value;

  if (function_ast_node != NULL && previous != NULL) {
    function_identifier = analyzer_get_identifier_ast_node_value(
      ast,
      function_ast_node
    );

    /* Checking whether the previous external declaration */
    /* contains a comment */

    comment_roots = list_create(1, previous);

    comment_ast_node = graph_dfs_preorder_find_first(
      ast, comment_roots, "type", comment_labels
    );

    if (comment_ast_node != NULL) {
      documentation_report = analyzer_report_documentation_create(
        ast, comment_ast_node
      );

      hashmap_set(
        report_program->documentation_reports,
        function_identifier,
        documentation_report
      );
    }

    /* Clean-up */

    list_free(comment_roots);
  }

  /* Clean-up */

  list_free(function_roots);
}

/* Clean-up */

list_free(function_labels);
list_free(comment_labels);
```

```
  list_free(external_labels);
  list_free(external_roots);
  list_free(external_node_list);
}
```

8.5 Main

8.5.1.1 `main.c`

```
#include <stdio.h>
#include <stdlib.h>

#include "mod_analyzer.h"
#include "mod_lexer.h"
#include "mod_parser.h"

int main(int argc, char **argv) {
  FILE *input;
  Lexer *lexer;
  Graph *ast;

  if (argc == 1) {
    printf("[ERRORS] No arguments.\n");
    exit(-1);
  }

  input = fopen(argv[1], "r");

  lexer = lexer_create(input);

  ast = parser_parse_file(lexer);

  lexer_free(lexer);

  analyzer_analyze(ast);

  ast_free(ast);

  fclose(input);

  return 0;
}
```

Capitolo 9

Bibliografia

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein (2009). *Introduction to Algorithms, 3rd edition*. The Massachusetts Institute of Technology.
- Paul Deitel, Harvey Deitel (2012). *C: How to Program, 7th edition*. Pearson.
- International Organization for Standardization (1990). *American National Standard for Programming Languages – C (ISO/IEC 9899:1990)*. American National Standards Institute.
- John E. Hopcroft, Rajeev Motwani, Jeffrey D. Ullman (2007). *Introduction to Automata Theory, Languages, and Computation, 3rd edition*. Pearson.
- Jeff Lee (1985). *ANSI C grammar, Lex*. Disponibile visitando l'indirizzo web https://www.lysator.liu.se/c/ANSI-C-grammar-l.html (*data ultima visita: 13 novembre 2023*).