Estensione dell'editor Bluefish per il linguaggio descrittivo di robot didattico NXT-GTD

Valentini Riccardo

Indice

Sommario

Il linguaggio NXT-GTD è la versione testuale del linguaggio visuale NXT usato per la programmazione dei robot didattici prodotti dalla LEGO®. La presenza di una versione testuale è dovuta alla necessità di documentare programmi relativamente complessi e resi in NXT-G che è un linguaggio grafico-iconico. Per questo linguaggio tuttavia, non esistono ad ora editor in grado di riconoscerlo. Per questo motivo si è deciso di fornire un ambiente di sviluppo che permetta di avere le stesse facilitazioni disponibili per gli altri linguaggi di programmazione. Per far ciò è stato usato come base l'editor Bluefish, il quale permette la definizione di sintassi personali mediante la creazione di un file in formato *bflang2*, dalla struttura molto simile ad XML. Scopo di questa trattazione è presentare i passi necessari alla creazione del file di definizione della sintassi, e all'integrazione con Bluefish in modo da ottenere un editor che riconosca il linguaggio NXT-GTD.

Capitolo 1

Introduzione

1.1 Scopo

Il linguaggio NXT-G nasce per la programmazione dei robot didattici Mindstorm NXT, linea prodotta dalla LEGO®. Questo linguaggio, usato per definire il comportamento del robot, nasce in modalità grafica, mediante l'uso di un software apposito. La descrizione del programma, e quindi l'insieme delle azioni da compiere, è costituita da uno schema a blocchi il quale visualizza il flusso delle istruzioni in modo iconico. Per poter avere una visione completa del programma si ha bisogno di visualizzare l'intero schema che è possibile solo usando il software proprietario.

Per questo motivo è stata prodotta una versione testuale del linguaggio, che permettesse una descrizione più compatta e più fruibile, ovvero l'NXT-GTD. Come per ogni linguaggio di programmazione, esso può essere scritto mediante un semplice programma per file di testo txt, ma la scrittura con questa modalità non è la più comoda e può portare facilmente a commettere errori. Inoltre, il linguaggio NXT-GTD posside al suo interno varie ripetizioni e l'uso di acronimi che possono rendere pesante e non scorrevole la scrittura del codice.

Questo è il motivo per cui si è scelto di integrare un editor affinché riconosca il linguaggio NXT-GTD ed aiuti il programmatore nella scrittura del codice. L'editor utilizzato è Bluefish, un programma gratuito multipiattaforma, che permette la definizione di sintassi personalizzate in aggiunta a quelle standard. La descrizione di una nuova sintassi viene effettuata mediante file .bflang2. I file .bflang2 contengono una descrizione della sintassi mediante

tag in una struttura molto simile all'XML.

Lo scopo di questo lavoro è di mostrare come si è giunti all'integrazione dell'editor Bluefish affinchè riconosca il linguaggio NXT-GTD, e permetta quindi di applicare le proprie funzionalità di evidenziazione e suggerimento del codice al linguaggio in versione testuale dei robot didattici LEGO®.

1.2 Il linguaggio NXT-GTD

NXT-G (NXT Graphic) è il linguaggio visuale che permette di programmare il componente principale di un robot NXT, ovvero il *brick NXT*. Questo linguaggio è costituito da blocchi con parametri di ingresso e di uscita che rappresentano i reali collegamenti del brick NXT con le periferiche. I vari blocchi del linguaggio vengono affiancati secondo un ordine logico, in modo da determinare un flusso continuo di operazioni che il robot dovrà eseguire. NXT-GTD è una trasposizione testuale di tali blocchi i cui nomi sono contrazioni dei nomi dei comandi della versione italiana. Questo linguaggio descrittivo rende più comprensibile ed immediata la presentazione di programmi o parti di esso sostituendosi al codice grafico o affiancandolo in modo completo includendo anche i parametri.

1.3 Aspetti generali del linguaggio

- Un nome viene racchiuso tra " se include degli spazi;
- Un blocco è rappresentato da un nome seguito dal carattere ':'. Questa definizione consente di riferirsi al blocco in altri punti del programma;
- I parametri di un blocco sono specificati mediante una coppia <nome parametro>=<valore> dove valore può essere una costante predefinita, un valore numerico o logico oppure l'uscita di un altro blocco;
- Le variabili possono essere specificate mediante il comando speciale **DichVar** nel quale bisogna specificare il nome della variabile ed il suo tipo. Il tipo della variabile può essere: **LOGICA, NUM** o **TESTO**;
- Si possono definire dei blocchi personalizzati mediante il comando **MioBlocco**. Oltre a specificare il nome del blocco bisogna fornire due liste di parametri, una per quelli di ingresso ed una per quelli di uscita. Le

due liste (**ParIn, ParUsc**) sono costituite da una sequenza di dichiarazioni del tipo: ¡nome¿.¡tipo parametro¿, dove ¡tipo parametro¿ è da scegliersi fra i tipi possibili per una variabile. All'interno del blocco può essere inserito un qualsiasi altro comando del linguaggio;

- I blocchi possono essere raggrupati in strutture di controllo, il cui corpo è delimitato da parentesi quadre spesso accompagnate da un'etichetta per migliorarne la leggibilità.
  Nel controllo interruttore l'etichetta è accompagnata da una parola a seconda del tipo di divisione del flusso.

  Nel caso di due scelte l'etichetta prende la forma <nome>.VERO oppure <nome>.FALSO a seconda se il blocco corrisponde al verificarsi o meno della condizione.

  Nel caso di più scelte l'etichetta assume la forma <nome>.caso.n dove n è il numero dell'opzione corrispondente al valore assunto dalla variabile nella condizione specificata;

- Il linguaggio è case sensitive, ovvero fa differenza tra maiuscole e minuscole;

- I commenti iniziano con – (due trattini) e terminano a fine riga.

Per quanto riguarda le etichette si consiglia che esse inizino con una lettera minuscola per facilitarne l'interpretazione da parte dell'editor. Etichette con la lettera iniziale maiuscola potrebbero non essere correttamente riconosciute anche se il loro uso non costituisce un errore sintattico.

Gli elementi del linguaggio fin qui illustrati servono solo a dare un'idea generale sulla forma del linguaggio. Una descrizione più completa ed esaustiva del linguaggio può essere trovata in [1].

1.4 Esempi di comandi

Per maggiore chiarezza vengono forniti alcuni esempi di comandi di NXT-GTD.

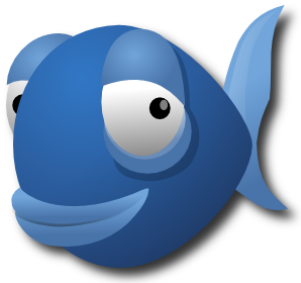
```
Sposta(
    Porte = A|B|C|AB|AC|BC|ABC
    Dir = AVA|IND|STOP
```

```
    StSin = A|B|C
    StDes = A|B|C
    Sterza = <-100...100>
    Pot = <0..100>
    Dur = ILLIM|<valore intero>[GRA|ROT|s]
    ProsAz = FRENA|FOLLE
)
In = MotoreSin(=1|3|3), MotoreDes(=1|2|3), Dir(=VERO|FALSO),
     Sterza, Pot, Dur, ProsAz(=VERO|FALSO)
Usc = vedi In


Motore(
    Porta = A|B|C
    Dir = AVA|IND|STOP
    Az = COST|ACC|RAL --se Dur.Tipo==GRA||Dur.Tipo=ROT
    Pot = <0..100>
    PotMot = SI|NO
    Dur = ILLIM|<valore intero>[GRA|ROT|s]
    Att = SI|NO     --se Dur.Tipo==GRA||Dur.Tipo==ROT
    ProsAz = FRENA|FOLLE    --se Dir=STOP||Dur.Tipo=s||Att==SI
)
In = Porta. Dir(=VERO|FALSO), Az(=0|1|2), Pot, PotMot(=VERO|FALSO),
     Dur, Att(=VERO|FALSO), ProsAz(=VERO|FALSO)
Usc = vedi In, DirUsc(=VERO|FALSO), GraUsc(=<valore naturale>)
```

Capitolo 2

L'ambiente di sviluppo Bluefish



2.1 L'editor

Bluefish è un editor libero e gratuito rilasciato sotto la licenza GNU GPL. Esso può lavorare su diverse piattaforme quali Linux, Windows e MacOS-X.

Grazie alla sua interfaccia grafica semplice e pulita può essere facilmente utilizzabile anche da un non esperto di questo tipo di programmi. Il menu contestuale inoltre è molto intuitivo e dall'uso immediato. Il programma è fornito in diverse lingua, tra cui è disponibile anche una versione in italiano.

Come editor esso è orientato principalmente allo sviluppo di siti web, infatti nell'interfaccia principale sono presenti varie scorciatoie per i principali comandi di HTML, CSS, e PHP. Nonostante il naturale orientamento al web, i linguaggi di programmazione supportati da Bluefish sono molteplici, i principali dei quali sono:

- Ada;

- ASP .NET and VBS;
- C/C++;
- CSS;
- Google Go;
- HTML, XHTML and HTML5;
- Java and JSP;
- JavaScript and jQuery;
- Lua;
- Octave/MATLAB;
- MediaWiki;
- Pascal;
- Perl;
- PHP;
- Python;
- Ruby;
- Shell;
- Scheme;
- Wordpress;
- XML;

Per ciascuno dei linguaggi supportati, l'editor fornisce varie funzioni.

2.2 Caratteristiche dell'editor

Le principali caratteristiche (o funzioni) dell'editor che interessano per il nostro scopo sono: *syntax-highlighting* e *syntax-proposal*, ovvero la capacità di evidenziare le parole nel testo in base al significato sintattico e di suggerire le istruzioni in base a qunato digitato ed al contesto. La funzione di *syntax proposal* ci permette una lettura del codice molto più chiara ed ordinata, poiché mette in risalto con colori differentii quelle che sono parole chiavi, tipi base, variabili, funzioni ecc... dando la possibilità di un controllo più veloce della correttezza del codice. Essendo un semplice editor, Bluefish non svolge una funzione di controllo degli errori, perciò la funzione di evidenziazione è il principale strumento per verificare l'esattezza del codice. La *syntax-highlighting* viene svolta anche da molti altri editor concorrenti ed è una delle funzioni base per un buon editor.

Il *syntax-proposal* invece non viene implementato, in generale, dagli altri editor presenti in commercio o scaricabili dalla rete, poiché esso necessita un riconoscimento al volo di quanto digitato, e viene di solito implementato dagli editor che eseguono anche controllo della grammatica. Il suggerimento della sintassi si rende particolarmente utile quando essa è costituita da parole lunghe, o abbreviazioni facilmente confondibili o non particolarmente mnemoniche. Bluefish ad ogni digitazione visualizza un elenco delle parole che corrispondono ai caratteri fin'ora inseriti e che potrebbero quindi far parte di un comando (o istruzione) del nostro linguaggio. L'elenco può essere scorso velocemente con i tasti freccia e una volta selezionata la parola da noi voluta essa viene inserita premendo 'invio'. Bastano due o tre caratteri per sfoltire di molto la lista e trovare subito il comando desiderato, velocizzando di molto la scrittura del codice e proteggendoci da eventuali errori di battitura.

Il motivo principale per cui Bluefish è adatto al nostro scopo non è però la presenza delle due funzioni sopra descritte, ma la possibilità di aggiungere altri linguaggi riconoscibili oltre a quelli standard. L'aggiunta di un nuovo linguaggio avviene mediante la descrizione della sintassi attraverso un file di tipo bflang2 (Bluefish Language 2) la cui struttura è quella di un file XML personalizzato. Mediante questa procedura è possibile integrare l'editor in modo che riconosca non solo i linguaggi non ancora supportati, ma anche linguaggi personali da noi ideati o specifici per applicazioni personali. È proprio la semplicità con cui è possibile definire una nuova sintassi il punto di forza dell'editor, che non necessita di usare lexer o altre procedure più complesse ed articolate. La struttura di un file bflang2 e la procedura per far

riconoscere una descrizione sintattica personale all'editor verranno descritte in dettaglio successivamente.

Un ulteriore vantaggio di Bluefish è la possibilità di definire delle *references* ovvero dei riferimenti o delle guide per ogni elemento della nostra sintassi. Mediante questa funzione possiamo definire una descrizione di un dato comando oppure inserire dei suggerimenti sull'uso di un tipo di dati o qualsiasi altro elemento del linguaggio. Questo ovviamente è da implementare solo nel caso di linguaggi non già presenti nell'editor e quindi da noi aggiunti. Per i linguaggi standard, nativamente riconosciuti dall'editor è fornita una descrizione delle istruzioni principali. I *references*, se presenti, vengono visualizzati durante la digitazione dell'istruzione o semplicemente posizionando il puntatore del mouse sopra la parola interessata. La presenza di questi elementi permette ad un programmatore non esperto o non pratico di quel linguaggio di avere un aiuto nella scrittura del codice.

2.3 Altre funzionalità

Vi sono inoltre svariate altre funzionalità dell'editor che lo rendono una buona scelta per un programmatore. Bluefish si avvia molto velocemente ed è in grado di gestire molti file contemporaneamente i quali vengono aperti mediante schede della stessa finestra. Per ogni scheda è possibile definire un diverso linguaggio con il quale lavorare.

La funzione cerca e sostituisci è compatibile con le espressioni regolari Perl, ed agisce anche su file sull'hard disk (anche se non aperti).

Recupera in automatico i cambiamenti sui file modificati in caso di chiusura non regolare del programma.

Supporta codifiche multiple: esso lavora internamente con UTF-8 ma permette il salvataggio in una qualsiasi altra codifica.

Traduzioni in 17 lingue.

Dalla pagina internet principale di Bluefish (www.bluefish.openoffice.nl) si possono leggere le seguenti dichiarazioni rilasciate da alcuni siti internet noti nella rete.

> *Bluefish is by far the most powerful among the HTML editors we tested. It is probably the most potent editor for Linux in general.* (www.suse.com)

> Bluefish è alla lunga il più strabiliante tra gli editor HTML che abbiamo testato. Esso è probabilmente il più potente editor per Linux in generale. (www.suse.com)
>
> *One of the most powerful editors for Linux + Supports many programming and markup languages + Lots of time saving tools for experienced users + Friendly enough for beginners + Its wealth of features will make your programming easier + While letting you maintain control over your code* (www.lindows.com)
>
> Uno dei più strabilianti editor per Linux + Supporta molti linguaggi di programmazione e di *markup* + Molti strumenti che fanno risparmiare tempo per utenti esperti + Abbastanza amichevole per i principianti + La sua ricchezza di opzioni renderà la vostra programmazione più semplice + Permettendovi di mantenere il controllo sul vostro codice. (www.lindows.com)

La versione utilizzata per questo progetto è la 2.2.3. Essa è reperibile direttamente dal sito dell'editor: *http://www.bluefish.openoffice.nl/download.hml* dal quale è possibile ottenere sia la versione per Linux che per Windows.

Capitolo 3

Il linguaggio Bflang2 (Bluefish language 2)

3.1 Qualche cenno su XML

L'eXtensible Markup Language è un metalinguaggio usato per rappresentare contenuti testuali in gerarchia. Esso è una semplificazione del linguaggio SGML (Standard Generalized Markup Language) creato con lo scopo di avere una descrizione standard dei contenuti per il web. Il nome XML ci indica che esso è un linguaggio a marcatori (come HTML) ed è estensibile, ovvero è possibile definire dei tag personalizzati. A differenza di HTML che è un linguaggio usato per creare documenti per il web, l'XML serve a descrivere altri linguaggi o documenti con gerarchie insite. Un esempio di utilizzo attuale, si ha nell'esportazione dei dati tra diversi DBMS. Poichè un file scritto mediante questo linguaggio è composto di solo testo, esso è compatibile con tutte le piattaforme in grado di supportare file testuali.

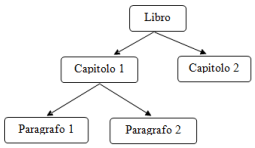
Nonostante siano due linguaggi differenti, l'XML è composto da tag ed attributi come l'HTML, le cui regole sono però, sotto certi aspetti, più rigide rispetto a quest'ultimo. Ogni tag ha il seguente formato:

```
<nome_tag attributo1="valore_attributo1"... attributo(n)="valore_attributo(n)" >
    contenuto
</nome_tag>
```

dal quale possiamo notare un elemento di apertura con il nome del tag e le informazioni ad esso associate (attributi) ed un elemento di chiusura che deve essere sempre presente. Ogni tag è delimitato dalle parentesi angolate <> ed

al suo interno il contenuto può essere costituito da semplice testo oppure da altri tag. Mediante l'annidamento di diversi tag si crea la struttura gerarchica del documento che vogliamo descrivere. Come conseguenza, tutti i tag di chiusura seguono l'ordine inverso dei rispettivi tag di apertura. Il valore degli attributi deve essere sempre espresso tra apici (").

In ogni documento vi è un unico tag contenitore che racchiude tutti gli altri che viene chiamato radice. Per la struttura propria del linguaggio, a volte un file XML viene rappresentato come un albero, nel quale ogni tag interno rappresenta un nuovo ramo. Ad esempio il seguente albero può essere



rappresentato mediante il seguente codice XML:

```
<libro titolo="Nome_titolo" autore="nome_autore" casaEditrice="nome_casa">
  <capitolo titolo="Capitolo 1">
      <paragrafo>Paragrafo 1</paragrafo>
      <paragrafo>Paragrafo 2</paragrafo>
  </capitolo>
  <capitolo titolo="Capitolo 2">
  </capitolo>
</libro>
```

Quando un tag non accetta contenuto al suo interno si può semplificarne la scrittura omettendo l'elemento di chiusura e usando `/>` anzichè la semplice `>`. I due seguenti tag che rappresentano il ritorno a capo in HTML sono equivalenti:

```
<br></br>
<br />
```

Mentre in HTML viene accettato anche il tag `<br>` senza elementi di chiusura, in XML esso costituisce un errore. Altra differenza è che l'XML fa distinzione tra caratteri minuscoli e maiuscoli, perciò i nomi dei tag devono coincidere anche sotto questo aspetto.

Come per ogni linguaggio è possibile inserire all'interno del codice dei commenti, i quali sono racchiusi tra i simboli speciali `<!-` e `->`.

XML Schema

Nel 2001 il W3C ha rilasciato un metodo per la descrizione dei linguaggi mediante XML, l'XSD (XML Schema Definition). Riportando la definizione di [2] di XSD, possiamo dire che questa descrizione permette di definire qualsiasi struttura XML specificandone:

- i nomi dei tag e degli attributi di un documento, nonché la loro posizione all'interno dello stesso;
- i contenuti di tag e attributi tramite tipi di dati noti e diffusi in tutto il panorama informatico, come integer, double, string, ecc;
- i valori di eventuali nodi di tipo testo o attributo tramite espressioni regolari.

Ogni documento XSD inizia con il seguente tag di testa

```
<?xml version="1.0"?>
```

nel quale viene indicata la versione del documento XML usata ed eventuali informazioni di encoding.

Il nodo radice di ogni Schema è costituito dal seguente tag:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" />
```

all'interno del quale viene inserita la descrizione del documento, la quale è costituita principalmente da due elementi: tipi semplici e tipi complessi.

I **tipi semplici** definiscono elementi che non possono contenerne altri e non posseggono attributi. Il w3c ha definito alcuni tipi di dato semplici predefiniti alcuni dei quali sono:

- xs:string (stringhe)
- xs:integer (numeri interi)
- xs:decimal (numeri decimali)
- xs:boolean (valori booleani)
- xs:date (data)
- xs:time (ora)

- xs:uriReference (URL)

Un esempio di dichiarazione di un tipo semplice è il seguente:

```
<xs:element name='is_regex' type='xs:boolean' />
```

con il quale si indica che l'elemento 'is_regex' accetta solo valori booleani e cioè 1 o 0.

I **tipi complessi** definiscono elementi che posseggono attributi e possono contenere al loro interno ulteriori elementi. Essi vengono specificati descrivendone la struttura. Un tipo complesso viene dichiarato con il seguente codice:

```
<xs:element name='NOME_ELEMENTO'>
   <xs:complexType>
      ...
   </xs:complexType>
</xs:element>
```

dove al posto dei puntini vengono inseriti altri tag che possono essere di descrizione o attributi. I tag utilizzabili per la definzione di un tipo complesso sono:

- `<xs:sequence>` che elenca una sequenza di elementi definiti al suo interno;
- `<xs:choice>` che indica un gruppo di elementi di cui uno solo può essere presente;
- `<xs:all>` che elenca elementi che possono comparire o meno in un qualsiasi ordine;
- `<xs:simpleContent>` indica che l'elemento avrà degli attributi ma può contenere all'interno solo testo o tipi semplici;
- `<xs:complexContent>` indica che l'elemento può contenere solo altri tag o essere vuoto;
- `<xs:attribute>` indica un singolo attributo.

Per il tag **attribute** è possibile specificare se esso sia obbligatorio o meno, e se possiede un valore di default. Prendiamo i seguenti tag come esempio:

```
<xs:attribute name='autore' type='xs:string' use='required' />
<xs:attribute name='anno' type='xs:integer' use='default' value='2012' />
```

il primo tag specifica un attributo di tipo stringa il cui valore deve essere indicato, mentre il secondo tag indica un attributo di tipo numerico il cui valore di default (specificato da `value`) è 2012.

L'XSD è uno strumento molto potente la cui descrizione è più articolata di quanto fin qui descritto. Esso permette una definizione più dettagliata delle grammatiche e di definire altre strutture non menzionate poiché non necessarie alla comprensione degli argomenti qui affrontati. Per una migliore trattazione rimando a [2] o [3].

3.2 Bflang2 Schema

Dal sito principale di Bluefish è possibile visualizzare lo Schema del linguaggio bflang2. Poiché non vi sono spiegazioni della sintassi del linguaggio in guide od altro materiale fornito dagli sviluppatori, questo documento è l'unica descrizione completa del linguaggio. Qui di seguito è riportato lo Schema completo con i commenti ai singoli tag. Gli spazi presenti all'interno di una stringa nel codice vengono evidenziati con il simbolo: ␣.

La parola **element** specifica un tag a sè stante, e quando vi sono più element in uno stesso tag essi devono considerarsi mutuamente esclusivi. La parola **attribute** invece indica i parametri del tag che possono essere specificati con, tra parentesi graffe, il tipo di valore che accetta.

Il tag di partenza è **bflang** il quale racchiude tutti gli altri elementi. Esso definisce il nome letto dall'utente, la versione del codice, ed altri parametri utili per il compilatore.

bflang2 Schema

```
element bflang
{
  attribute name { xsd:normalizedString },
  attribute version { xsd:decimal },
  (
    attribute contexts { xsd:positiveInteger },
    attribute matches { xsd:positiveInteger },
    attribute table { xsd:positiveInteger }
    )?,
```

I valori di **contexts**, **matches**, **table**, definiscono quanta memoria dovrebbe essere allocata per le rispettive tabelle, in modo da ottimizzare la velocità di caricamento.

Vi sono tre tag principali interni a **bflang** che individuano tre regioni del documento. Il primo, **header**, specifica una regione che viene sempre caricata, anche se il linguaggio non è attualmente in uso. La seconda, **properties**, specifica alcune proprietà del linguaggio che sono comuni per i linguaggi di programmazione in genere. Infine, nella regione individuata dal tag **definition**, vi è la descrizione vera e propria della sintassi, che viene caricata dall'editor solo quando viene selezionato il linguaggio descritto per uno dei file aperti.

3.2.1 header

Il tag *mime type* definisce su quale tipo di file il linguaggio viene salvato. Ogni linguaggio deve avere almeno un unico *mime type* utilizzabile dall'utente, ma possono anche essere multipli.

Con il tag **option** è possibile attivare/disabilitare le classi o i gruppi (e quindi gli elementi in essi contenuti) mediante il parametro **default**.

```
10      element header
    {
12          element mime
            {
14              attribute type { xsd:token },
                empty
16          }+,
            (
18            element option
                {
20                  attribute name { xsd:ID },
                    attribute default { xsd:boolean },
22                  attribute description { xsd:string }?,
                    empty
24              }
```

Il tag **highlight** permette di definire gli stili dei contesti,elementi, gruppi e tag che possono evidenziati nel documento. Tra i possibili stili dell'attributo **style** sono elencati i nomi di quelli predefiniti dall'edito, ma è comunque

possibile modificarli o definirne di nuovi (tramite le opzioni dell'editor).

```
            element highlight
              {
                   attribute name { xsd:ID },
                   attribute style
                   {
                     xsd:token "attribute"
                     xsd:token "brackets"
                     xsd:token "comment"
                     xsd:token "function"
                     xsd:token "keyword"
                     xsd:token "operator"
                     xsd:token "preprocessor"
                     xsd:token "special-attribute"
                     xsd:token "special-function"
                     xsd:token "special-keyword"
                     xsd:token "special-tag"
                     xsd:token "special-tag2"
                     xsd:token "special-tag3"
                     xsd:token "special-type"
                     xsd:token "special-value"
                     xsd:token "string"
                     xsd:token "tag"
                     xsd:token "type"
                     xsd:token "value"
                     xsd:token "variable"
                     xsd:token "warning"
                   },
                   empty
              }
         )*,
    empty
    },
```

3.2.2 properties

All'interno del tag **properties** è possibile definire alcune caratteristiche comuni del linguaggio.

Il tag **comment** serve a specificare i caratteri di inizio e fine che identificano un commento su più linee, e/o il carattere di inizio dei commenti a singola linea.

I tag **smartindent** e **smartoutdent** specificano quali caratteri (seguiti da un a capo) debbano incrementare o diminuire l'indentazione.

Il tag **default_spellcheck** specifica se le regioni che non sono evidenziate debbano avere controllo ortografico o meno; questa opzione viene usata solitamente per linguaggi come HTML o XML.

```
    element properties
    {
     (
        element comment
        {
            attribute type
            {
                xsd:token "block" | xsd:token "line"
            },
            attribute start { xsd:string },
            attribute end { xsd:string }?,
            empty
        }
        element smartindent
        {
            attribute characters { xsd:string },
            empty
        }
        element default_spellcheck
        {
            attribute enabled { xsd:boolean }
        }
     )+,
    empty
    }?,
```

3.2.3 definition

Questa sezione inizia mediante il tag:

```
    element definition { bf.element.context* },
    empty
  }
```

All'interno del tag **definition** vengono inseriti gli elementi che descrivono la sintassi del linguaggio i cui principali elementi sono: *element, context, group, reference*.

Poiché i tag di questa regione possono essere annidati tra loro su più livelli essi vengono definiti mediante una struttura leggermente differente da quella fin qui usata nello schema. Per ciascuno di essi viene definito un nome il quale viene descritto in un punto diverso dello schema. Ad esempio *bf.attribute.autocomplete.element* indica che all'interno del tag in cui è inserito può essere inserito il tag *autocomplete* che viene però specificato in un altro punto.

Il tag **element** indica un elemento del codice che viene evidenziato, di cui viene suggerito l'autocompletamento o che ha delle informazioni riferite (references). L'attributo principale di questo tag è il parametro **pattern** che può essere una parola chiave oppure una espressione regolare (nel qual caso bisogna impostare *is_regex=1*). Al suo interno possono essere inseriti altri tag quali *element, context, group, reference*.

```
bf.element.element =
element element
    {
      (
       (
         attribute pattern { xsd:string },
         bf.attribute.autocomplete.element,
         bf.attribute.autocomplete.extended,
         bf.attribute.class.notclass,
         bf.attribute.case_insens,
         bf.attribute.highlight,
         bf.attribute.is_regex,
         attribute starts_block { xsd:boolean }
         | attribute ends_block { xsd:boolean },
         attribute blockstartelement {
             xsd:IDREF { pattern = "[et]\.[a-zA-Z0-9\.]+" }
         }
       )
      )?,
       attribute blockhighlight { xsd:string }?,
       attribute ends_context { xsd:positiveInteger }?,
       attribute id
       {
         xsd:ID { pattern = "e\.[a-zA-Z0-9_.]+" }
```

```
        }?,
110     attribute mayfold { [ a:defaultValue = "0" ] xsd:boolean }?,
        attribute tagclose_from_blockstack
112     { [ a:defaultValue = "0" ] xsd:boolean }?,
       ( bf.element.context | bf.element.element |
114           bf.element.group | bf.element.reference )*
     )
116  attribute idref { xsd:IDREF { pattern = "e\.[a-zA-Z0-9_.]+" } }
    )
118 }
```

Poiché i *context* possono essere nidificati il parametro ends_context assume come valore il numero di contesti che si vogliono chiudere. Gli elementi *starts_block* e *ends_block* sono opzionali e, nel caso il loro valore venga impostato ad '1' (true), deve essere specificato qual è l'elemento che fa iniziare il blocco mediante il parametro *blockstartelement*. Specificare il parametro **id** è essenziale per poter far riferimento a questo elemento in altri punti del codice. Il parametro **idref** invece, se specificato, indica che l'elemento assume la stessa definizione dell'elemento specificato e pertanto nessun'altro parametro deve essere specificato.

Il tag **context** permette di definire una regione all'interno del codice nel quale vengono riconosciuti solamente i tag indicati al suo interno. Il parametro più importante per questo tag è *symbols*, il quale è un elenco di tutti i simboli (escludendo l'alfabeto) che si possono incontrare all'interno del contesto. L'uso di *id* e *idref* è identico a quanto descritto in precedenza.

```
bf.element.context =
120     element context
        {
122     (
         (
124         attribute symbols { xsd:string },
            bf.attribute.highlight,
126         attribute id { xsd:ID { pattern = "c\.[a-zA-Z0-9_.]+" } }?,
            ( bf.element.element | bf.element.group | bf.element.tag )+
128         )
            | attribute idref {
130              xsd:IDREF { pattern = "c\.[a-zA-Z0-9_.]+" }
                }
132      ),
         empty
134     }
```

Quando più elementi condividono gli stessi valori di alcuni attributi o parametri è possibile raggrupparli assieme mediante il tag **group** così da definire tali valori una volta sola. I gruppi definiti mediante questo tag possono essere abilitati o disabilitati con il tag *option* nell'header.

```
bf.element.group =
    element group
    {
        bf.attribute.autocomplete.attribute,
        bf.attribute.autocomplete.element,
        bf.attribute.autocomplete.extended,
        bf.attribute.attribhighlight,
        bf.attribute.case_insens,
        bf.attribute.class.notclass,
        bf.attribute.highlight,
        bf.attribute.is_regex,
        ( bf.element.element | bf.element.group |
                bf.element.tag )+,
        empty
    }
```

Il tag **tag** permette di definire linguaggi basati su SGML/XML. Con esso è possibile definire qualunque cosa usando una giusta combinazione di elementi multipli e tag di contesto. Il parametro *no close* serve ad inidcare tag che sono singoli come ad esempio
 e <hr>.

```
bf.element.tag =
    element tag
    {
     (
      (
        attribute name { xsd:token },
        bf.attribute.autocomplete.attribute,
        bf.attribute.autocomplete.element,
        bf.attribute.autocomplete.extended,
        bf.attribute.case_insens,
        bf.attribute.highlight,
        bf.attribute.is_regex,
        (
            attribute attributes {
                xsd:string { pattern = "[a-zA-Z0-9][a-zA-Z0-9:,-_]+" }
            },
            bf.attribute.attribhighlight
```

```
          )?,
168       attribute id { xsd:ID { pattern = "t\.[a-zA-Z0-9_.]+" } }?,
          attribute no_close { xsd:boolean }?,
170       attribute sgml_shorttag { xsd:boolean }?,
         ( bf.element.context | bf.element.reference )*
172      )
         | attribute idref
174      {
           xsd:IDREF { pattern = "t\.[a-zA-Z0-9_.]+" }
176      }
        ),
178     empty
    }
```

Il tag **reference** permette di definire una descrizione degli elementi o dei tag, la quale viene visualizzata durante la scrittura o posizionandosi sopra con il mouse. La descrizione deve contenere solo codice Pango valido e/o testo in accordo con http://library.gnome.org/devel/pango/stable/PangoMarkupFormat.htm la cui correttezza non è verificata dall'editor.

```
180 bf.element.reference =
    element reference { ( text | element * { text } )*, empty }
```

I successivi tag di **autocomplete** determinano se l'opzione di autocompletamento automatico dell'elemento è attivo o meno e quale testo aggiungere in caso positivo. Questa funzione è diversa dal *syntax-proposal* poiché non suggerisce le istruzioni, ma una volta digitato il comando essa inserisce in automatico dell'altro codice specificato da *autocomplete_append*.

```
182 bf.attribute.autocomplete.element =
    // Enable ('1') or disable ('0') autocompletion feature for item
184 attribute autocomplete { xsd:boolean }?,
    // Automatically appended string to item
186 attribute autocomplete_append { xsd:string }?

188 bf.attribute.autocomplete.attribute =
    // Automatically appended string to attribute
190 attribute attrib_autocomplete_append { xsd:string }?,
    // Set cursor position after auto-completion of
192 //attribute back by X characters
    attribute attrib_autocomplete_backup_cursor { xsd:positiveInteger }?
194
```

```
bf.attribute.autocomplete.extended =
    // Auto-complete regular expression with this string
    attribute autocomplete_string { xsd:string }?,
    // Set cursor position after auto-completion
    //back by X characters
    attribute autocomplete_backup_cursor { xsd:positiveInteger }?
```

I parametri *class* e *notclass* vengono usati similmente agli elementi di un gruppo per determinare l'abilitazione o meno degli elementi di una stessa classe mediante le opzioni dell'header.

```
bf.attribute.class.notclass =
    (
        attribute class { xsd:IDREF } | attribute notclass { xsd:IDREF }
    )?
```

Il parametro *highlight* specifica a quale categoria definita nell'header quell'elemento appartiene e di conseguenza con quale stile sarà evidenziato.

Il parametro *case_insens* posto ad '1' indica che il pattern è non case sensitive.

Il parametro *is_regex* posto ad '1' indica che il pattern è descritto mediante un'espressione regolare.

```
bf.attribute.highlight =
    attribute highlight { xsd:IDREF }?

bf.attribute.attribhighlight =
    attribute attribhighlight { xsd:IDREF }?

bf.attribute.case_insens =
    attribute case_insens { xsd:boolean }?

bf.attribute.is_regex =
    attribute is_regex { xsd:boolean }?
```

3.3 Descrizione del linguaggio XML mediante bflang

Riporto di seguito a titolo esemplificativo la descrizione mediante bflang del linguaggio XML. Questa descrizione è riportata dal sorgente xml.bflang2 del-

l'editor ed è perciò la descrizione usata nella versione attuale.

```
<bflang name="XML" version="2.0" table="60" contexts="8" matches="17">
<header>
  <mime type="text/xml"/>
  <mime type="application/xml"/>
  <option name="show_in_menu" default="1"/>
  <highlight name="assignment" style="brackets" />
  <highlight name="attribute" style="attribute" />
  <highlight name="comment" style="comment" />
  <highlight name="entity" style="value" />
  <highlight name="string" style="string" />
  <highlight name="tag" style="tag" />
  <highlight name="xml-declaration" style="preprocessor" />
</header>
<properties>
  <comment id="cm.htmlcomment" type="block"
               start="&lt;!--" end="--&gt;"/>
  <default_spellcheck enabled="1" spell_decode_entities="1" />
</properties>
<definition>
<context symbols="&gt;&lt;&amp;;␣&#9;&#10;&#13;-"
        commentid_block="cm.htmlcomment" commentid_line="none">
  <element id="e.xml.open" pattern="&lt;?xml"
           highlight="xml-declaration" starts_block="1">
  <context symbols="&gt;&amp;;&#34;'␣&#9;&#10;&#13;=">
        <element pattern="[a-zA-Z0-9:-]+" is_regex="1"
                highlight="xml-declaration" />
        <element pattern="?&gt;" ends_context="1" ends_block="1"
        blockstartelement="e.xml.open" highlight="xml-declaration" />
        <element id="e.xml.doublequote" pattern="&#34;"
                                        highlight="string">
          <context symbols="&amp;;&#34;" highlight="string" >
                <element id="e.xml.entity" pattern="&amp;[a-z0-9#]+;"
            is_regex="1" highlight="entity" />
                <element pattern="&#34;" highlight="string"
                                           ends_context="1"/>
          </context>
    </element>
        <element id="e.xml.singlequote" pattern="'" highlight="string">
          <context symbols="&amp;;\'" highlight="string" >
                <element idref="e.xml.entity"/>
                <element pattern="'" highlight="string"
                                        ends_context="1"/>
          </context>
```

```
      </element>
44  </context>
    </element>
46  <element id="e.xml.lcomment" pattern="&lt;!--"
                              highlight="comment" starts_block="1">
48        <context symbols="-&gt;␣&#9;&#10;&#13;" highlight="comment">
            <element pattern="--&gt;" ends_block="1"
50            blockstartelement="e.xml.lcomment" highlight="comment"
              mayfold="1" ends_context="1" />
52        </context>
    </element>
54  <element id="e.xml.tag.open" pattern="&lt;[_a-zA-Z0-9:-]+"
              is_regex="1" highlight="tag" starts_block="1"
56            tagclose_from_blockstack="1">
          <context symbols="&gt;&amp;;&#34;'␣&#9;&#10;&#13;=">
58          <element pattern="[_a-zA-Z0-9:-]+"
              is_regex="1" highlight="attribute" />
60          <element pattern="=" highlight="assignment" />
            <element idref="e.xml.doublequote"/>
62          <element idref="e.xml.singlequote"/>
            <element pattern="/&gt;" highlight="tag" ends_context="1"
64            ends_block="1" blockstartelement="e.xml.tag.open" />
            <element pattern="&gt;" highlight="tag"
66        blockstartelement="e.xml.tag.open" stretch_blockstart="1">
                  <context symbols="&gt;&lt;&amp;;␣&#9;&#10;&#13;">
68                  <element pattern="&lt;/[_a-zA-Z0-9:-]+&gt;"
               is_regex="1" highlight="tag" ends_context="2"
70             ends_block="1" blockstartelement="e.xml.tag.open"/>
                      <element idref="e.xml.tag.open"/>
72          <element idref="e.xml.lcomment"/>
                      <element idref="e.xml.entity"/>
74                </context>
            </element>
76        </context>
    </element>
78  <element idref="e.xml.entity" />
</context>
80 </definition>
</bflang>
```

Capitolo 4

Dalla sintassi al file .bflang2

In questo capitolo verrà mostrato come è stato definito il linguaggio NXT-GTD mediante il *Bluefish Language* (bflang2) seguendo le regole indicate dal relativo Schema XML.

4.1 Header

Nel tag **bflang**, radice del documento, vengono definiti il nome visualizzato nell'editor (NXT-GTD) e la versione del linguaggio che, essendo questa la sua prima definizione ha il numero 1.0. Per quanto riguarda la scelta del tipo di file, attraverso il tag **mime**, si è definito un tipo personalizzato in modo da poi associarvi un'estensione creata appositamente per file contenenti codice GTD (.nxtd). Affinché tale associazione sia valida è necessario seguire le istruzioni indicate nel capitolo successivo. Oltre ad esso viene inserito anche il mime type relativo al testo puro (text/plain) in modo che Bluefish apra il file di definizione anche in sistemi nei quali non è stato definito il mime type nxtd. In questi ultimi, all'apertura del file, il linguaggio non verrà caricato in automatico ma dovrà essere selezionato manualmente dall'utente.

Le opzioni sono state "visualmente" divise tra standard e le altre (l'editor non fa questa distinzione) poichè le prime riguardano specifiche proprie dell'editor e possono essere comuni a tutti i linguaggi per Bluefish mentre le seconde sono definite dal programmatore. Partiamo da quelle personali, esse sono due abilitazioni dei gruppi di elementi *keyword* e *type* definiti successivamente. Le opzioni standard sono anch'esse delle abilitazioni a riguardo di:

- caricamento dei riferimenti e delle descrizioni definite con il tag *reference* secondo le impostazioni definite dalla finestra di dialogo dell'editor (*load_reference*)
- funzione di suggerimento per il completamento delle istruzioni e visualizzazione del box con l'elenco delle istruzioni (*load_completion*)
- funzione di suggerimento per il completamento delle funzioni definite mediante i due punti (:) (*autocompl_with_semicolon*)
- contrazione del codice raggruppato tra parentesi graffe, quadre e tonde (*Parentheses block_foldable,Square brackets block_foldable,Curly brackets block_foldable*)
- visualizzazione del linguaggio nell'elenco dell'editor (e quindi possibilità di usarlo)

Nei tag **highlight** è possibile definire un nome personale (*name*) per far riferimento ai tipi di evidenziature definiti nell'editor(*style*).

```
<?xml version="1.0"?>

<bflang name="NXT-GTD" version="1.0">
<header>
    <mime type="text/x-nxtd"/>
        <mime type="text/plain" />

        <!-- Opzioni standard -->
        <option name="load_reference" default="1"
            description="Load␣reference␣data"/>
        <option name="load_completion" default="1"
            description="Load␣completion␣data"/>
        <option name="autocompl_with_semicolon" default="1"
            description="Autocomplete␣function␣names␣with␣semicolon"/>
        <option name="Parentheses␣block_foldable" default="1"
            description="Allow␣folding␣of␣Parentheses␣block"/>
        <option name="Square␣brackets␣block_foldable" default="1"
            description="Allow␣folding␣of␣Square␣brackets␣block"/>
        <option name="Curly␣brackets␣block_foldable" default="1"
            description="Allow␣folding␣of␣Curly␣brackets␣block"/>
        <option name="show_in_menu" default="1" />

        <!-- opzioni gruppi -->
```

```
        <option name="keyword" default="1" />
        <option name="type" default="1" />

        <!-- Associazione degli stili ai tipi di parole chiave -->
        <highlight name="error" style="warning" />
        <highlight name="keyword" style="keyword" />
        <highlight name="bracket-square" style="brackets" />
        <highlight name="bracket-round" style="brackets" />
        <highlight name="bracket-curly" style="brackets" />
        <highlight name="type" style="keyword" />
        <highlight name="line-comment" style="comment" />
        <highlight name="block-comment" style="comment" />
        <highlight name="string" style="string" />
        <highlight name="value" style="value" />
        <highlight name="preprocessor" style="preprocessor" />
        <highlight name="skeyword" style="special-keyword" />
</header>
```

4.2 Properties

Il tag **comment** è stato usato per definire i simboli di apertura e chiusura di un commento in blocco. I tag **smartindent** e **smartoutdent** sono stati definiti per le sole parentesi tonde.

```
<properties>
        <comment type="block" start="/*" end="*/" />
        <smartindent characters="(" />
        <smartoutdent characters=")" />
</properties>
```

4.3 Definition

In questa sezione vi è la descrizione vera e propria del linguaggio. L'NXT-GTD, tranne qualche caso, è costituito da elementi molto simili per quanto riguarda la struttura, infatti ciascuno è composto da:

- un nome del blocco che rappresenta l'azione che il robot deve svolgere
- una serie di attributi costituiti dalla coppia nome = valore

- eventuali commenti

Si è scelto di usare l'evidenziatura "keyword" per i nomi dei comandi, l'evidenziatura "type" per i nomi dei parametri e l'evidenziatura "value" per i valori assegnati ai comandi e per i numeri. Abbiamo già visto nell'header l'intenzione di suddividere i comandi in due gruppi: *keyword* e *type*. Mediante il tag **group**, con una logica associazione di significati, verrà fatto un raggruppamento dei tag che rappresentano i nomi dei comandi e uno dei loro attributi.

Viene riportato di seguito la definizione di un comando d'esempio poiché, tutti gli altri si possono ottenere in modo simile sostituendo i nomi delle variabili. Il comando scelto d'esempio è il comando *Sposta* del quale nel paragrafo 1.4 è presente la descrizione letterale. Il codice bflang è il seguente:

```
<definition>
 <context symbols="␣;(){}[]:\&#34;\\',&gt;&lt;*&amp;^%!+=-|/?#&#9;&#10;&#13;">
   <group  highlight="keyword" >
        <autocomplete enable="1" append="(" />
    <element pattern="Sposta" id="idSposta">
      <reference>
      <b>Parametri accettati:</b>
      Porte, Dir, StSin, StDes, Sterza, Pot, Dur, ProsAz
      </reference>
      <context symbols="␣()=.,&#10;&#13;" idref="idSposta">

           <group highlight="type" is_regex="1">
                <autocomplete enable="1" append="␣=␣"/>
                       <element pattern="Porte" />
                       <element pattern="Dir" />
                       <element pattern="StSin" />
                       <element pattern="StDes" />
                       <element pattern="Sterza" />
            <element pattern="Pot" />
                       <element pattern="ProsAz" />
                       <element pattern="Dur" />
                       <element pattern="MotoreSin" />
                       <element pattern="MotoreDes" />
                       <element pattern="MotoreAlt" />
               </group>

        <group highlight="value" is_regex="1">
                <autocomplete enable="1" />
                       <element pattern="A" />
```

```
                        <element pattern="B" />
                        <element pattern="C" />
                        <element pattern="AB" />
                        <element pattern="AC" />
                        <element pattern="BC" />
                        <element pattern="ABC" />
                        <element pattern="AVA" />
                        <element pattern="IND" />
                        <element pattern="STOP" />
                        <element pattern="ILLIM" />
                        <element pattern="GRA" />
                        <element pattern="ROT" />
                        <element pattern="s" />
                        <element pattern="FRENA" />
                        <element pattern="FOLLE" />
                        <element pattern="VERO" />
                        <element pattern="FALSO" />
                </group>

            <!-- commenti -->
                <element idref="e.comment" />
        <!-- punteggiature -->
                <element idref="e.number" />
                <element idref="e.doublestring" />
                <element idref="e.singlestring" />
                <element pattern=")" ends_context="1" />
          </context>
         </element>
```

Il tag **autocomplete** viene usato per far aggiungere in automatico la parentesi tonda aperta dopo la scrittura del nome del comando ed il simbolo di uguale dopo la scrittura di un parametro. È possibile far aggiungere un qualsiasi testo si voglia, e nel caso si debbano inserire degli ulteriori dati in mezzo alla stringa aggiunta si può usare l'attributo *backup_cursor* per far tornare indietro il cursore in automatico. Ad esempio, supponendo di voler velocizzare la scrittura di una funzione possiamo specificare gli attributi

```
append = "(Parametro1 = , Parametro2 = )" backup_cursor = "16"
```

sicché il cursore si porta dopo il primo uguale velocizzando di molto la scrittura, in particolar modo per un utente esperto.

Il tag **reference** è stato usato in questo caso per suggerire i parametri che il comando accetta. Questo tipo di suggerimento viene visualizzato durante la digitazione ed è diverso dal suggerimento dei comandi attivabile alla pressione

del primo carattere, poiché esso viene popolato in automatico con gli attributi *pattern* di tutti gli elementi definiti.

Particolare attenzione bisogna prestare all'inserimento di un ulteriore tag **context** all'interno dell'**element** che definisce il nome del comando. Esso ci permette di fare in modo che gli elementi definiti all'interno di *Sposta* non vengano riconosciuti all'esterno delle parentesi del comando stesso. Poiché essi non sono riconosciuti all'esterno del blocco, non sono nemmeno evidenziati, fornendoci un test visuale sulla correttezza della posizione di tali elementi. Vi è però un aspetto negativo in questa tecnica, che consiste nel dover definire elementi comuni a più comandi più volte, una per ciascuno di essi. Nel caso di elementi la cui definizione non è banale, la ripetizione della definizione appesantirebbe molto il codice, problema a cui si può ovviare facendo l'uso di riferimenti ai comandi, mediante *idref*. Nell'esempio di *Sposta* vi sono quattro riferimenti ad elementi che definiscono: la sintassi per i commenti, i numeri, e le stringhe. Il codice di definizione di tali elementi è il seguente:

```
<!-- commenti -->
    <element id="e.comment" pattern="--[^&#10;&#13;]*"
             highlight="line-comment" is_regex="1" />

<!-- numeri -->
    <element id="e.number" pattern="[0-9.]+" is_regex="1" highlight="value"/>

<!-- stringhe fra doppio apice -->
     <element id="e.doublestring" pattern="&#34;" highlight="string">
        <context symbols="\&#34;nrt" highlight="string">
                <element pattern="\\." is_regex="1" highlight="string" />
                <element pattern="&#34;" highlight="string" ends_context="1" />
        </context>
     </element>

 <!-- stringhe fra singolo apice -->
    <element id="e.singlestring" pattern="'" highlight="string">
        <context symbols="\'" highlight="string" >
                <element pattern="\\." is_regex="1" highlight="string" />
                <element pattern="'" highlight="string" ends_context="1" />
        </context>
     </element>
```

Questi quattro elementi sono posizionati nel codice all'interno del context più esterno contenuto nel tag *definition* e vengono richiamati nella definizione di

ogni comando.

Per questo difetto nel modo di descrivere un comando è stato scelto di utilizzare una politica diversa nella definizione del comando *MioBlocco*. Esso permette di definire un blocco personalizzato specificandone il nome, i nomi dei parametri e i loro tipi, e le azioni da compiere (usando altri comandi del linguaggio GTD). Poiché qualsiasi comando può essere usato al suo interno, usando la stessa tecnica di *Sposta* sarebbe necessario elencare nel context i riferimenti per tutti i comandi dell'intero linguaggio. Si è perciò scelto di definire gli attributi di *MioBlocco* come elementi singoli e slegati dal comando stesso.

```
<element pattern="ImpUsc" highlight="skeyword" is_regex="1" >
  <autocomplete enable="1"
                append="(␣Nome␣=␣,␣Val␣=␣)"
                backup_cursor="9" >
  </autocomplete>
  <reference>
        <b>Nome</b> segue la struttura:
        Nome = &lt;nome del parametro&gt;.[LOGICA|NUM|TESTO]
  </reference>
</element>

<group is_regex="1">
  <autocomplete enable="1" append="␣=␣SI"/>
  <element pattern="Nome" highlight="type" />
  <element pattern="ParIn" highlight="skeyword" />
  <element pattern="ParUsc" highlight="skeyword" />
  <element pattern="Val" highlight="type" />
</group>

<element pattern="NUM" highlight="value" is_regex="1" />
<element pattern="LOGICA" highlight="value" is_regex="1" />
```

Altri elementi con una struttura differente da quella descritta per il comando *Sposta* sono le parentesi, per le quali va definita la funzione di raggruppamento del codice al loro interno come un unico blocco, e il riconoscimento delle etichette. Si noti l'uso di espressioni regolari nel campo *pattern* per il riconoscimento delle etichette, alla loro definizione (label:) e al loro utilizzo (label.VERO, label.SE, ...). Il codice usato è il seguente:

```
 <element id="e.lbrace" pattern="{"
```

```
            starts_block="1"
            highlight="bracket-curly"
            block_name="Curly␣brackets␣block"/>
  <element  pattern="}"
            ends_block="1"
            blockstartelement="e.lbrace"
            highlight="bracket-curly" />

  <element  id="e.lbracket" pattern="["
            starts_block="1"
            highlight="bracket-square"
            block_name="Square␣brackets␣block" />
  <element  pattern="]"
            ends_block="1"
            blockstartelement="e.lbracket"
            highlight="bracket-square" />

  <element  id="e.lparen" pattern="("
            starts_block="1"
            highlight="bracket-round"
            block_name="Parentheses␣block" />
  <element  pattern=")"
            ends_block="1"
            blockstartelement="e.lparen"
            highlight="bracket-round" />

  <element  pattern="[A-Za-z0-9]+\:"
              highlight="tag" is_regex="1" />
  <element  pattern="[A-Za-z0-9]+\.SE"
              highlight="special-tag" is_regex="1" />
  <element  pattern="[A-Za-z0-9]+\.ELSE"
              highlight="special-tag" is_regex="1" />
  <element  pattern="[A-Za-z0-9]+\.VERO"
              highlight="special-tag" is_regex="1" />
  <element  pattern="[A-Za-z0-9]+\.FALSO"
              highlight="special-tag" is_regex="1" />
```

Come si può notare i tag utilizzabili per la descrizione di una sintassi mediante bflang non sono molti, e la loro combinazione è molto intuitiva. Le sintassi con cui si può trovare maggiore difficoltà, sono quelle in cui le parole chiave sono strutturate in particolari gerarchie non fisse i cui elementi sono interscambiabili (ad esempio HTML). In questi casi la definizione è comunque possibile ma risulta meno immediata che nella descrizione di linguaggi le cui parole chiave, anche se molte, si presentano quasi sempre isolate o senza

specifici collegamenti alle altre (ad esempio C).

Capitolo 5

Estensione dell'editor

Una volta creato il file *nxt-gtd.bflang2* bisogna fare in modo che Bluefish lo riconosca e lo carichi assieme agli altri linguaggi. Per far ciò è sufficiente copiare il file di definizione nella cartella ***bluefish***, che è il percorso da cui l'editor prende i file dei linguaggi personalizzati. Se un file porta lo stesso nome di uno di quelli standard (salvato nella cartella *bflang*), esso viene utilizzato al posto di quello originale senza sovrascriverlo.

Sistemi Windows

In caso di sistemi operativi Windows bisogna per prima cosa trovare la cartella *Bluefish*, la quale si trova dove ci sono tutte le installazioni dei programmi. Alcuni path possibili sono:

```
C:\Program Files\Bluefish}
C:\Programmi\Bluefush
C:\Programmi(x86)\Bluefish
```

Trovata la cartella principale, il file di definizione va copiato nel percorso: */.bluefish/*.

Per associare Bluefish ad un file con estensione *nxtd* è sufficiente seguire i seguenti semplici passi:

1. clic con il tasto destro del mouse sul file *.nxtd
2. premere *Apri con...*
3. Selezionare Bluefish dalla lista di programmi (eventualmente cercarlo tramite il tasto *Sfoglia...*

4. Spuntare la casella *Usa sempre questo programma* e premere OK.

Da questo momento in poi verrà usato Bluefish come applicazione predefinita per i file di tipo nxtd. All'apertura, sarà sufficiente selezionare la modalità di linguaggio corretta per poter lavorare agevolmente sul file.

Sistemi Linux

Per sistemi operativi linux, il file di definizione va copiato nella cartella: */.bluefish/*. Per far si che Bluefish riconosca l'estensione nxtd è necessario aggiungere il mime type: *text/x-nxtd*. Per far ciò bisogna modificare il file *HOME/.local/share/mime/packages/Override.xml* o crearlo nel caso esso non sia già esistente (è necessario l'accesso come root). Una volta aperto esso avrà la seguente forma:

```
<?xml version="1.0" encoding="UTF-8"?>
<mime-info xmlns="http://www.freedesktop.org/standards/shared-mime-info">
  <mime-type type="... MIME type ...">
   <comment> ... comment ... </comment>
   <glob pattern="*..."/>
   <icon name="...name...">
  </mime-type>
  <mime-type type="... another MIME type ...">
    ... other child elements ...
  </mime-type>
   .... etc.
</mime-info>
```

quello che dobbiamo fare è aggiungere il seguente codice all'interno del tag **mime-info**:

```
<mime-type type="text/x-nxtd">
 <glob pattern="*nxtd"/>
</mime-type>
```

e dopo aver salvato il file, eseguire da terminale il comando:

```
update-mime-database ~/.local/share/mime
```

Bluefish caricherà in automatico tutti i linguaggi presenti nella cartella bflang e li renderà disponibili per il programmatore. Per rendere effettive le modifiche è necessario riavviare il programma.

Una volta aperto l'editor è possibile selezionare il linguaggio dal menù: Documento->Modalità linguaggio->NXT-GTD.
L'evidenziazione delle parole segue alcuni stili predefiniti i quali possono essere personalizzati dalle preferenze dell'editor (*Modifica->Preferenze->Stili di testo*). Comparirà una finestra come quella di figura 5 dalla quale sarà possibile definire colore, peso, sfondo e stile dell'evidenziazione. Mediante il tasto *aggiungi voce* è inoltre possibile definire degli stili personalizzati senza dover modificare quelli di default.

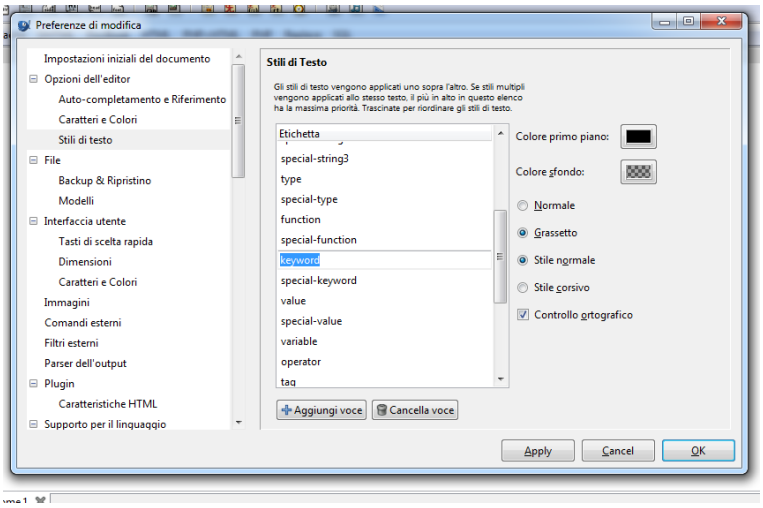


Figura 5.1: Opzioni degli stili di evidenziazione.

Selezionando la voce *supporto per il linguaggio* dal menù a sinistra, è possibile modificare in modo visuale i valori dei tag **option** (nel riquadro superiore a destra) e **highlight** (nel riquadro in basso a destra) definiti nella sezione *header* del codice bflang.

Nella figura 5 è possibile vedere l'effetto dell'evidenziazione su di un esempio di codice.

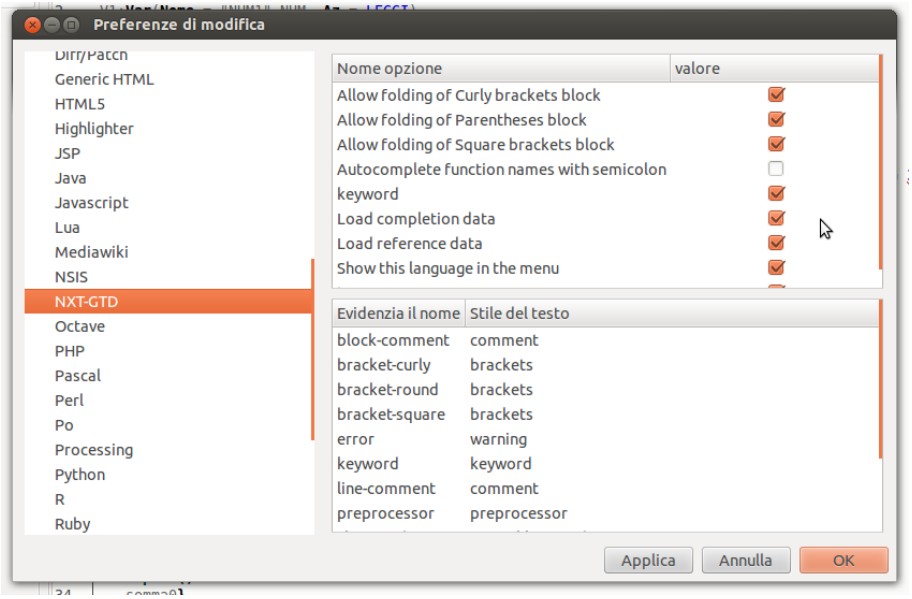


Figura 5.2: Opzioni del linguaggio.

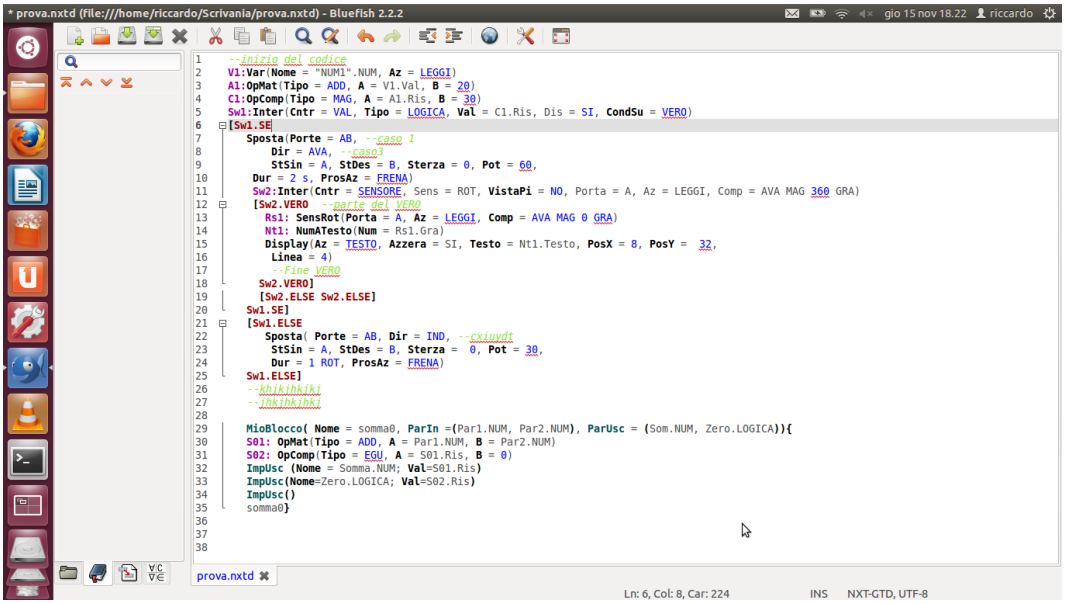


Figura 5.3: Esempio di codice su Bluefish Ubuntu.

Conclusioni

Bluefish, con la sua portabilità e la sua funzionalità risulta essere un buon editor di testo, che cerca di andare incontro alle esigenze del programmatore. La semplicità e l'immediatezza con cui è possibile estendere l'editor ne fanno lo strumento più adatto per chi vuole uno strumento da usare con un linguaggio personale non usato nel panorama informatico.

Il linguaggio NXT-GTD ha una struttura abbastanza diversa dai linguaggi di programmazione più consoni sulla base dei quali è stato sviluppato bflang. Nonostante questo, bflang risulta essere adeguato ad avere una descrizione buona del linguaggio e permette diversi livelli di specificità a seconda di ciò che più preme allo sviluppatore. Sarebbe stato possibile creare una definizione più basilare basata sulla semplice elencazione delle parole chiave che però non avrebbe fornito alcuno strumento di limitazione degli errori (è importante però ricordare che la semplice evidenziazione del codice non assicura la correttezza dello stesso).

Limiti di Bluefish

Sarebbe stato comodo poter implementare un metodo che riconoscesse le etichette uguali e le evidenziasse nel testo in modo differente, indicando il punto di definizione della stessa (il punto in cui è stata usata la prima volta). Bluefish però (a questa versione) non fornisce strumenti che ne permettano l'implementazione e le etichette non vengono quindi distinte tra loro.

Bibliografia

[1] Dimitris Alimisis. Teacher Education on Robotics-Enhanced Constructivist Pedagogical Methods. ASPETE, 2009.

[2] Paolo Pialorsi. XML. Il nuovo linguaggio del web. Mondadori Informatica, 2002.

[3] Harold Elliotte Rusty, Scott W. Means. XML Guida di riferimento. Apogeo.

Siti internet

- bluefish.openoffice.nl
- bluefish.openoffice.nl/ns/bflang/2.0/schema.html
- blog.gmane.org
- w3.org/XML/Schema
- html.it/xml
- mindstorms.lego.com