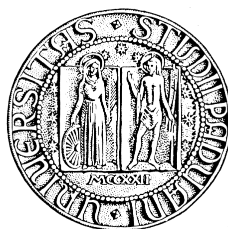

UNIVERSITÀ DEGLI STUDI DI PADOVA

Facoltà di Ingegneria

Tesina

IL LINGUAGGIO DI PROGRAMMAZIONE GO

Laureando: Fabrizio Tonus

Relatore: Prof. Michele Moro

Corso di laurea in Ingegneria Informatica

Data Laurea: 27 settembre 2011

Anno Accademico 2010/2011

Indice generale

Sommario....7
1 Introduzione....8
  1.1 Cause scatenanti l'inizio del progetto Go....9
  1.2 Gli antenati di Go....10
  1.3 Mascotte di Go....10
2 Obiettivi del linguaggio....11
  2.1 Caratteristiche....11
  2.2 Aspettative....12
3 Ambito applicativo....13
  3.1 Installazione....13
  3.2 Stato sperimentale....13
  3.3 Utilizzi applicativi e possibili sviluppi....13
4 Specifiche del linguaggio....14
  4.1 Introduzione....14
  4.2 Notazione....14
  4.3 Rappresentazione del codice sorgente....14
    4.3.1 Caratteri 15
    4.3.2 Lettere e cifre 15
  4.4 Elementi lessicali 15
    4.4.1 Commenti 15
    4.4.2 Tokens 15
      4.4.2.1 Identificatori 16
      4.4.2.2 Parole chiave 16
      4.4.2.3 Operatori e delimitatori 16
    4.4.3 Punti e virgola 16
    4.4.4 Valori interi....17
    4.4.5 Valori in virgola mobile 17
    4.4.6 Valori immaginari....17
    4.4.7 Valori carattere....18
    4.4.8 Valori stringa....19
  4.5 Costanti 20
  4.6 Tipi 20
    4.6.1 Metodo set....21
    4.6.2 Tipi booleani....21
    4.6.3 Tipi numerici 21
    4.6.4 Tipi stringa....22
    4.6.5 Tipi array 22
    4.6.6 Tipi slice 22
    4.6.7 Tipi struttura....23
    4.6.8 Tipi puntatore....24
    4.6.9 Tipi funzione 24
    4.6.10 Tipi interfaccia 25
    4.6.11 Tipi mappa 26
    4.6.12 Tipi canale 26
  4.7 Proprietà dei tipi e dei valori 27
    4.7.1 Tipi identità 27
    4.7.2 Assegnabilità 28
  4.8 I blocchi 28
  4.9 Dichiarazioni e campi di applicazione....29

4.9.1 Etichette di visibilità 29
4.9.2 Identificatori prefissati 29
4.9.3 Identificatori esportati 30
4.9.4 Identificatore blank 30
4.9.5 Dichiarazioni di costanti 30
4.9.6 Iota 31
4.9.7 Dichiarazioni di tipo 32
4.9.8 Dichiarazioni di variabile 33
4.9.9 Dichiarazioni di variabili short 33
4.9.10 Dichiarazioni di funzioni 34
4.9.11 Dichiarazioni di metodo 34
4.10 Espressioni 35
4.10.1 Operandi 35
4.10.2 Identificatori qualificati 35
4.10.3 Valori composti 35
4.10.4 Valori di funzione 37
4.10.5 Espressioni principali 37
4.10.6 Selettori 38
4.10.7 Indici 39
4.10.8 Slice 39
4.10.9 Tipo asserzioni 40
4.10.10 Chiamate 40
4.10.11 Argomenti passati ai parametri 41
4.10.12 Operatori 41
4.10.12.1 Operatore di precedenza 42
4.10.12.2 Operatori aritmetici 42
4.10.12.3 Overflow intero 43
4.10.12.4 Operatori di confronto 44
4.10.12.5 Operatori logici 44
4.10.12.6 Operatori indirizzo 44
4.10.13 Operatore receive 45
4.10.14 Espressioni di metodo 45
4.10.15 Le conversioni 46
4.10.15.1 Conversioni tra tipi numerici 46
4.10.15.2 Conversioni a e da un tipo stringa 47
4.10.16 Espressioni costanti 47
4.10.17 Ordine di valutazione 48
4.11 Istruzioni 49
4.11.1 Istruzioni vuote 49
4.11.2 Istruzioni etichettate 49
4.11.3 Istruzioni espressioni 49
4.11.4 Istruzioni send 49
4.11.5 Istruzioni incremento-decremento 50
4.11.6 Assegnazioni 50
4.11.7 Istruzioni If 51
4.11.8 Istruzioni switch 51
4.11.8.1 Espressioni switch 51
4.11.8.2 Tipi switch 52
4.11.9 Istruzioni for 53
4.11.10 Istruzioni Go 55

4.11.11 Istruzioni select......55
4.11.12 Istruzioni return......56
4.11.13 Istruzioni break......57
4.11.14 Istruzioni continue......57
4.11.15 Istruzioni goto......57
4.11.16 Istruzioni fallthrough......57
4.11.17 Istruzioni defer......57
4.12 Funzioni built-in......58
4.12.1 Close e closed......58
4.12.2 Lunghezza e capacità......58
4.12.3 Allocazione......59
4.12.4 Creazione di slices, mappe e canali......59
4.12.5 Aggiunta e copia di slice......59
4.12.6 Assemblaggio e disassemblaggio di numeri complessi......60
4.12.7 Trattamento panici......60
4.12.8 Bootstrapping......61
4.13 I Package......61
4.13.1 Organizzazione del file sorgente......61
4.13.2 Clausola package......62
4.13.3 Importazione dichiarazioni......62
4.13.4 Un esempio di package......63
4.14 Inizializzazione ed esecuzione del programma......63
4.14.1 Il valore zero......63
4.14.2 Esecuzione del programma......64
4.15 Panici nel tempo di esecuzione......65
4.16 Considerazioni di sistema......65
4.16.1 Package unsafe......65
4.16.2 Garanzie della dimensione e dell'allineamento......66
5 Costrutti significativi......67
5.1 Tokens......67
5.1.1 Identificatori......67
5.1.2 Parole chiave......67
5.1.3 Operatori e delimitatori......67
5.2 Punti e virgola......68
5.3 Valori stringa......68
5.4 Slice......69
5.5 Mappa......70
5.6 Canale......71
5.7 Goroutines......71
6 Applicazioni significative......73
6.1 GO Launcher Ex......73
6.2 GO SMS Pro......74
6.3 GO Weather......75
6.4 GO Contacts......76
6.5 GO Keyboard......77
6.6 GO Score......77
6.7 GO Wallpaper......78
Conclusioni......79
Bibliografia......80

Indice delle figure

Illustrazione 1.1: logo Google....8
Illustrazione 1.2: logo Google Go....8
Illustrazione 1.3: mascotte di Google Go....8
Illustrazione 1.1.1: Google Team....9
Illustrazione 6.1: logo Go Dev Team....73
Illustrazione 6.1.1: immagine di Go Launcher Ex....73
Illustrazione 6.2.1: immagine di Go Sms Pro....74
Illustrazione 6.3.1: immagine di Go Weather....75
Illustrazione 6.4.1: immagine di Go Contacts....76
Illustrazione 6.5.1: immagine di Go Keyboard....77
Illustrazione 6.6.1: immagine di Go Score....77
Illustrazione 6.7.1: un wallpaper di Sakura Falling....78

Sommario

La seguente tesi tratterà un'analisi sul linguaggio di programmazione Go.
Go è stato sviluppato da Rob Pike, ingegnere software di casa Google, con l'obiettivo di creare un linguaggio di programmazione che soddisfi le esigenze di una compilazione efficiente, di un'esecuzione veloce e di una facilità di programmazione. Le cause scatenanti la nascita di questo nuovo linguaggio sono lo sviluppo software fermo da una decina d'anni, il garbage collection, il parallelismo e i processori multi-core che non sono ben supportati.
Le caratteristiche di questo linguaggio sono velocità, semplicità, similarità al C, concorrenza, multi-core, multi-processing, sicurezza, gestione della memoria basata sulla garbage collection.
La velocità di compilazione e di esecuzione è dovuta al componente essenziale di ogni programma Go: la Goroutine. Go ha una sintassi semplice paragonabile a quella di un C misto a Python, permette la concorrenza e, insieme alla Goroutine, si possono scrivere programmi che sfruttano il parallelismo.
Con le caratteristiche che ci sono, sembra proprio che i laboratori di Google abbiano preso ispirazione dalle difficoltà dei programmatori C, dalla lentezza dei linguaggi interpretati e dai limiti di performance delle macchine virtuali in circolazione, come Java e .net.
Per ora il linguaggio Go è in uno stato sperimentale e le sue applicazioni sono compatibili con il sistema operativo Android di proprietà Google.
Di seguito è presente una guida sulle specifiche di Go tradotta dal sito ufficiale del linguaggio.
Seguono una descrizione dettagliata di alcuni costrutti non presenti nel linguaggio C quali i tokens, i valori stringa, le slice, le mappe, i canali e infine le Goroutines.
Applicazioni significative del linguaggio sono sviluppate dal Go Dev Team per dispositivi “mobile” muniti di sistema operativo Android. Le applicazioni di rilievo sono Go Launcher Ex, Go Sms Pro, Go Weather, Go Contacts, Go Keyboard, Go Score, Go Wallpaper.

1 Introduzione

«Hey! Ho! Let's Go!» cantavano i Ramones, «Hey! Ho! Let's Go!» si intitola il post con il quale Google annuncia ufficialmente l'11 novembre 2009 la nascita di un nuovo linguaggio di programmazione open source, denominato proprio Go.

Illustrazione 1.1: logo Google

"Go prova a combinare la velocità di sviluppo nel lavorare con un linguaggio dinamico come Python con le performance e la sicurezza di un linguaggio compilato come C o C++" spiega Google presentando la sua nuova creatura, con l'obiettivo ultimo di ottenere un codice compilato con le stesse prestazioni possibili con C contemporaneamente a una predisposizione naturale a girare al meglio su sistemi a più core.
"Abbiamo sviluppato Go perché eravamo un po' frustrati da quanto lo sviluppo software fosse divenuto complicato negli ultimi 10 anni" dice Rob Pike, l'ingegnere software di Mountain View.

Sul sito ufficiale il linguaggio viene definito "semplice, veloce, concorrente e divertente".
Tutti ci saremmo aspettati un nuovo linguaggio riscritto quasi totalmente e comunque innovativo rispetto a quello che si trova sulla piazza invece l'innovazione forse sta nel tornare indietro, almeno sarà questo che avranno pensato a casa Google.

Illustrazione 1.2: logo Google Go

Si tratta del secondo esperimento di Google in questo settore, poiché a luglio 2009 la società aveva già rilasciato Simple, un dialetto Basic per lo sviluppo di applicazioni per Android.
A Google, si sa, sta particolarmente a cuore la velocità di esecuzione di appliance e programmi: e per tale motivo la società non lesina sulla distribuzione di strumenti di sviluppo ad alto tasso di ottimizzazione. Dalla realizzazione di tool specifici a un vero e proprio linguaggio di programmazione il passo può essere breve, e infatti Go sta lì a dimostrare gli sforzi compiuti da Google durante i due anni precedenti l'uscita nel tentativo di rendere più facile la vita agli sviluppatori.

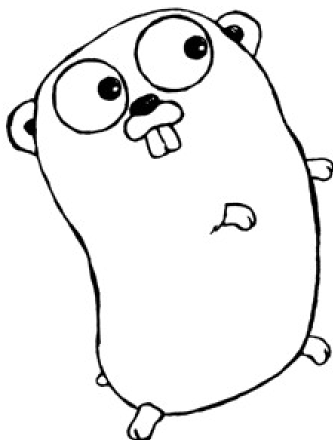

Illustrazione 1.3: mascotte di Google Go

Forse non si sentiva la necessità di questa nuova uscita di Google, i linguaggi di programmazione sono un campo delicato dove è molto difficile imporsi in modo semplice, certo è che Google ha dalla sua un grande bacino di utenza che usa costantemente i suoi prodotti e sviluppa soluzioni intimamente legate ai propri servizi, magari questa potrebbe essere la chiave di volta per permettere alla grande G di affermare sul mercato quello che ad oggi sembra un mero esperimento.
Inizialmente sottovalutato e perfino ridicolizzato, Google Go è stato nominato da TIOBE il linguaggio dell'anno (2009) in forza della rapida crescita fatta segnare nei mesi appena successivi l'uscita. TIOBE stila su base mensile una classifica che intende fotografare l'interesse destato dai vari linguaggi di programmazione evidenziando quelli che sono, a livello mondiale, maggiormente utilizzati. In testa (dati aggiornati a marzo 2010) c'è Java (17,5%), seguito da C (17,3%) e PHP

(9,9%). In quarta posizione C++ (9,6%) mentre in quinta c'è Visual Basic (6,6%). Come si sottolinea dalla stessa società, la classifica pubblicata da TIOBE non intende assolutamente indicare quale sia il migliore linguaggio di programmazione ma semplicemente con quale linguaggio è stato scritto il maggior numero di righe di codice.
Google Go si pone in tredicesima posizione con appena l'1% ma TIOBE ha voluto sottolineare la crescita "esponenziale" in termini di utilizzo di questo linguaggio. Go è infatti cresciuto in silenzio giungendo quasi alla maturità.
Google ha affrontato qualche problema per il nome scelto, in quanto esiste (con i primi documenti risalenti al 2003) un linguaggio di programmazione chiamato Go! creato da McCabe e Keith Clark, il primo dei quali ha pubblicamente espresso il proprio disappunto.

1.1 Cause scatenanti l'inizio del progetto Go

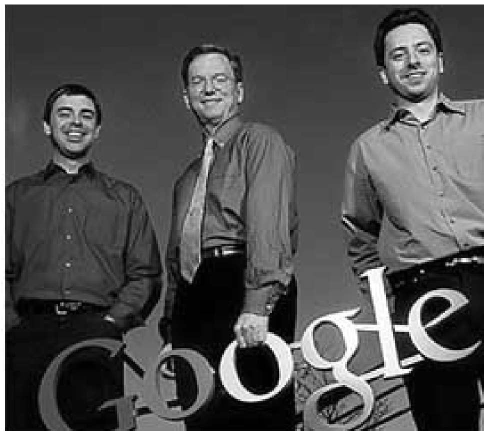

Illustrazione 1.1.1: Google Team

Secondo gli sviluppatori, l'esigenza di creare un nuovo linguaggio di programmazione nasce dal fatto che attualmente non esiste un linguaggio di programmazione che soddisfi le esigenze di una compilazione efficiente, di una esecuzione veloce e di una facilità di programmazione.
I programmatori che potevano, sceglievano la facilità rispetto alla sicurezza e all'efficienza passando a linguaggi tipicamente dinamici come Python e JavaScript piuttosto che C++ o, in misura minore, Java.
La domanda che molti si pongono è: perché un altro linguaggio? La risposta in breve: nessun linguaggio di programmazione è emerso nell'ultimo decennio, ma questo non rispecchia il mondo informatico che al contrario, sia a livello hardware (computer sempre più veloci, multicore) sia a livello di linguaggi diffusi (la tendenza generica per quanto riguarda il sistema dei tipi si è spostata sulla tipizzazione dinamica dei vari Python o Javascript piuttosto che sui vari C++ e Java), ha fatto dei passi da gigante e ha modificato l'ambiente informatico in modo enorme e pervasivo. Inoltre, esiste una mancanza o carenza per quanto riguarda garbage collection e concorrenza.
Ricapitoliamo sinteticamente le cause scatenanti l'inizio del progetto Go:

- I computer sono divenuti tremendamente più veloci ma nonostante ciò lo sviluppo del software non è oggi più rapido di qualche anno fa.
- La gestione delle dipendenze è una parte fondamentale nello sviluppo del software, ma gli "header files" dei diversi linguaggi, nella tradizione del C, ricoprono una posizione antitetica rispetto ad una semplice analisi delle dipendenze e ad una compilazione veloce.
- C'è una crescente rivolta contro gli ormai scomodi sistemi di tipizzazione statica come quelli del Java o del C++, che spinge i programmatori a scegliere sempre più i linguaggi a tipizzazione dinamica come il Python o Javascript.
- Alcuni concetti fondamentali come la "garbage collection" e il calcolo parallelo non sono ben supportati dai più diffusi linguaggi di programmazione.
- La nascita dei processori multicore ha generato preoccupazione e confusione.

1.2 Gli antenati di Go

Go ricorda da vicino C e i suoi vari derivati, che sono ad oggi i più diffusi, con un importante contributo della famiglia Pascal/Modula/Oberon (dichiarazioni, pacchetti), più alcune idee da linguaggi ispirati dal CSP di Tony Hoare, come Newsqueak e Limbo (concorrenza).

1.3 Mascotte di Go

La mascotte di Go è quella che vedete nell'*illustrazione 1.3*, si chiama Gordon ed è un esemplare di gopher.

2 Obiettivi del linguaggio

Il linguaggio di programmazione Go è un progetto open source per rendere più produttivi i programmatori. Go è espressivo, conciso, pulito ed efficiente.
Obiettivo dichiarato, rendere la programmazione veloce, produttiva e divertente. Go, assicura l’azienda di Mountain View, è in grado di offrire un’elevata velocità di compilazione.
Lo scopo del progetto è ambizioso, perché si tenta di creare uno strumento multi-purpose adatto in particolar alla programmazione di sistema, veloce ma al tempo stesso facile da usare e che prenda il meglio dei linguaggi compilati come C++, ovvero velocità di compilazione ed esecuzione, ma anche di quelli dinamici che consentono uno sviluppo veloce e sicuro.
Go è un tentativo di combinare la facilità di programmazione di un linguaggio interpretato tipicamente dinamico con l'efficienza e la sicurezza di un linguaggio compilato tipicamente statico.
I suoi meccanismi di concorrenza rendono facile scrivere programmi che ottenere il massimo dal multicore e macchine in rete, mentre il suo nuovo tipo di sistema permette la costruzione di un programma flessibile e modulare.
Go è un linguaggio object-oriented sì e no. Anche se Go ha tipi e metodi e consente uno stile di programmazione orientato agli oggetti, non esiste una gerarchia dei tipi. Il concetto di "interfaccia" in Go offre un approccio diverso che noi crediamo è facile da usare e in qualche modo più generale. Ci sono anche modi per incorporare tipi in altri tipi per fornire qualcosa di analogo, ma non identico, a sottoclassi. Inoltre, i metodi di Go sono più generali rispetto a C++ o Java: possono essere definiti per ogni tipo di dati, non solo le strutture.
Inoltre, la mancanza di gerarchia dei tipi dà la sensazione che gli "oggetti" in Go siano molto più leggeri rispetto ai linguaggi come C++ o Java.
Infine, è destinato ad essere veloce: esso dovrebbe stare al massimo pochi secondi per costruire un grande file eseguibile su un singolo computer. Per raggiungere questi obiettivi richiesti bisogna affrontare una serie di questioni linguistiche: un sistema di tipo espressivo, ma leggero, la concorrenza e la garbage collection; specifica della dipendenza rigida, e così via. Questi obiettivi non possono essere affrontati bene con le librerie o gli strumenti; un nuovo linguaggio è stato creato per questo.

2.1 Caratteristiche

Go è un linguaggio di programmazione rilasciato da Google come Open Source (licenza BSD).
Le caratteristiche del linguaggio sono:

- velocità – offre un'alta velocità di esecuzione e anche di compilazione in quanto il codice viene trasformato in binario in maniera più o meno istantanea, è possibile realizzare programmi che girano alla stessa velocità di applicazioni native in C, consente una velocità di compilazione eccellente su singola macchina e anche per grossi file binari. Il componente essenziale - che poi rende il sistema così propenso alla velocità - di ogni "Go-programma" sono le "Go-routine": processi leggeri ed efficienti a cui vengono delegati i compiti di gestire i vari sistemi e server da sviluppare;
- semplicità – presenta una sintassi semplice paragonabile a quella di un C misto a Python, fornisce un modello per la costruzione di un software che rende facile l'analisi delle dipendenze ed evita gran parte del sovraccarico dello stile C che include i file e le librerie;
- similarità C – il linguaggio si presenta come un C object-oriented molto ottimizzato. Presenta caratteristiche tipiche dei linguaggi dinamici, ma mantiene, quasi immutate, caratteristiche del C;
- concorrenza – implementata a livello nativo e grazie all’utilizzo delle “goroutine”, semplici funzioni eseguite concorrentemente, possiamo scrivere programmi che sfruttano il parallelismo in tutta tranquillità, fornisce il supporto fondamentale per l'esecuzione e la comunicazione

concorrente ed è stato progettato per ottimizzare i tempi di compilazione anche per hardware modesti;
- multi core – offre il supporto nativo ai processori multi core;
- multi processing – ha un buon supporto per il multi-processing, una gestione leggera e innovativa della programmazione orientata agli oggetti (OOP);
- sicurezza – possiede un sistema di tipizzazione non gerarchico così da non perder tempo nell'analisi e definizione delle gerarchie tra tipi di dato. Anche se possiede dei tipi di dato statici, Go consente una "tipizzazione leggera" che non è facilmente riscontrabile negli altri linguaggi OOP. Il linguaggio, a differenza del C, è controllato strettamente nella tipizzazione e nell'uso della memoria. Ci sono i puntatori, ma non un'aritmetica dei puntatori. Per lavorare con gli array si importa il concetto di slice da Python;
- facilità – la gestione della memoria si basa sulla garbage collection e il feeling generale del linguaggio è da linguaggio interpretato, senza le asprezze tipiche del lavoro col compilatore di un linguaggio strettamente tipizzato.

Con le caratteristiche che ci sono, sembra proprio che i laboratori di Google abbiano preso ispirazione dalle difficoltà dei programmatori C, dalla lentezza dei linguaggi interpretati e dai limiti di performance delle macchine virtuali in circolazione, come Java e .net.
Una semplice traduzione di un programma C++ o Java in Go è improbabile per produrre un risultato soddisfacente – i programmi Java sono scritti in Java, non in Go. D'altra parte, pensare al problema da una prospettiva Go potrebbe produrre un programma efficace, ma molto diverso. In altre parole, per scrivere Go bene, è importante capire le sue proprietà e idiomi. È anche importante conoscere le convenzioni stabilite per la programmazione in Go, come ad esempio la denominazione, la formattazione, la costruzione del programma e così via, in modo che i programmi che scrivete saranno semplici da capire per gli altri programmatori Go.

2.2 Aspettative

Le aspettative per questo linguaggio sembrano essere molte ma quella principale sembra essere quella di scalzare il C++ dal dominio della programmazione concorrente.
Go promette un salto generazionale con l'introduzione delle "goroutine"(piccoli processi leggeri ed estremamente affidabili da aprire a migliaia per la gioia di Google e dei suoi sviluppatori) che dovrebbe permettere di sovrapporre diversi processi e di "dire addio agli stack overflows".
Secondo quanto riportato nelle FAQ sul sito ufficiale la curva di apprendimento non è ripida e se si hanno conoscenze di programmazione di base il livello di difficoltà è lo stesso di Java.
Il rilascio come progetto open source dovrebbe far sì che arrivino tool quali plug-in per l'IDE di Eclipse (al momento non esiste un IDE per Go) e altre migliorie da terze parti. Non è chiaro se Go avrà un ruolo nello sviluppo di Chrome OS, ma Rob Pike, ingegnere capo a Google ha fatto sapere che il sistema è integrabile con Native Client, la tecnologia open-source di Google che consente di eseguire codice nativo dalle applicazioni web.

3 Ambito applicativo

3.1 Installazione

L'installazione richiede alcuni accorgimenti come il settaggio di alcune variabili d'ambiente in bashrc, oltre alla presenza di Mercurial e Bison. Quanto ai comandi per la compilazione, sono anch'essi descritti nella sezione istruzioni.

3.2 Stato sperimentale

Go è un esperimento, ed è importante esserne consapevoli.
Google usa Go internamente ma con cautela per la sua ancora breve vita. Ora ci sono diversi programmi Go distribuiti in produzione all'interno di Google. Per esempio, il server dietro http://golang.org è un programma di Go, infatti è solo il godoc Document Server in esecuzione in una configurazione di produzione.
Al momento disponibile solo su Linux e Mac OS X, Go rimane un progetto che prenderà il 20% del tempo lavorativo, come di consueto per gli esperimenti, in attesa di vedere se si evolverà in qualcosa di importante.
Google si augura che utenti avventurosi lo proveranno e valuteranno se è un buon prodotto per loro e spera che più di qualche programmatore troverà soddisfazione nell'approccio che offre per giustificare un ulteriore sviluppo del linguaggio.

3.3 Utilizzi applicativi e possibili sviluppi

Al momento, date le ridotte dimensioni del progetto, non esiste una versione per Windows, ma gli autori si dicono disponibili a collaborare con chiunque volesse sviluppare una versione per questo sistema operativo. Il programma funziona sotto Linux e, con qualche limitazione legata ai processori ARM, anche con Mac OS X.
Sul versante "mobile", i programmatori hanno verificato piena compatibilità delle applicazioni Go con i dispositivi ARM basati sul sistema operativo Android.
I campi di applicazioni più indicati per il nuovo linguaggio sono quelli dei software di sistema inclusi web server, sistemi di storage e database, anche se Google non chiude le porte a ulteriori ambiti di utilizzo.
Go sembra, ovviamente, fortemente orientato alla creazione di servizi ad alte prestazioni, come quelli tipici di una sala macchine aperta sul web.
Molti dei pacchetti Google Go incentrano le loro attività sul web: accanto ai protocolli più comuni, è stato integrato il supporto per XML, JSON (*J*avaScript Objecti Notation) e per la crittografia dei dati.
Lo sviluppo di questo linguaggio non è da datarsi a fine 2009, ma nel 2007. Dunque possiamo dire che questo progetto è da configurarsi come un possibile tassello nello sviluppo di un'intera piattaforma software che faccia capo ad un Google OS di prossima commercializzazione/distribuzione. In questo senso è da intendersi lo sviluppo costante di Google Android (il sistema operativo per architettura mobile ARM), il browser, le varie applicazioni web e i sistemi di integrazione e socializzazione come Google Wave.
Google Go sembra essere destinato a rivestire, in futuro, per i dispositivi Android, il medesimo ruolo ricoperto dal linguaggio "Objective-C" nel caso degli Apple iPhone.

4 Specifiche del linguaggio

4.1 Introduzione

Questo è un manuale di riferimento per il linguaggio di programmazione Go. Per ulteriori informazioni e altri documenti, vedere http://golang.org .
Go è un linguaggio general-purpose progettato per sistemi di programmazione. È fortemente tipizzato e garbage collection e ha il supporto esplicito per la programmazione concorrente. I programmi utilizzano i packages, le cui proprietà consentono una gestione efficiente delle dipendenze. Le implementazioni esistenti utilizzano un tradizionale modello di compilazione / linking per generare i files binari eseguibili.
La grammatica è compatta e regolare, consentendo una facile analisi con strumenti automatici come ambienti di sviluppo integrati (IDE).

4.2 Notazione

La sintassi è specificata utilizzando la forma estesa di Backus-Naur (EBNF):

```
proposizione = nome_proposizione "=" espressione "." .
espressione = alternativa { "|" alternativa } .
alternativa = termine { termine } .
termine = nome_proposizione | token [ "..."  token ] | gruppo | opzione | ripetizione .
gruppo = "(" espressione ")" .
opzione = "[" espressione "]" .
ripetizione = "{" espressione "}" .
```

Le proposizioni sono espressioni formate dai termini e dai seguenti operatori, in ordine crescente di precedenza:

```
| alternativa
() raggruppamento
[] opzione (0 o 1 volte)
{} ripetizione (0 a n volte)
```

I nomi di proposizione con lettera minuscola sono utilizzati per identificare i token lessicali. I simboli lessicali sono racchiusi tra doppie virgolette `""` o singole virgolette `''` .
La forma `a ... b` `a ... b` rappresenta l'insieme di caratteri da `a` verso `b` come alternative.

4.3 Rappresentazione del codice sorgente

Il codice sorgente è un testo Unicode codificato in UTF-8. Il testo non è canonico, quindi una singola lettera accentata è diversa dallo scrivere un accento seguito da una lettera che sono trattati come due caratteri. Per semplicità, questo documento utilizzerà il termine carattere per riferirsi ad un simbolo Unicode.
Ciascun simbolo è distinto, per esempio, lettere maiuscole e minuscole sono caratteri diversi.
Restrizione dell'implementazione: per la compatibilità con altri strumenti, un compilatore può non consentire il carattere NUL (U+0000) nel testo sorgente.

4.3.1 Caratteri

I seguenti termini sono utilizzati per indicare specifiche classi di caratteri Unicode:

carattere_unicode = / * un arbitrario simbolo Unicode * /.
lettera_unicode = / * un simbolo Unicode classificato come “lettera” */.
cifra_unicode = / * un simbolo Unicode classificato come "cifra decimale" * /.

In “Lo standard Unicode 6.0”, Sezione 4.5 "categoria generale" si definisce un insieme di categorie di caratteri. Go tratta tali caratteri nella categoria Lu, Ll, Lt, Lm o Lo come lettere Unicode, e quelli della categoria Nd come cifre Unicode.

4.3.2 Lettere e cifre

Il carattere di sottolineatura _ (U+005 F) è considerato una lettera.

lettera = lettera_unicode | "_" .
cifre_decimali = "0" ... "9" .
cifre_ottali = "0" ... "7" .
cifre_esadecimali = "0" ... "9" | "A" ... "F" | "a" ... "f" .

4.4 Elementi lessicali

4.4.1 Commenti

Ci sono due tipi di commenti:

1. I commenti di linea iniziano con la sequenza di caratteri // e finiscono alla fine della linea. Una linea di commento è come una nuova linea.
2. I commenti generici si aprono con la sequenza di caratteri /* e continuano fino alla sequenza di caratteri */ . Un commento generico che si estende su più righe è come una nuova linea, altrimenti si comporta come uno spazio.

Non esistono i commenti annidati.

4.4.2 Tokens

I tokens formano il vocabolario del linguaggio Go. Ci sono quattro classi: identificatori, parole chiave, operatori e delimitatori, e valori. Lo spazio bianco, formato da spazi (U+0020), tabulazioni orizzontali (U+0009), ritorni a capo (U+000D), e nuove righe (U+000A), viene ignorato eccetto quando separa tokens che altrimenti si combinerebbero in un unico token. Inoltre, una nuova linea o la fine del file potrebbe causare l'inserimento di un punto e virgola. La sequenza di caratteri in ingresso è suddivisa in tokens, dove ogni token è la più lunga sequenza di caratteri che formano un token valido.

4.4.2.1 Identificatori

Gli identificatori sono le essenze del programma come le variabili e i tipi. Un identificatore è una sequenza di una o più lettere e cifre. Il primo carattere di un identificatore deve essere una lettera.

identificatore = lettera { lettera | cifra_unicode }.

Esempi:
un
_x9
ThisVariableIsExported
αβ

Alcuni identificatori sono prefissati.

4.4.2.2 Parole chiave

Le seguenti parole chiave sono riservate e non possono essere utilizzate come nomi di identificatori.

```
break     default      func     interface  select
case      defer        go       map        struct
chan      else         goto     package    switch
const     fallthrough  if       range      type
continue  for          import   return     var
```

4.4.2.3 Operatori e delimitatori

Le seguenti sequenze di caratteri rappresentano operatori, delimitatori, e altri tokens particolari:

```
+    &     +=    &=     &&    ==    !=    ( )
-    |     -=    |=     ||    <     <=    [ ]
*    ^     *=    ^=     ←     >     >=    { }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     …     .    :
&^   &^=
```

4.4.3 Punti e virgola

La grammatica formale utilizza i punti e virgola ";" come terminatori in una serie di istruzioni. I programmi Go possono omettere la maggior parte di questi punti e virgola utilizzando le seguenti due regole:

1. Quando la sequenza di ingresso è suddivisa in tokens, un punto e virgola viene automaticamente inserito nel flusso di token alla fine di una riga non vuota se il token finale della riga è
   - un identificatore
   - un valore intero, a virgola mobile, immaginario, carattere o stringa
   - una delle parole chiave `break` , `continue` , `fallthrough` , o `return`
   - uno degli operatori e delimitatori ++ , `--` , `)` , `]` , o `}`
2. Per consentire istruzioni complesse di occupare una sola riga, un punto e virgola può essere omesso prima di una chiusura ")" o "}" .

Per rispecchiare l'uso del particolare linguaggio, esempi di codice in questo documento sopprimono i punti e virgola con queste regole.

4.4.4 Valori interi

Un valore intero è una sequenza di cifre che rappresenta una costante intera. Un prefisso facoltativo fissa una base non decimale: `0` per ottale, `0x` o `0X` per esadecimale. Nei valori esadecimali, le lettere `af` e `AF` rappresentano valori da 10 a 15.

valore_intero = valore_decimale | valore_ottale | valore_esadecimale.
valore_decimale = ("1" ... "9") { cifra_decimale } .
valore_ottale = "0" { cifra_ottale } .
valore_esadecimale = "0" ("x" | "X") cifra_esadecimale{ cifra_esadecimale } .

Esempi: 42
0600
0xBadFace
170141183460469231731687303715884105727

4.4.5 Valori in virgola mobile

Un valore in virgola mobile (floating point) è una rappresentazione decimale di una costante in virgola mobile. Ha una parte intera, un punto decimale, una parte frazionaria, e una parte esponente. La parte intera e frazionaria comprendono cifre decimali, la parte esponente è una `e` o `E` seguita facoltativamente da un esponente decimale con segno. Una delle parti intera o frazionaria può essere annullata, uno tra il punto decimale e l'esponente può essere tralasciato.

valore_float = decimali "." [decimali] [esponente] | esponente decimali | "." decimali [esponente].
decimali = cifra_decimale { cifra_decimale }.
esponente = ("e" | "E") ["+" | "-"] decimali .

Esempi: 0.
72.40
072.40 // == 72.40
2.71828
1.e+0
6.67428e-11
1E6
.25
.12345E+5

4.4.6 Valori immaginari

Un valore immaginario è una rappresentazione decimale della parte immaginaria di una costante complessa. Si tratta di un valore floating point o intero decimale seguito dalla lettera minuscola `i` .

valore_immaginaria = (decimali | valore_float) "i" .

Esempi: 0i
011i // == 11i
0.i
2.71828i
1.e+0i
6.67428e-11i
1E6i
.25i
.12345E+5i

4.4.7 Valori carattere

Un valore carattere rappresenta una costante intera, di solito un simbolo Unicode, come uno o più caratteri racchiusi tra singoli apici. Tra virgolette, ogni carattere può apparire tranne tra singolo apice e nuova linea. Un singolo carattere tra apici rappresenta se stesso, mentre le sequenze di più caratteri che iniziano con un backslash codificano i valori in vari formati.
La forma più semplice rappresenta il singolo carattere all'interno delle virgolette, poiché il testo sorgente Go è codificato in caratteri Unicode in UTF-8, più byte codificati UTF-8 possono rappresentare un singolo valore intero. Per esempio, la lettera `'a'` occupa un singolo byte che rappresenta una lettera `a` , Unicode U+0061, valore `0x61` , mentre `'ä'` occupa due byte (`0xc3 0xa4`) che rappresentano una lettera `a` -dieresi, U+00E4, valore `0xe4` .
Diversi backslash consentono valori arbitrari per essere rappresentati come testo ASCII. Ci sono quattro modi per rappresentare il valore intero come una costante numerica: `\x` seguito da esattamente due cifre esadecimali; `\u` seguito da esattamente quattro cifre esadecimali; `\U` seguito da esattamente otto cifre esadecimali, e una semplice backslash `\` seguita da esattamente tre cifre ottali. In ogni caso il valore della lettera è il valore rappresentato dalle cifre nella base corrispondente.
Anche se tutte queste rappresentazioni indicano un numero intero, hanno differenti intervalli validi. Gli escape ottali devono rappresentare un valore tra 0 e 255 incluso. Gli escape esadecimali soddisfano questa condizione da costruzione. Gli escape `\u` e `\U` rappresentano i simboli Unicode così al loro interno alcuni valori non sono validi, in particolare quelli sopra `0x10FFFF` e metà sostituiti.
Dopo un backslash, certi escape di un solo carattere rappresentano valori speciali:

```
\a U+0007     campanello di allarme
\b U+0008     backspace
\f U+000C     salto pagina
\n U+000A     line feed or newline
\r U+000D     ritorno a capo
\t U+0009     tabulazione orizzontale
\v U+000b     tabulazione verticale
\\ U+005c     backslash
\' U+0027     apice singolo (escape valido solo all'interno di valori di caratteri)
\" U+0022     apice doppio (escape valido solo all'interno di valori di stringhe)
```

Tutte le altre sequenze che iniziano con un backslash non sono valide all'interno dei valori di carattere.

```
carattere = "'" ( valore_unicode | valore_byte ) "'" .
valore_unicode = carattere_unicode | valore_piccolo | valore_grande | carattere_escape .
valore_byte = valore_byte_ottale | valore_byte_esadecimale .
valore_byte_ottale = `\` cifra_ottale cifra_ottale cifra_ottale .
valore_byte_esadecimale = `\` "x" cifra_esadecimale cifra_esadecimale.
valore_piccolo = `\` "u" cifra_esadecimale cifra_esadecimale cifra_esadecimale
    cifra_esadecimale .
valore_grande = `\` "U" cifra_esadecimale cifra_esadecimale cifra_esadecimale
    cifra_esadecimale cifra_esadecimale cifra_esadecimale cifra_esadecimale cifra_esadecimale .
carattere_escape = `\` ( "a" | "b" | "f" | "n" | "r" | "t" | "v" | `\` | "'" | `"` ) .
```

Esempi:

```
'a'
'ä'
'本'
'\t'
'\000'
```

'\007'
'\377'
'\x07'
'\xff'
'\u12e4'
'\U00101234'

4.4.8 Valori stringa

Un valore stringa rappresenta una costante stringa ottenuta dalla concatenazione di una sequenza di caratteri. Ci sono due forme: i valori di stringhe raw e i valori di stringhe interpreted.
I valori di stringhe raw sono le sequenze di caratteri tra singole virgolette ' ' . Tra le virgolette, qualsiasi carattere è valido ad eccezione delle singole virgolette. Il valore di una stringa raw è la stringa composta dai caratteri non interpretati tra le virgolette, in particolare, i backslash non hanno alcun significato particolare e la stringa può estendersi su più righe.
I valori di stringhe interpreted sono sequenze di caratteri tra virgolette doppie " ". Il testo tra le virgolette, che non può estendersi su più righe, costituisce il valore della stringa interpreted, con escapes backslash interpretati come lo sono nei valori carattere (tranne che `\'` non è valido e `\"` è valido). Gli escape delle tre cifre ottali (`\` *nnn)* e delle due cifre esadecimali (`\x` *nn)* rappresentano singoli byte della stringa risultante, tutti gli altri escape rappresentano la codifica UTF-8 (possibilmente multi-byte) dei singoli caratteri. Così all'interno di un valore stringa, `\377` e `\xFF` rappresentano un singolo byte di valore `0xFF` = 255, mentre ÿ , `\u00FF` , `\U000000FF` e `\xc3\xbf` rappresentano i due byte `0xc3 0xbf` della codifica UTF-8 di carattere U+00FF.

```
valore_stringa = valore_stringa_raw | valore_stringa_interpreted.
valore_stringa_raw = "`" { carattere_unicode } "`" .
valore_stringa_interpreted = `"` { valore_unicode | valore_byte } `"` .
```

Esempi:

```
`abc`        // come "abc"
`\n
\n`   // come "\\n\n\\n"
"\n"
""
"Hello, world!\n"
"日本語"
"\u65e5 本\U00008a9e"
"\xff\u00FF"
```

Tutti questi esempi rappresentano la stessa stringa:

```
"日本語"   // testo di ingresso in UTF-8
`日本語`   // testo di ingresso in UTF-8 come valore stringa raw
"\u65e5\u672c\u8a9e"      // codice Unicode esplicito
"\U000065e5\U0000672c\U00008a9e"    // codice Unicode esplicito
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e" // bytes UTF-8 espliciti
```

Se il codice sorgente contiene un valore carattere formato da due simboli, come una combinazione che unisce un accento e una lettera, il risultato darà un errore se collocati in un valore carattere (non è un unico simbolo!!) e apparirà come due simboli, senza generare un errore, se collocati in un valore stringa.

4.5 Costanti

Ci sono costanti booleane, intere, floating-point, complesse e stringa. Le costanti intere, a virgola mobile e complesse sono chiamate costanti numeriche.
Un valore costante è rappresentato da un valore intero, a virgola mobile, immaginario, carattere o stringa, o un identificatore che indica una costante, o un'espressione costante, o il valore del risultato di alcune funzioni built-in come `unsafe.Sizeof` è applicata a qualsiasi valore, `cap` o `len` sono applicate ad alcune espressioni, `real` e `imag sono` applicate ad una costante complessa e `complex` è applicata a costanti numeriche. I valori di verità booleani sono rappresentati dalle costanti prefissate `true` e `false`. L'identificatore predichiarato iota denota una costante intera.
In generale, le costanti complesse sono una forma di espressioni costanti e sono discusse in quella sezione.
Le costanti numeriche rappresentano i valori di precisione arbitraria e non vanno in overflow.
Le costanti possono essere tipizzate o non tipizzate. I valori costanti `true` , `false` , `iota` , e alcune espressioni costanti contenenti solo operandi di costanti non tipizzate non sono tipizzate.
A una costante può essere dato esplicitamente un tipo, con una dichiarazione di costante o una conversione, o implicitamente, se utilizzata in una dichiarazione di variabile o di un'assegnazione o come operando in un'espressione. È un errore se il valore della costante non può essere rappresentato come un valore del rispettivo tipo. Per esempio, `3.0` , può essere dato a un qualsiasi numero intero o floating point, mentre `2147483648.0` (pari a `1<<31`) può essere dato ai tipi `float32` , `float64` o `uint32` ma non a `int32` o a `string` .
Non ci sono costanti dell'IEEE-754 con valori di infinito e not-a-number(NaN), ma le funzioni Inf, NaN, IsInf e IsNaN del package math ritornano ed esaminano quei valori in fase di esecuzione.
Restrizione implementazione: Un compilatore può implementare costanti numeriche scegliendo una rappresentazione interna con almeno il doppio del numero di bit di qualsiasi tipo di macchina, per i valori in virgola mobile, sia la mantissa e l'esponente devono essere il doppio più grandi.

4.6 Tipi

Un tipo determina l'insieme di valori e di operazioni specifiche per i valori di quel tipo. Un tipo può essere specificato con un nome di tipo (possibilmente qualificato) (§ identificatore qualificato, § dichiarazioni di tipo) o un valore di tipo, che crea un nuovo tipo da tipi precedentemente dichiarati.

```
Tipo = NomeTipo | ValoreTipo | "(" Tipo ")"
NomeTipo = IdentificatoreQualificato .
ValoreTipo = TipoArray | TipoStruttura | TipoPuntatore | TipoFunzione | TipoInterfaccia |
    TipoSlice | TipoMappa | TipoCanale .
```

Gli esempi dei tipi booleano, numerico e stringa sono già stati affrontati. Tipi composti – tipi array, struttura, puntatore, funzione, interfaccia, slice,mappa e canale - possono essere costruiti utilizzando valori di tipo.
Il tipo statico (o semplicemente tipo) di una variabile è il tipo definito dalla sua dichiarazione. Le variabili del tipo interfaccia hanno anche un diverso tipo dinamico, che è il tipo effettivo del valore memorizzato nella variabile in fase di esecuzione. Il tipo dinamico può variare durante l'esecuzione, ma è sempre assegnabile al tipo statico della variabile interfaccia. Per i tipi non-interfaccia, il tipo dinamico è sempre il tipo statico.
Ogni tipo `T` ha un tipo sottostante: Se `T` è un tipo prefissato o un valore di tipo, il corrispondente tipo sottostante è `T` stesso. In caso contrario, `il` tipo sottostante di T è il tipo sottostante del tipo a cui `T` fa riferimento nella sua dichiarazione del tipo.

```
type T1 string
```

type T2 T1
type T3 []T1
type T4 T3

Il tipo sottostante di `string` , `T1,` e `T2` è `string` . Il tipo sottostante di `[]T1` , `T3` e `T4` è `[]T1` .

4.6.1 Metodo set

Un tipo può avere un metodo di set associato (§ tipi di interfaccia , § dichiarazioni di metodo). Il metodo set di un tipo di interfaccia è la sua interfaccia. Il metodo set di ogni altro tipo denominato `T` è costituito da tutti i metodi con ricevente il tipo `T` . Il metodo set del corrispondente tipo puntatore `*T` è l'insieme di tutti i metodi con ricevente `*T` o `T` (cioè contiene anche il metodo set di `T`). Qualsiasi altro tipo è un metodo set vuoto. In un metodo set, ogni metodo deve avere un nome univoco.

4.6.2 Tipi booleani

Un tipo booleano rappresenta l'insieme di valori di verità booleani denotati dalle costanti prefissate `true` e `false`. Il tipo booleano è indicato con `bool`.

4.6.3 Tipi numerici

Un tipo numerico rappresenta gli insiemi di valori interi o floating-point. I tipi numerici prefissati indipendenti dall'architettura sono:

uint8 l'insieme di tutti i numeri interi senza segno a 8 bit (da 0 a 255)
uint16 l'insieme di tutti i numeri interi senza segno a 16 bit (da 0 a 65535)
uint32 l'insieme di tutti i numeri interi senza segno a 32 bit (da 0 a 4294967295)
uint64 l'insieme di tutti i numeri interi senza segno a 64 bit (da 0 a 18446744073709551615)
int8 l'insieme di tutti i numeri interi con segno a 8 bit (da -128 a 127)
int16 l'insieme di tutti i numeri interi con segno a 16 bit (da -32768 a 32767)
int32 l'insieme di tutti i numeri interi con segno a 32 bit (da -2147483648 a 2147483647)
int64 l'insieme di tutti i numeri interi con segno a 64 bit (da -9223372036854775808 a 9223372036854775807)
float32 l'insieme di tutti i numeri in virgola mobile a 32-bit dell'IEEE-754
float64 l'insieme di tutti i numeri in virgola mobile a 64-bit dell'IEEE-754
complex64 l'insieme di tutti i numeri complessi con parti reale e immaginaria float32
complex128 l'insieme di tutti i numeri complessi con parti reale e immaginaria float64

per i byte si usa di solito uint8

Il valore di un intero a n-bit è grande n bit ed è rappresentato usando l'aritmetica in complemento a due.

Vi è anche un insieme di tipi numerici prefissati con un'implementazione specifica dei formati:

uint è a 32 o a 64 bit
int ha le stesse dimensioni di uint
uintptr un numero intero senza segno abbastanza grande per memorizzare i bit non interpretati di un valore puntatore

Per evitare problemi di portabilità tutti i tipi numerici sono distinti tranne `byte` , che è un alias per `uint8` . Le conversioni sono necessarie quando diversi tipi numerici sono mescolati in un'espressione o in un'assegnazione. Per esempio, `int32` e `int` non sono dello stesso tipo anche

se possono avere la stessa dimensione su una particolare architettura.

4.6.4 Tipi stringa

Un tipo stringa rappresenta l'insieme di valori di stringa. Le stringhe si comportano come array di byte, ma sono immutabili: una volta creati, è impossibile modificare il contenuto di una stringa. Il tipo stringa è indicato con `string` .
Gli elementi di stringhe hanno tipo `byte` e si possono accedere utilizzando le consuete operazioni di indicizzazione. Non è permesso riportare l'indirizzo di tale elemento, se `s[i]` è il i-esimo byte di una stringa, l'istruzione `&s[i]` non è valida. La lunghezza della stringa `s` può essere scoperta usando la funzione built-in `len` . La lunghezza è costante durante il tempo di compilazione se `s` è un valore stringa.

4.6.5 Tipi array

Un array è una sequenza numerata di elementi di un unico tipo, chiamato il tipo dell'elemento. Il numero di elementi è chiamato lunghezza e non è mai negativa.

```
TipoArray = "[" LunghezzaArray "]" TipoElemento .
LunghezzaArray = Espressione .
TipoElemento = Tipo .
```

La lunghezza è parte del tipo di array e deve essere un'espressione costante che restituisce un valore intero non negativo. La lunghezza dell'array `a` può essere scoperta usando la funzione built-in len(a). Gli elementi possono essere indicizzati con indici interi tra 0 e `len(a)-1` (§ Indici). I tipi array sono sempre unidimensionali, ma possono essere composti per formare tipi multi-dimensionali.

```
[32]byte
[2*N] struct { x, y int32 }
[1000]*float64
[3][5]int
[2][2][2]float64 // come [2] ([2] ([2] float64))
```

4.6.6 Tipi slice

Una slice è un riferimento a un segmento contiguo di un'array e contiene una sequenza numerata di elementi di tale array. Un tipo slice indica l'insieme di tutte le slice di array di questo tipo di elemento. Il valore di una slice non inizializzata è `nil` .

```
TipoSlice = "[" "]" TipoElemento .
```

Come gli array, le slice sono indicizzabili e hanno una lunghezza. La lunghezza di una slice `s` può essere scoperta utilizzando la funzione built-in len(s), a differenza degli array può cambiare durante l'esecuzione. Gli elementi possono essere indirizzati con indici interi tra 0 a `len(s)-1` (§ Indici). L'indice di slice di un dato elemento può essere inferiore rispetto all'indice dello stesso elemento dell'array sottostante.
Una slice, una volta inizializzata, è sempre associata ad un'array di base che contiene i suoi elementi. Una slice condivide pertanto la memoria con il suo array e con altre slice dello stesso array, invece, gli array distinti rappresentano sempre aree di memoria distinte.
L'array di base di una slice si può estendere oltre la fine della slice. La capacità è una misura di tale lunghezza: è la somma della lunghezza della slice e della lunghezza dell'array al di là della slice; una slice di lunghezza fino a quella capacità può essere creata tagliandone una nuova dalla slice

originale (§ slice). La capacità di una slice `a` può essere scoperta usando la funzione built-in cap(a). Un nuovo valore slice inizializzato per un dato tipo di elemento `T` è realizzato utilizzando la funzione built-in make, che prende un tipo di slice e i parametri specificando la lunghezza e, facoltativamente, la capacità:

```
make([]T, lunghezza)
make([]T, lunghezza, capacità)
```

La chiamata `make()` alloca un nuovo array nascosto a cui il valore della slice fa riferimento. Cioè, l'esecuzione

```
make([]T, lunghezza, capacità)
```

produce la stessa slice allocando un array e tagliando esso, quindi questi due esempi formano la stessa slice:

```
make([]int, 50, 100)
new([100]int)[0:50]
```

Come gli array, le slice sono sempre unidimensionali ma possono essere composte per costruire oggetti di dimensioni superiori. Con array di array, gli array interni hanno, per costruzione, sempre la stessa lunghezza, ma con le slice di slice (o array di slice), le lunghezze possono variare dinamicamente. Inoltre, le slice interne devono essere allocate individualmente (con `make`).

4.6.7 Tipi struttura

Una struttura è una sequenza di elementi, chiamati campi, ognuno dei quali ha un nome e un tipo. I nomi dei campi possono essere specificati in modo esplicito (ListaIdentificatori) o implicito (CampoAnonimo). All'interno di una struttura, i nomi dei campi non vuoti devono essere univoci.

```
TipoStruttura = "struttura" "{" { DichiarazioneCampo ";" } "}" .
DichiarazioneCampo = ( Tipo ListaIdentificatori | CampoAnonimo ) [ Tag ] .
CampoAnonimo = [ "*" ] NomeTipo .
Tag = variabile_stringa.
// Una struttura vuota.
struct {}
// Una struttura con 6 campi.
struct {
        x, y int
        u float32
        _ float32 // padding
        A *[]int
        F func()
    }
```

Un campo dichiarato con un tipo ma con nessun nome di campo esplicito è un campo anonimo (colloquialmente chiamato un campo integrato). Tale tipo di campo deve essere specificato come un tipo di nome `T` o come un puntatore a un tipo di non-interfaccia di nome `*T` , e `T` stessa non può essere un tipo puntatore. Il nome di un tipo non qualificato si comporta come il nome del campo.

```
// Una struttura con quattro campi anonimi di tipo T1, *T2, P.T3 e *P.T4
struct {
        T1 // nome del campo è T1
        *T2 // nome del campo è T2
        P.T3 // nome del campo è T3
        *P.T4 // nome del campo è T4
        x, y int // i nomi dei campi sono x e y
    }
```

La seguente dichiarazione non è valida perché i nomi dei campi devono essere univoci in un tipo struttura:

```
struct {
		T // conflitti con i campi anonimi *T e *PT
		*T // conflitti con i campi anonimi T e *PT
		*PT // conflitti con i campi anonimi T e *T
	}
```

I campi e i metodi (§ dichiarazioni di metodo) di un campo anonimo sono promossi per essere campi ordinari e metodi della struttura (§ selettori). Le seguenti regole si applicano per un tipo struttura chiamato `S` e un tipo chiamato `T` :

- Se `S` contiene un campo anonimo `T` , il metodo set di `S` include il metodo set di `T` .
- Se `S` contiene un campo anonimo `*T` , il metodo set di `S` include il metodo set di `*T` (che a sua volta include il metodo set di `T`).
- Se `S` contiene un campo anonimo `T` o `*T` , il metodo set di `*S` include il metodo set di `*T` (che a sua volta include il metodo set di `T`).

Una dichiarazione di campo può essere seguita da un tag opzionale del valore stringa, che diventa un attributo per tutti i campi della dichiarazione di campo corrispondente. I tag sono resi visibili attraverso un'interfaccia di riflessione, altrimenti sono ignorati.

```
// Una struttura corrispondente al protocollo di buffer TimeStamp.
// Le stringhe tag definiscono il numero dei campi del protocollo buffer.
struct {
		microsec uint64 "field 1"
		serverIP6 uint64 "field 2"
		process string "field 3"
	}
```

4.6.8 Tipi puntatore

Un tipo puntatore indica l'insieme di tutti i puntatori a variabili di un dato tipo, chiamato il tipo base del puntatore. Il valore di un puntatore non inizializzato è `nil` .

```
TipoPuntatore = "*" TipoBase .
TipoBase = Tipo .
*int
*map[string] *chan int *
```

4.6.9 Tipi funzione

Un tipo funzione indica l'insieme di tutte le funzioni con gli stessi tipi di parametri e risultati. Il valore di una variabile non inizializzata di tipo funzione è `nil` .

```
TipoFunzione = "funzione" Tipo .
Tipo = Parametri [ Risultato ] .
Risultato = Parametri | Tipo .
Parametri = "(" [ ListaParametri [ "," ] ] ")" .
ListaParametri = DichiarazioneParametro { "," DichiarazioneParametro } .
DichiarazioneParametro = [ ListaIdentificatori ] [ "… "] Tipo .
```

All'interno di una lista di parametri o risultati, i nomi (ListaIdentificatori) devono essere tutti presenti o essere tutti assenti. Se presente, ogni nome indica un elemento (parametro o risultato) del tipo specificato, se assente, ogni tipo indica un elemento di quel tipo. Le liste di parametri e di

risultati sono sempre tra parentesi, tranne se c'è esattamente un risultato senza nome che può essere scritto come un tipo non tra parentesi.
Il parametro finale nella firma di una funzione può avere un tipo prefissato. Una funzione con tale parametro è chiamata variadic e può essere invocata con zero o più argomenti per tale parametro.

```
func()
func(x int)
func() int
func(stringa di prefisso, valori ...int)
func(a, b int, z float32) bool
func(a, b int, z float32) (bool)
func(a, b int, z float64, opt ...interface{}) (successo booleano)
func(int, int, float64) (float64, *[]int)
func(n int) func(p *T)
```

4.6.10 Tipi interfaccia

Un tipo interfaccia specifica un metodo set chiamato la sua interfaccia. Una variabile di tipo interfaccia può memorizzare un valore di qualsiasi tipo con un metodo set che è un qualsiasi super insieme di interfaccia. Tale tipo si dice che implementa l'interfaccia. Il valore di una variabile non inizializzata del tipo interfaccia è `nil` .

```
TipoInterfaccia = "interfaccia" "{" { MethodSpec ";" } "}" .
MethodSpec = Firma NomeMetodo | NomeTipoInterfaccia .
NomeMetodo = identificatore .
NomeTipoInterfaccia = NomeTipo .
```

Come con tutti i metodi set, in un tipo interfaccia, ogni metodo deve avere un nome univoco.

```
// Un semplice file di interfaccia
interface {
                Read(b Buffer) bool
                Write(b Buffer) bool
                Close()
           }
```

Più di un tipo può implementare un'interfaccia. Per esempio, se due tipi `S1` e `S2` hanno il metodo set

```
func (p T) Read(b Buffer) bool { return ...  }
func (p T) Write(b Buffer) bool { return ... }
func (p T) Close() { ... }
```

(dove `T` sta per `S1` o `S2`) allora l'interfaccia `File` viene implementata da entrambi `S1` e `S2`, indipendentemente da ciò che altri metodi `S1` e `S2` possono avere o condividere.
Un tipo implementa qualsiasi interfaccia che comprende un sottoinsieme dei suoi metodi e che possono quindi implementare diverse interfacce distinte. Per esempio, tutti i tipi implementano l'interfaccia vuota:

```
interface{}
```

Allo stesso modo, si consideri questa specifica d'interfaccia, che appare all'interno di una dichiarazione di tipo per definire un'interfaccia chiamata `Lock` :

```
type Lock interface {
                        Lock()
                        Unlock()
                     }
```

Se `S1` e `S2` implementano anche

```
func (p T) Lock() { ... }
func (p T) Unlock() { ... }
```
implementano l'interfaccia `Lock` così come l'interfaccia `File`.
Un'interfaccia può contenere un tipo di un'interfaccia di nome `T` al posto di una specifica di metodo. L'effetto è equivalente ad enumerare i metodi di `T` esplicitamente nell'interfaccia.
```
type ReadWrite interface {
                        Read(b Buffer) bool
                        Write(b Buffer) bool
                    }
type File interface {
                    ReadWrite // come l'enumerazione dei metodi in ReadWrite
                    Lock // come l'enumerazione dei metodi in Lock
                    Close()
                }
```

4.6.11 Tipi mappa

Una mappa è un gruppo non ordinato di elementi di un tipo, chiamato il tipo elemento, indicizzato da un insieme di chiavi univoche di un altro tipo, chiamato il tipo chiave. Il valore di una mappa non inizializzata è `nil` .
```
TipoMappa = "mappa" "[" TipoChiave "]" TipoElemento .
TipoChiave = Tipo .
```
Gli operatori di confronto == e != (§ operatori di confronto) devono essere completamente definiti per operandi del tipo chiave, quindi il tipo chiave non deve essere una struttura, un array o una slice. Se il tipo chiave è un tipo di interfaccia, tali operatori di confronto devono essere definiti per i valori della chiave dinamica; in caso di errore si verificherà un panico nel tempo di esecuzione.
```
map [string] int
map [*T] struct { x, y float64 }
map [string] interface {}
```
Il numero degli elementi di una mappa rappresenta la lunghezza della mappa stessa. Per una mappa `m` , la lunghezza può essere scoperta usando la funzione built-in len(m) e può cambiare durante l'esecuzione. I valori possono essere aggiunti e rimossi durante l'esecuzione mediante forme speciali di assegnazione .
Un nuovo e vuoto valore di mappa è realizzato con la funzione di built-in make, che prende il tipo di mappa e opzionalmente una parte di capacità come argomenti:
```
make(map[string] int)
make(map[string] int, 100)
```
La capacità iniziale non vincola le sue dimensioni: le mappe crescono per accogliere il numero di elementi memorizzati all'interno.

4.6.12 Tipi canale

Un canale fornisce un meccanismo per due funzioni, in esecuzione contemporanea, per sincronizzare l'esecuzione e comunicare passando un valore di un tipo di elemento specificato. Il valore di un canale non inizializzato è `nil` .
```
TipoCanale = ( "canale" [ "<-" ] | "<-" "canale" ) TipoElemento .
```
L'operatore `<-` specifica la direzione, l'inviare e il ricevere del canale. Se nessuna direzione è data, il canale è bidirezionale. Un canale può essere limitato solo per inviare o soltanto per ricevere la conversione o l'assegnazione.

```
chan T            // può essere usato per inviare e ricevere valori di tipo T
chan<- float64    // può essere utilizzato solo per inviare float64s
<-chan int        // può essere utilizzato solo per ricevere int
```

L'operatore <− associa il chan più a sinistra possibile:

```
chan<- chan int         // come chan<- (chan int)
chan<- <-chan int       // come chan<- (<-chan int)
<-chan <-chan int       // come <-chan (<-chan int)
chan (<-chan int)
```

Un nuovo valore di canale inizializzato può essere realizzato utilizzando la funzione built-in make, che prende il tipo di canale e opzionalmente una capacità come argomenti:

```
make(chan int, 100)
```

La capacità, in numero di elementi, imposta la dimensione del buffer nel canale. Se la capacità è maggiore di zero, il canale è asincrono: le operazioni di comunicazione hanno esito positivo senza bloccarsi se il buffer non è pieno (invia) o non vuoto (riceve), e gli elementi vengono ricevuti nell'ordine in cui vengono inviati. Se la capacità è pari a zero o assente, la comunicazione ha successo solo quando sia il mittente che il ricevente sono pronti.
Un canale può essere chiuso e testato per la chiusura con le funzioni built-in close e closed.

4.7 Proprietà dei tipi e dei valori

4.7.1 Tipi identità

Due tipi sono tra loro identici o diversi.
Due tipi sono identici se i loro nomi hanno origine nello stesso tipo di dichiarazione. Un tipo con nome e senza nome sono sempre diversi. Due tipi senza nome sono identici se i valori di tipo corrispondente sono identiche, cioè, se hanno lo stesso valore di struttura e i corrispondenti componenti hanno tipi identici. Nel dettaglio:

- Due tipi di array sono identici se hanno tipi di elementi identici e la stessa lunghezza dell'array.
- Due tipi di slice sono identiche se hanno tipi di elementi identici.
- Due tipi di strutture sono identiche se hanno la stessa sequenza di campi, e se i campi corrispondenti hanno gli stessi nomi, tipi identici e le tag identiche. Due campi anonimi sono considerati come aventi lo stesso nome. I nomi dei campi con lettere minuscole di diversi package sono sempre diversi.
- Due tipi di puntatore sono identici se hanno tipi di basi identiche.
- Due tipi di funzioni sono identiche se hanno lo stesso numero di parametri e valori di ritorno, il parametro corrispondente e tipi di valori di ritorno sono identici, ed entrambe le funzioni sono variadic oppure nessuna. I nomi dei parametri e dei valori di ritorno non sono tenuti ad essere identici.
- Due tipi di interfaccia sono identici se hanno lo stesso insieme di metodi con gli stessi nomi e tipi di funzione identici. I nomi di metodo con lettere minuscole di diversi pacchetti sono sempre diversi. L'ordine dei metodi è irrilevante.
- Due tipi di mappe sono identiche se hanno identici tipi di chiave e di valore.
- Due tipi di canale sono identici se hanno tipi di valori identici e la stessa direzione.

Date le dichiarazioni

```
type (
        T0 []string
        T1 []string
        T2 struct { a, b int }
        T3 struct { a, c int }
```

T4 func(int, float64) *T0
T5 func(x int, y float64) *[]string
)

questi tipi sono identici:

T0 e T0
[]int e []int
struct { a, b *T5 } e struct { a, b *T5 }
func(x int, y float64) *[]string e func(int, float64) (result *[]string)

`T0` e `T1` sono diversi perché sono tipi dichiarati con dichiarazioni distinte; `func(int, float64) *T0` e `func(x int, y float64) *[]string` sono diversi perché `T0` è diverso da `[]string`.

4.7.2 Assegnabilità

Un valore `x` è assegnabile ad una variabile di tipo `T` (" `x` è assegnabile a `T` ") in uno qualsiasi di questi casi:

- Il tipo di x è identico a `T`.
- Il tipo di x, che è `V`, e `T` hanno identici tipi sottostanti e almeno uno tra `V` o `T` non è un tipo fissato.
- `T` è un tipo di interfaccia e `x implementa T`.
- x è un valore di canale bidirezionale, T è un tipo di canale, il tipo di x, che è V, e T hanno tipi di elementi identici, e almeno uno tra V o T non è un tipo fissato.
- x ha l'identificatore prefissato nil e T è un tipo puntatore, tipo funzione, tipo slice, tipo mappa, tipo canale o tipo interfaccia.
- x è una costante non tipizzata rappresentabile da un valore di tipo T.

Se T è un tipo struttura con campi non esportati, l'assegnazione deve essere nello stesso package in cui T è dichiarato, o x deve contenere il valore di ritorno di un metodo chiamato. In altre parole, un valore della struttura può essere assegnato a una variabile struttura solo se ogni campo della struttura può essere in modo valido assegnato individualmente dal programma, o se l'assegnazione sta inizializzando il valore di ritorno di un metodo del tipo struttura.
Qualsiasi valore può essere assegnato all'identificatore blank.

4.8 I blocchi

Un blocco è una sequenza di dichiarazioni e istruzioni all'interno di parentesi graffe.

Blocco = "{" { Istruzione ";" } "}" .

In aggiunta ai blocchi espliciti nel codice sorgente, ci sono blocchi impliciti:

1. L'intero blocco circonda tutto il testo sorgente Go.
2. Ogni package ha un blocco package contenente tutto il testo sorgente Go di quel package.
3. Ogni file ha un blocco file contenente tutto il testo sorgente Go di quel file.
4. Ogni istruzione if, for e switch è considerata essere il proprio blocco implicito.
5. Ogni clausola in un'istruzione switch o select agisce come un blocco implicito.

I blocchi annidati influenzano il campo di applicazione.

4.9 Dichiarazioni e campi di applicazione

Una dichiarazione lega un identificatore non blank a una costante, a un tipo, a una funzione o a un package.
Ogni identificatore in un programma deve essere dichiarato. Nessun identificatore può essere dichiarato due volte nello stesso blocco e nessun identificatore può essere dichiarato sia nel file che nel blocco package.

```
Dichiarazione = DichiarazioneCostante | DichiarazioneTipo | DichiarazioneVariabile .
DichiarazioneAltoLivello = Dichiarazione | DichiarazioneFunzione | DichiarazioneMetodo .
```

Il campo di applicazione di un identificatore dichiarato è l'estensione del testo sorgente in cui l'identificatore indica la specifica costante, il tipo, la variabile, la funzione o il package.
Go è lessicalmente visibile usando i blocchi:

1. La visibilità di un identificatore prefissato è l'intero blocco.
2. La visibilità di un identificatore che sta ad indicare una costante, un tipo, una variabile o una funzione (ma non un metodo) dichiarato a livello alto (fuori da qualsiasi funzione) è il blocco package.
3. La visibilità di un identificatore di package importato è il blocco file del file contenente la dichiarazione importata.
4. La visibilità di un identificatore che indica un parametro di funzione o variabile di ritorno è il corpo della funzione.
5. La visibilità di un identificatore dichiarato costante o variabile all'interno di una funzione inizia alla fine di SpecificaCostante o SpecificaVariabile e si conclude alla fine del blocco interno.
6. La visibilità di un tipo di identificatore dichiarato all'interno di una funzione inizia nell'identificatore nel SpecificaTipo e si conclude alla fine del blocco interno.

Un identificatore dichiarato in un blocco può essere dichiarato di nuovo in un blocco interno. Quando l'identificatore della dichiarazione interna è nel campo di applicazione, esso indica l'elemento dichiarato dalla dichiarazione interna.
La clausola package non è una dichiarazione; il nome del package non appare in un qualsiasi campo di applicazione. Il suo scopo è identificare i file appartenenti allo stesso package e specificare il nome del package di default per le dichiarazioni importate.

4.9.1 Etichette di visibilità

Le etichette sono dichiarate da istruzioni etichettate e sono usate nelle istruzioni break, continue e goto (§ istruzioni break, § istruzioni continue, § istruzioni goto). In contrasto con gli altri identificatori, le etichette non sono blocchi visibili e non sono in conflitto con identificatori che non sono etichette. La visibilità di un'etichetta è il corpo della funzione in cui è dichiarata ed esclude il corpo di qualsiasi funzione annidata.

4.9.2 Identificatori prefissati

Gli identificatori seguenti sono implicitamente dichiarati nell'intero blocco:
Tipi base:

```
bool  byte  complex64  complex128  float32  float64
int8  int16  int32  int64  string  uint8  uint16  uint32  uint64
```

Tipi di convenienza con architettura specifica:

```
int  uint  uintptr
```

Costanti:

```
true  false  iota
```

Valore zero:

nil
Funzioni:
append cap close closed complex copy imag len
make new panic print println real recover

4.9.3 Identificatori esportati

Un identificatore può essere esportato per permettere l'accesso a esso da un altro package usando un identificatore qualificato. Un identificatore è esportato se entrambi:

1. il primo carattere del nome dell'identificatore è una lettera Unicode maiuscola (classe Unicode “Lu”); e
2. l'identificatore è dichiarato nel blocco package o indica un campo o un metodo di un tipo dichiarato in quel blocco.

Tutti gli altri identificatori non sono esportati.

4.9.4 Identificatore blank

L'identificatore blank, rappresentato dal carattere sottolineato _, può essere usato in una dichiarazione come qualsiasi altro identificatore ma la dichiarazione non introduce un nuovo obbligo.

4.9.5 Dichiarazioni di costanti

Una dichiarazione di costante lega una lista di identificatori (i nomi delle costanti) ai valori di una lista di espressioni costanti. Il numero degli identificatori deve essere uguale al numero delle espressioni, e l'identificatore n-esimo sulla sinistra è limitato al valore dell'espressione n-esima sulla destra.

```
DichiarazioneCostante = "costante" ( SpecificaCostante | "(" { SpecificaCostante ";" } ")" ) .
SpecificaCostante = ListaIdentificatori [ [ Tipo ] "=" ListaEspressioni ] .
ListaIdentificatori = identificatore { "," identificatore } .
ListaEspressioni = Espressione { "," Espressione } .
```

Se il tipo è presente, tutte le costanti prendono il tipo specificato e le espressioni devono essere assegnabili a quel tipo. Se il tipo è omesso, le costanti prendono i tipi individuali delle espressioni corrispondenti. Se i valori dell'espressione sono costanti senza tipi, le costanti dichiarate rimangono senza tipo e gli identificatori della costante indicano i valori della costante. Per esempio, se l'espressione è un valore floating point, l'identificatore della costante indica una costante floating point, anche se il valore della parte frazionaria è zero.

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0     // costante floating-point senza tipo
const (
        size int64 = 1024
        eof = -1          // costante intera senza tipo
    )
const a, b, c = 3, 4, "foo" // a = 3, b = 4, c = "foo", costanti stringa e intere senza tipo
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

All'interno di una lista di dichiarazioni tra parentesi const la lista delle espressioni può essere omessa da qualsiasi dichiarazione tranne la prima. Una simile lista vuota è equivalente alla sostituzione testuale della prima precedente lista di espressioni non vuota e del suo tipo se non è la prima dichiarazione. Omettendo la lista delle espressioni è poi equivalente al ripetere la lista

precedente. Il numero di identificatori deve essere uguale al numero delle espressioni nella lista precedente. Insieme con il generatore costante iota questo meccanismo permette una dichiarazione leggera di valori sequenziali:

```
const (
        Sunday = iota
        Monday
        Tuesday
        Wednesday
        Thursday
        Friday
        Partyday
        numberOfDays // questa costante non è esportata
)
```

4.9.6 Iota

All'interno di una dichiarazione di costante, l'identificatore prefissato iota rappresenta successive costanti intere non tipizzate. Iota è imposto a zero ogniqualvolta la parola riservata const appare nella sorgente e si incrementa dopo ogni SpecificaCostante. Può essere usato per costruire un insieme di costanti relazionate:

```
const (             // iota è reimpostata a 0
        c0 = iota           // c0 == 0
        c1 = iota           // c1 == 1
        c2 = iota           // c2 == 2
)
const (
        a = 1 << iota   // a == 1 (iota è stata reimpostata a 0)
        b = 1 << iota   // b == 2
        c = 1 << iota   // c == 4
)
const (
        u = iota * 42   // u == 0 (costante intera non tipizzata)
        v float64 = iota * 42   // v == 42.0 (costante float64)
        w = iota * 42   // w == 84 (costante intera non tipizzata)
)
const x = iota      // x == 0 (iota è stata reimpostata a 0)
const y = iota      // y == 0 (iota è stata reimpostata a 0)
```

All'interno di una ListaEspressioni, il valore di ogni iota è lo stesso perché è solo incrementato dopo ogni SpecificaCostante:

```
const (
        bit0, mask0 = 1 << iota, 1 << iota - 1          // bit0 == 1, mask0 == 0
        bit1, mask1         // bit1 == 2, mask1 == 1
        _, _                // salta a iota == 2
        bit3, mask3         // bit3 == 8, mask3 == 7
)
```

L'ultimo esempio utilizza la ripetizione implicita dell'ultima lista di espressioni non vuota.

4.9.7 Dichiarazioni di tipo

Una dichiarazione di tipo lega un identificatore, chiamato il nome del tipo, a un nuovo tipo che ha lo stesso tipo sottostante come un tipo esistente. Il nuovo tipo è diverso dal tipo esistente.

```
DichiarazioneTipo = "tipo" ( SpecificaTipo | "(" { SpecificaTipo ";" } ")" ) .
SpecificaTipo = TipoIdentificatore .
type IntArray [16]int
type (
        Point struct { x, y float64 }
        Polar Point
    )
type TreeNode struct {
                left, right *TreeNode
                value *Comparable
            }
type Cipher interface {
                BlockSize() int
                Encrypt(src, dst []byte)
                Decrypt(src, dst []byte)
            }
```

Il tipo dichiarato non eredita alcun metodo dal tipo esistente, e il metodo set di un tipo interfaccia o degli elementi di un tipo composto rimane invariato:

```
// un Mutex è un tipo di dato con due metodi Lock e Unlock.
type Mutex struct { /* campi Mutex */ }
func (m *Mutex) Lock() { /* implementazione Lock */ }
func (m *Mutex) Unlock() { /* implementazione Unlock */ }

// NewMutex ha la stessa composizione di Mutex ma il suo metodo set è vuoto
type NewMutex Mutex

// il metodo set di un tipo base di PtrMutex rimane invariato
// ma il metodo set di PtrMutex è vuoto
type PtrMutex *Mutex

// il metodo set di *PrintableMutex contiene i metodi
// Lock e Unlock che saltano al suo campo anonimo Mutex.
type PrintableMutex struct {
                Mutex Mutex
            }

// MyCipher è un tipo interfaccia che ha lo stesso metodo set come Cipher.
type MyCipher Cipher
```

Una dichiarazione di tipo può essere usata per definire un differente tipo booleano, numerico o stringa e aggiungere metodi ad essa:

```
type TimeZone int
const (
        EST TimeZone = -(5 + iota)
        CST CST
        MST MST
        PST PST
    )
```

```
func (tz TimeZone) String() string {
                    return fmt.Sprintf("GMT+%dh", tz)
                }
```

4.9.8 Dichiarazioni di variabile

Una dichiarazione di variabile crea una variabile, lega un identificatore ad essa, le associa un tipo e opzionalmente un valore iniziale.

```
DichiarazioneVariabile = "variabile" ( SpecificaVariabile | "(" { SpecificaVariabile ";" } ")" ) .
SpecificaVariabile = ListaIdentificatori ( Tipo [ "=" ListaEspressioni ] | "=" ListaEspressioni ) .
var i int
var U, V, W float64
var k = 0
var x, y float32 = -1, -2
var (
      i int
      u, v, s = 2.0, 3.0, "bar"
     )
var re, im = complexSqrt(-1)
var _, found = entries[name] // mappa di ricerca; solo interessato in "found"
```

Se la lista delle espressioni è data, le variabili sono inizializzate assegnando in ordine le espressioni alle variabili (§ assegnazioni); tutte le espressioni devono essere eliminate e tutte le variabili inizializzate dalle espressioni. Altrimenti, ogni variabile è inizializzata al suo valore zero.
Se il tipo è presente, ogni variabile ha quel tipo. Altrimenti, i tipi sono dedotti dall'assegnazione della lista delle espressioni.
Se il tipo è assente e la corrispondente espressione valuta una costante non tipizzata, il tipo della variabile dichiarata è rispettivamente bool, int, float64 o string, che dipende se il valore è una costante booleana, intera, floating point o stringa:

```
var b = true         // t è di tipo bool
var i = 0     // i è di tipo int
var f = 3.0  // f è di tipo float64
var s = "OMDB"    // s è di tipo string
```

4.9.9 Dichiarazioni di variabili short

Una dichiarazione di variabile short usa la sintassi:

```
DichiarazioneVariabileShort = ListaIdentificatori ":=" ListaEspressioni .
```

È una stenografia per una regolare dichiarazione di variabile con espressioni inizializzate ma nessun tipo:

```
"var" ListaIdentificatori = ListaEspressioni .
i, j := 0, 10
f := func() int { return 7 }
ch := make(chan int)
r, w := os.Pipe(fd)  // os.Pipe() ritorna due valori
_, y, _ := coord(p)  // coord() ritorna tre valori; solo interessati nella coordinata y
```

Diversamente da regolari dichiarazioni di variabile, una dichiarazione di variabile short può ridichiarare variabili già fornite che erano originalmente dichiarate nello stesso blocco con lo stesso tipo, e almeno una delle variabili non vuote è nuova. Come conseguenza, la ridichiarazione può solo apparire in una dichiarazione short multivariabile. La ridichiarazione non introduce una nuova

variabile; essa solo assegna un nuovo valore all'originale.

```
field1, offset := nextField(str, 0)
field2, offset := nextField(str, offset)       // ridichiara l'offset
```

Le dichiarazioni di variabili short possono apparire solo all'interno delle funzioni. In molti contesti simili gli inizializzatori per le istruzioni if, for o switch possono essere usati per dichiarare variabili temporanee locali (§ istruzioni).

4.9.10 Dichiarazioni di funzioni

Una dichiarazione di funzione lega un identificatore a una funzione (§ tipi funzioni).

```
DichiarazioneFunzione = "funzione" identificatore Firma [ Corpo ] .
Corpo = Blocco .
```

Una dichiarazione di funzione può omettere il corpo. Una simile dichiarazione fornisce la firma per una funzione implementata fuori Go, simile come una routine assembly.

```
func min(x int, y int) int {
                        if x < y
                        {       return x }
                        return y
                       }
func flushICache(begin, end uintptr)       // implementata esternamente
```

4.9.11 Dichiarazioni di metodo

Un metodo è una funzione con un destinatario. Una dichiarazione di un metodo lega un identificatore a un metodo.

```
DichiarazioneMetodo = "funzione" Destinatario NomeMetodo Firma [ Corpo ] .
Destinatario = "(" [ identificatore ] [ "*" ] NomeTipoBase ")" .
NomeTipoBase = identificatore .
```

Il tipo destinatario deve essere nella forma T o *T dove T è il nome del tipo. T è chiamato tipo base del destinatario o solo tipo base. Il tipo base non deve essere un tipo puntatore o un tipo interfaccia e deve essere dichiarato nello stesso package come il metodo. Il metodo si dice limitato al tipo base ed è visibile solo all'interno dei selettori di quel tipo (§ tipo dichiarazioni, § selettori).

Dato il tipo Point, le dichiarazioni

```
func (p *Point) Length() float64 {
                              return math.Sqrt(px * px + py * py)
                             }
func (p *Point) Scale(factor float64) {
                                  px *= factor
                                  py *= factor
                                 }
```

legano i metodi Length e Scale, con destinatario tipo *Point, al tipo base Point.

Se il valore del destinatario non fa riferimento all'interno del corpo del metodo, il suo identificatore può essere omesso nella dichiarazione. Lo stesso si applica in generale a parametri di funzioni e metodi.

Il tipo di un metodo è il tipo di una funzione con il destinatario come primo argomento. Per esempio, il metodo Scale ha tipo

```
func(p *Point, factor float64)
```

Comunque, una funzione dichiarata in questo modo non è un metodo.

4.10 Espressioni

Un'espressione specifica il calcolo di un valore applicando operatori e funzioni agli operandi.

4.10.1 Operandi

Gli operandi indicano i valori elementari in un'espressione.

Operando = Valore | IdentificatoreQualificato | EspressioneMetodo | "(" Espressione ")".
Valore = ValoreBase | ValoreComposto | ValoreFunzione .
ValoreBase = intero | float | immaginario | carattere | stringa .

4.10.2 Identificatori qualificati

Un identificatore qualificato è un identificatore non blank qualificato da un nome prefisso di package.

IdentificatoreQualificato = [NomePackage "."] identificatore .

Un identificatore qualificato accede ad un identificatore in un package separato. L'identificatore deve essere esportato da quel package, che significa che deve iniziare con una lettera maiuscola Unicode.

math.Sin

4.10.3 Valori composti

I valori composti producono valori per strutture, array, slice e mappe e creano un nuovo valore ogni volta che sono valutate. Esse sono formate dal tipo del valore seguito da una lista limitata di elementi composti tra parentesi graffe. Un elemento può essere una singola espressione o una coppia chiave-valore.

ValoreComposto = TipoValore Valore .
TipoValore = TipoStruttura | TipoArray | "[" "..." "]" TipoElemento | TipoSlice | TipoMappa | NomeTipo .
Valore = "{" [ListaElementi [","]] "}" .
ListaElementi = Elemento { "," Elemento } .
Elemento = [Chiave ":"] Valore .
Chiave = NomeCampo | IndiceElemento .
NomeCampo = identificatore .
IndiceElemento = Espressione .
Valore = Espressione | Valore .

Il TipoValore deve essere un tipo struttura, array, slice o mappa (la grammatica impone questo vincolo eccetto quando il tipo è dato da NomeTipo). I tipi delle espressioni devono essere assegnabili ai rispettivi tipi campo, elemento e chiave del TipoValore; non c'è una conversione aggiuntiva. La chiave è interpretata come un nome del campo per valori di struttura, un'espressione indice per i valori di array e slice, e una chiave per i valori di mappa. Per i valori di mappa, tutti gli elementi devono avere una chiave. È un errore specificare elementi multipli con lo stesso nome del campo o con il valore della chiave costante.

Per i valori di struttura si applicano le seguenti regole:

- Una chiave deve essere un nome del campo dichiarato in TipoValore.
- Un valore che non contiene alcuna chiave deve elencare un elemento per ogni campo struttura nell'ordine in cui i campi sono dichiarati.
- Se qualche elemento ha una chiave, ogni elemento deve avere una chiave.

- Un valore che contiene chiavi non ha bisogno di avere un elemento per ogni campo struttura. I campi omessi ottengono il valore zero per quel campo.
- Un valore può omettere la lista degli elementi; un simile valore attribuisce il valore zero per quel tipo.
- È un errore specificare un elemento per un campo non esportato appartenente a un package differente.

Date le dichiarazioni:

```
type Point3D struct { x, y, z float64 }
type Line struct { p, q Point3D }
```

si può scrivere

```
origin := Point3D{}          // valore zero per Point3D
line := Line{origin, Point3D{y: -4, z: 12.3}}      // valore zero per la linea .qx
```

Per i valori di array e slice si applicano le seguenti regole:

- Ogni elemento ha un indice intero associato che indica la sua posizione nell'array.
- Un elemento con una chiave usa la chiave come suo indice; la chiave deve essere un'espressione intera costante.
- Un elemento senza una chiave usa l'indice dell'elemento precedente più uno. Se il primo elemento non ha chiave, il suo indice è zero.

Prendendo l'indirizzo di un valore composto (§ Operatori indirizzo) si genera un puntatore a un caso unico del valore.

```
var pointer *Point3D = &Point3D{y: 1000}
```

La lunghezza di un valore di array è la lunghezza specificata nel TipoValore. Se alcuni elementi sono provvisti della lunghezza nella valore, gli elementi persi sono impostati al valore zero per il tipo di elemento dell'array. È un errore fornire elementi con valori indice fuori l'intervallo degli indici dell'array. La notazione specifica una lunghezza dell'array uguale all'indice massimo dell'elemento più uno.

```
buffer := [10]string{}        // len(buffer) == 10
intSet := [6]int{1, 2, 3, 5}            // len(intSet) == 6
days := [...]string{"Sat", "Sun"}   // len(days) == 2
```

Un valore di slice descrive l'intero valore dell'array sottostante. Così, la lunghezza e la capacità di un valore di slice sono il massimo indice dell'elemento più uno. Un valore di slice ha la forma

```
[]T{x1, x2, ... xn}
```

ed è un'abbreviazione per un'operazione slice applicata a un valore di array:

```
[n]T{x1, x2, ... xn}[0 : n]
```

All'interno di un valore composto di tipo array, slice o mappa T, gli elementi che sono a loro volta valori composti possono eliminare il tipo di valore rispettivo se è identico al tipo elemento di T.

```
[...]Point{{1.5, -3.5}, {0, 0}}       // come [...]Point{Point{1.5, -3.5}, Point{0, 0}}
[][]int{{1, 2, 3}, {4, 5}}   // come [][]int{[]int{1, 2, 3}, []int{4, 5}}
```

Un'ambiguità nella grammatica si presenta quando un valore composto usando la forma NomeTipo del TipoValore appare tra la parola chiave e la parentesi graffa aperta del blocco di un'istruzione if, for o switch, perché le parentesi graffe circondanti le espressioni nel valore sono confuse con quelle introdotte nel blocco delle istruzioni. Per risolvere l'ambiguità in questo caso raro, il valore composto deve apparire all'interno delle parentesi.

```
if x == (T{a,b,c}[i]) { ... } }
if (x == T{a,b,c}[i]) { ... } }
```

Esempi di valori di array, slice e mappa valide:

```
// lista di numeri primi
primes := []int{2, 3, 5, 7, 9, 11, 13, 17, 19, 991}
// vowels[ch] è vera se ch è una vocale
vowels := [128]bool{'a': true, 'e': true, 'i': true, 'o': true, 'u': true, 'y': true}
```

```
// l'array [10]float32{-1, 0, 0, 0, -0.1, -0.1, 0, 0, 0, -1}
filter := [10]float32{-1, 4: -0.1, -0.1, 9: -1}
// frequenze in Hz per un egual grado temperato (A4 = 440Hz)
noteFrequency := map[string]float32{
                                "C0": 16.35, "D0": 18.35, "E0": 20.60, "F0": 21.83,
                                "G0": 24.50, "A0": 27.50, "B0": 30.87,
                            }
```

4.10.4 Valori di funzione

Un valore di funzione rappresenta una funzione anonima. Consiste di una descrizione dettagliata del tipo funzione e di un corpo della funzione.

```
ValoreFunzione = TipoFunzione Corpo .
func(a, b int, z float64) bool { return a*b < int(z) }
```

Un valore di funzione può essere assegnata a una variabile o invocata direttamente.

```
f := func(x, y int) int { return x + y }
func(ch chan int) { ch <- ACK } (reply_chan)
```

I valori di funzione hanno una visibilità chiusa: possono riferire a variabili definite in una funzione vicina. Quelle variabili sono poi condivise tra la funzione vicina e il valore di funzione, e sopravvivono finché sono accessibili.

4.10.5 Espressioni principali

Le espressioni principali sono gli operandi delle espressioni unarie e binarie.

```
EspressionePrincipale = Operando | Conversione | ChiamataBuiltIn | EspressionePrincipale
Selettore | EspressionePrincipale Indice | EspressionePrincipale Slice | EspressionePrincipale
TipoAsserzione | EspressionePrincipale Chiamata .
Selettore = "." identificatore .
Indice = "[" Espressione "]" .
Slice = "[" [ Espressione ] ":" [ Espressione ] "]" .
TipoAsserzione = "." "(" Tipo ")" .
Chiamata = "(" [ ListaArgomenti [ "," ] ] ")" .
ListaArgomenti = ListaEspressioni [ "..." ] . ].
```

Esempi:

```
x
2
(s + ".txt")
f(3.1415, true)
Point{1, 2}
m["foo"]
s[i : j + 1]
obj.color
math.Sin
fp[i].x()
```

4.10.6 Selettori

Un'espressione principale della forma

xf

indica il campo o il metodo f del valore indicato da x (o qualche volta *x; vedi sotto).
L'identificatore f è chiamato il selettore (campo o metodo); non deve essere l'identificatore blank. Il tipo dell'espressione è il tipo di f.
Un selettore f può indicare un campo o un metodo f di un tipo T, o può riferirsi a un campo o a un metodo f di un campo anonimo annidato di T. Il numero dei campi anonimi attraversati per raggiungere f è chiamato la sua profondità in T. La profondità di un campo o di un metodo f dichiarato in T è zero. La profondità di un campo o di un metodo f dichiarato in un campo anonimo A in T è la profondità di f più uno.
Le seguenti regole si applicano ai selettori:

1. Per un valore x di tipo T o *T dove T non è un tipo interfaccia, xf indica il campo o il metodo alla profondità più piccola in T dove c'è un simile f. Se non c'è precisamente un f con la profondità più piccola, l'espressione del selettore non è valida.
2. Per una variabile x di tipo I dove I è un tipo interfaccia, xf indica il metodo attuale con nome f di un valore assegnato a x se c'è un metodo simile. Se nessun valore o nil era assegnato a x, xf non è valido.
3. In tutti gli altri casi, xf non è valido.

I selettori automaticamente dereferenziano puntatori a strutture. Se x è un puntatore a una struttura, xy è un'abbreviazione di (*x).y; se il campo y è anche un puntatore a una struttura, xyz è un'abbreviazione di (*(*x).y).z, e così via. Se x contiene un campo anonimo di tipo A, dove A è anche un tipo struttura, xf è un'abbreviazione di (*xA).f.
Per esempio, date le dichiarazioni:

```
type T0 struct {
        x int
    }
func (recv *T0) M0()
type T1 struct {
        y int
    }
func (recv T1) M1()
type T2 struct {
        z int
        T1 T1
        *T0
    }
func (recv *T2) M2()
var p *T2   // con p != nil e p.T1 != nil
```

si può scrivere:

```
pz      // (*p).z
py      // ((*p).T1).y
px      // (*(*p).T0).x
p.M2    // (*p).M2
p.M1    // ((*p).T1).M1
p.M0    // ((*p).T0).M0
```

4.10.7 Indici

Un'espressione principale della forma

```
a[x]
```

indica l'elemento dell'array, della slice, della stringa o della mappa a indicizzata da x. Il valore di x è chiamato l'indice o chiave mappa, rispettivamente. Le seguenti regole si applicano:

Per a di tipo A o *A dove A è un tipo array, o per a di tipo S dove S è un tipo slice:

- x deve essere un valore intero e 0 <= x < len(a)
- a[x] è un elemento dell'array all'indice x e il tipo di a[x] è il tipo dell'elemento di A
- se l'indice x è fuori dell'intervallo, si verifica un panico nel tempo di esecuzione

Per a di tipo T dove T è un tipo stringa:

- x deve essere un valore intero e 0 <= x < len(a)
- a[x] è il byte all'indice x allora il tipo di a[x] è byte
- a[x] può non essere assegnato
- se l'indice x è fuori dell'intervallo, si verifica un panico nel tempo di esecuzione

Per a di tipo M dove M è un tipo mappa:

- il tipo di x deve essere assegnabile al tipo chiave di M
- se la mappa contiene un ingresso con chiave x, a[x] è il valore mappa con chiave x e il tipo di a[x] è il tipo valore di M
- se la mappa non contiene un simile ingresso, a[x] ha valore zero per il tipo valore di M

Altrimenti a[x] non è valida.

Un'espressione indice su una mappa a di tipo map[K]V può essere usata in un'assegnazione o un'inizializzazione della forma speciale.

```
v, ok = a[x]
v, ok := a[x]
var v, ok = a[x]
```

dove il risultato dell'espressione indice è una coppia di valori con tipi (V, bool). In questa forma, il valore di ok è vero se la chiave x è presente nella mappa ed è falso altrimenti. Il valore di v è il valore a[x] come nella forma del valore singolo.

In modo simile, se un'assegnazione a una mappa ha la forma speciale

```
a[x] = v, ok
```

e il booleano ok ha valore false, l'ingresso per la chiave x è cancellata dalla mappa; se ok è true l'istruzione si comporta come un'assegnazione regolare a un elemento della mappa.

4.10.8 Slice

Per una stringa, un'array o una slice a, l'espressione principale

```
a[low : high]
```

costruisce una sottostringa o una slice. Le espressioni dell'indice low e high selezionano quali elementi appaiono nel risultato. Il risultato ha indici che iniziano da 0 alla lunghezza uguale a high - low. Dopo aver tagliato l'array a

```
a := [5]int{1, 2, 3, 4, 5}
s := a[1:4]
```

lo slice `s` ha tipo `[]int` , lunghezza 3, capacità 4, ed elementi

```
s[0] == 2
s[1] == 3
s[2] == 4
```

Per convenienza, qualsiasi delle espressioni indice può essere omessa. Un indice low mancante è di default imposto a zero; un indice high mancante è di default imposto alla lunghezza dell'operando

sliced:

```
a[2:]		// come a[2 : len(a)]
a[:3]		// come a[0 : 3]
a[:]		// come a[0 : len(a)]
```

Per gli array e le stringhe, gli indici low e high devono soddisfare 0 <= low <= high <= lunghezza; per le slice, il limite superiore è la capacità piuttosto che la lunghezza.
Se l'operando sliced è una stringa o una slice, il risultato dell'operazione slice è una stringa o una slice dello stesso tipo. Se l'operando sliced è un array, deve essere indirizzabile e il risultato dell'operazione di slice è una slice con lo stesso tipo di elemento come l'array.

4.10.9 Tipo asserzioni

Per un'espressione x di tipo interfaccia e un tipo T, l'espressione principale

```
x.(T)
```

indica che x non è nil e che il valore immagazzinato in x è di tipo T. La notazione x.(T) è chiamata un tipo asserzione.
Più precisamente, se T non è un tipo interfaccia, x.(T) indica che il tipo dinamico di x è identico al tipo T. Se T è un tipo interfaccia, x.(T) indica che il tipo dinamico di x implementa l'interfaccia T (§ Tipi interfaccia).
Se il tipo asserzione tiene, il valore dell'espressione è il valore immagazzinato in x e il suo tipo è T. Se il tipo asserzione è falso, un panico nel tempo di esecuzione si verifica. In altre parole, anche se il tipo dinamico di x è conosciuto solo al tempo di esecuzione, il tipo di x.(T) è conosciuto essere T in un programma corretto. Se un tipo asserzione è usato in un'assegnazione o in un'inizializzazione della forma

```
v, ok = x.(T)
v, ok := x.(T)
var v, ok = x.(T)
```

il risultato dell'asserzione è una coppia di valori con tipi (T, bool). Se l'asserzione tiene, l'espressione ritorna la coppia (x.(T), true); altrimenti, l'espressione ritorna (Z, false) dove Z è il valore zero per il tipo T. Nessun panico nel tempo di esecuzione accade in questo caso. Il tipo asserzione in questo costrutto così si comporta come una chiamata di funzione che ritorna un valore e un valore booleano indicante il successo (§ Assegnazioni).

4.10.10 Chiamate

Data un'espressione f di tipo funzione F,

```
f(a1, a2, ... an)
```

chiamata f con argomenti a1, a2, … an. Eccetto per un caso speciale, gli argomenti devono essere espressioni valutate singolarmente assegnabili ai tipi parametro di F e sono valutate prima che la funzione sia chiamata. Il tipo dell'espressione è il tipo del risultato di F. Un'invocazione di metodo è simile ma il metodo stesso è specificato come un selettore su un valore del tipo destinatario per il metodo.

```
math.Atan2(x, y)	// chiamata di funzione
var pt *Point
pt.Scale(3.5)		// chiamata del metodo con destinatario pt
```

Come un caso speciale, se i parametri di ritorno di una funzione o di un metodo g sono uguali in numero e individualmente assegnabili ai parametri di un'altra funzione o metodo f, poi la chiamata f(g(parametri_di_g)) invocherà f dopo aver legato i valori di ritorno di g ai parametri di f in ordine. La chiamata di f deve contenere nessun altro parametro che la chiamata di g. Se f ha un parametro

finale, ad esso è assegnato i valori di ritorno di g che rimangono dopo l'assegnazione di parametri regolari.

```
func Split(s string, pos int) (string, string) {
                    return s[0:pos], s[pos:]
                }
func Join(s, t string) string {
            return s + t
        }
if Join(Split(value, len(value)/2)) != value {
                    log.Panic("test fails")
                }
```

Una chiamata di metodo xm() è valida se il metodo set di (il tipo di) x contiene m e la lista degli argomenti può essere assegnata alla lista dei parametri di m. Se x è indirizzabile e il metodo set di &x contiene m, xm() è l'abbreviazione di (&x).m():

```
var p Point
p.Scale(3.5)
```

Non esiste un tipo di metodo distinto e non esistono valori di metodo.

4.10.11 Argomenti passati ai parametri

Se f è variadic con parametro finale tipo T, poi all'interno la funzione l'argomento è equivalente a un parametro di tipo []T. Ad ogni chiamata di f, l'argomento passato al parametro finale è una nuova slice di tipo []T di cui gli elementi successivi sono gli attuali argomenti, dove tutti devono essere assegnabili al tipo T. La lunghezza della slice è poi il numero di argomenti limitati al parametro finale e può differire per ogni posizione chiamata.

Date la funzione e la chiamata

```
func Greeting(prefix string, who ... string)
Greeting("hello:", "Joe", "Anna", "Eileen")
```

all'interno Greeting, avrà il valore []string{"Joe","Anna","Eileen"}. Se l'argomento finale è assegnabile a un tipo slice []T, esso può essere passato invariato come il valore per un parametro T se l'argomento è seguito da … . In questo caso nessuna nuova slice è creata.

Date la slice s e la chiamata

```
s := []string{"James", "Jasmine"}
Greeting("goodbye:", s...)
```

all'interno Greeting, avrà lo stesso valore di s con lo stesso array sottostante.

4.10.12 Operatori

Gli operatori combinano operandi in espressioni.

```
Espressione = EspressioneUnaria | Espressione operando_binario EspressioneUnaria .
EspressioneUnaria = EspressionePrincipale | operando_unario EspressioneUnaria .
operando_binario = "||" | "&&" | operando_relazione | operando_somma |
    operando_moltiplicazione .
operando_relazione = "==" | "!=" | "<" | "<=" | ">" | ">=" .
operando_somma = "+" | "-" | "|" | "^" .
operando_moltiplicazione = "*" | "/" | "%" | "<<" | ">>" | "&" | "&^" .
operando_unario = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

I confronti sono discussi altrove. Per altri operatori binari, i tipi di operando devono essere identici a meno che l'operazione coinvolga canali, spostamenti o costanti non tipizzate. Per operazioni

coinvolgenti solo costanti, vedere la sezione sulle espressioni costanti.
In un canale inviante, il primo operando è sempre un canale e il secondo deve essere un valore assegnabile al tipo di elemento del canale.
Eccetto per operazioni di spostamento, se un operando è una costante non tipizzata e l'altro operando non lo è, la costante è convertita al tipo dell'altro operando.
L'operando destro in un'operazione di spostamento deve essere un tipo intero senza segno o essere una costante non tipizzata che può essere convertita a un tipo intero senza segno.
Se l'operando sinistro in un'operazione di spostamento non costante è una costante non tipizzata, il tipo di costante è quello che sarebbe stato se l'operazione di spostamento era sostituita dal solo operando sinistro.

```
var s uint = 33
var i = 1<<s          // 1 ha tipo int
var j = int32(1<<s) // 1 ha tipo int32; j == 0
var u = uint64(1<<s) // 1 ha tipo uint64; u == 1<<33
var f = float32(1<<s) // non valido: 1 ha tipo float32, non può essere spostato
var g = float32(1<<33) // valido; 1<<33 è un'operazione di spostamento costante; g == 1<<33
```

4.10.12.1 Operatore di precedenza

Gli operatori unari hanno la più alta precedenza. Siccome gli operatori ++ e – – formano istruzioni, non espressioni, essi cadono fuori la gerarchia degli operatori. Come conseguenza, il costrutto p++ è lo stesso come (*p)++.
Ci sono cinque livelli di precedenza per gli operatori binari. Gli operatori per la moltiplicazione sono i più forti, seguiti dagli operatori per l'addizione, dagli operatori di confronto, &&(and logico) e finalmente ||(or logico):
Operatori di precedenza

```
5 * / % << >> & &^
4 + - | ^
3 == != < <= > >=
2 &&
1 ||
```

Gli operatori binari con la stessa precedenza si associano da sinistra a destra. Per esempio, x/y * z è lo stesso come (x/y) * z.

```
+x
23 + 3*x[i]
x <= f()
^a >> b
f() || g()
x == y+1 && <-chan_ptr > 0
```

4.10.12.2 Operatori aritmetici

Gli operatori aritmetici si applicano a valori numerici e producono un risultato dello stesso tipo come il primo operando. I quattro operatori aritmetici standard (+, -, *, /) si applicano a tipi interi, floating point e complessi; + si applica anche a stringhe. Tutti gli altri operatori aritmetici si applicano a solo interi.

```
+ somma valori interi, float e complessi, stringhe
- differenza valori  interi, float e complessi
* prodotto valori interi, float, complessi
/ divisione valori interi, float, complessi
```

% resto intero
& and bit a bit tra interi
| or bit a bit tra interi
^ xor bit a bit tra interi
&^ bit clear (and not) bit a bit tra interi
<< spostamento sinistro intero << per un intero senza segno
>> spostamento destro intero >> per un intero senza segno

Le stringhe possono essere concatenate usando l'operatore + o l'operatore di assegnazione += :

s := "hi" + string(c)
s += " and good bye"

La somma di stringhe crea una nuova stringa concatenando gli operandi.
Per valori interi, / e % soddisfano la seguente relazione:

(a / b) * b + a % b == a
con (a/b) troncato a zero.
xyx / yx % y
5 3 1 2
-5 3 -1 -2
5 -3 -1 2
-5 -3 1 -2

Se il divisore è zero, un panico nel tempo di esecuzione si verifica. Se il dividendo è positivo e il divisore è una potenza costante di 2, la divisione può essere sostituita da uno spostamento a destra, e calcolando il resto può essere sostituito da un'operazione and bit a bit:

xx / 4 x % 4 x >> 2 x & 3
11 2 3 2 3
-11 -2 -3 -3 1

Gli operatori di spostamento spostano l'operando sinistro di un numero di bit pari al conteggio di spostamento specificato dall'operando destro. Essi implementano spostamenti aritmetici se l'operando sinistro è un intero con segno e spostamenti logici se è un intero senza segno. Non c'è un limite superiore sul conteggio di spostamento. Gli spostamenti si comportano come se l'operando sinistro è spostato n volte da 1 per un conteggio di spostamento di n. Di conseguenza, x << 1 è lo stesso come x*2 e x >> 1 è lo stesso come x/2 ma troncato verso l'infinito negativo.
Per operandi interi, gli operatori unari +, – e ^ sono definiti come segue:

+x è uguale a 0 + x
-x (negazione) è uguale a 0 - x
^x (complemento bit a bit) è uguale a m^x con m = "tutti i bit impostati a 1" per x senza segno e m = -1 per x con segno

Per numeri floating point, +x è uguale a x, mentre -x è la negazione di x. Il risultato di una divisione floating point per zero non è specificato al di là dello standard IEEE-754; se un panico nel tempo di esecuzione si verifica è specifico dell'implementazione.

4.10.12.3 Overflow intero

Per valori interi senza segno, le operazioni +, –, * e << sono calcolate modulo 2^n, dove n è il numero dei bit di un tipo di intero senza segno (§ Tipi numerici). Vagamente parlando, queste operazioni di interi senza segno eliminano i bit alti sull'overflow, e i programmi possono fare affidamento sulla copertura.
Per gli interi con segno, le operazioni +, –, * e << possono causare overflow e il valore risultante esiste ed è deterministicamente definito dalla rappresentazione intera senza segno, dall'operazione e dai suoi operandi. Nessuna eccezione è rilevata con un risultato di overflow. Un compilatore non può ottimizzare il codice sotto l'assunzione che l'overflow non si verifica. Per esempio, non si può

assumere che x < x+1 è sempre vero.

4.10.12.4 Operatori di confronto

Gli operatori di confronto confrontano due operandi e restituiscono un valore di tipo bool.

== uguale
!= non uguale
< minore
<= minore o uguale
> maggiore
>= maggiore o uguale

Gli operandi devono essere confrontabili; cioè, il primo operando deve essere assegnabile al tipo del secondo operando, e viceversa.
Gli operandi == e != si applicano a operandi di tutti i tipi eccetto array e strutture. Tutti gli altri operatori di confronto si applicano solo a valori interi, floating point e stringhe. Il risultato di un confronto è definito come segue:

- I valori interi sono confrontati nel modo usuale.
- I valori floating point sono confrontati come definiti dallo standard IEEE-754.
- I valori complessi u e v sono uguali se entrambi real(u) == real(v) e imag(u) == imag(v).
- I valori stringa sono confrontati byte a byte (lessicalmente).
- I valori booleani sono uguali se sono entrambi true o entrambi false.
- I valori puntatore sono uguali se puntano alla stessa locazione di memoria o se entrambi sono nil.
- I valori di funzione sono uguali se riferiscono alla stessa funzione o se entrambi sono nil.
- Un valore slice può essere solo confrontato con nil.
- I valori canale e mappa sono uguali se sono stati creati dalla stessa chiamata a make (§ make per slice, mappe e canali) o se entrambi sono nil.
- I valori interfaccia sono uguali se hanno i tipi dinamici identici e i valori dinamici uguali o se entrambi sono nil.
- Un valore interfaccia x è uguale a un valore non interfaccia y se il tipo dinamico di x è identico al tipo statico di y e il valore dinamico di x è uguale a y.
- Un valore puntatore, funzione, slice, canale, mappa o interfaccia è uguale a nil se è stato assegnato il valore esplicito nil, se non è inizializzato o se è stato assegnato un altro valore uguale a nil.

4.10.12.5 Operatori logici

Gli operatori logici si applicano a valori booleani e restituiscono un valore dello stesso tipo come gli operandi. L'operando destro è valutato sotto condizioni.

and condizionato &&, p && q è "se p è true si ha risultato q altrimenti si ha false"
or condizionato ||, p || q è "se p è true si ha risultato true altrimenti si ha q"
not !, !p è "non p"

4.10.12.6 Operatori indirizzo

Per un operando x di tipo T, l'operazione indirizzo &x genera un puntatore di tipo *T a x.
L'operando deve essere indirizzabile, cioè, è una variabile, o un puntatore non direzionale, o un'operazione indicizzata slice, o un campo selettore di un operando di una struttura indirizzabile, o un'operazione indicizzante un array di un array indirizzabile. Un'eccezione al requisito dell'indirizzabilità è che x può essere anche una variabile composta.

Per un operando x di tipo puntatore *T, il puntatore non direzionale *x indica il valore di tipo T puntato da x.

```
&x
&a[f(2)]
*p
*pf(x)
```

4.10.13 Operatore receive

Per un operando ch di tipo canale, il valore dell'operazione receive <-ch è il valore ricevuto dal canale ch. Il tipo del valore è il tipo dell'elemento del canale. L'espressione si blocca finché un valore è disponibile.

```
v1 := <-ch
v2 = <-ch
f(<-ch)
<-strobe     // aspetta fino all'impulso di clock ed elimina il valore ricevuto
```

Se si riceve nil da un canale si verifica un panico nel tempo di esecuzione.

4.10.14 Espressioni di metodo

Se M è nel metodo set di tipo T, TM è una funzione che è chiamata come una regolare funzione con gli stessi argomenti di M prefissati da un argomento aggiuntivo che è il ritorno del metodo.

```
EspressioneMetodo = TipoRitorno "." NomeMetodo .
TipoRitorno = NomeTipo | "(" "*" NomeTipo ")" .
```

Si consideri una struttura tipo T con due metodi, Mv, il cui ritorno è di tipo T, e Mp, il cui ritorno è di tipo *T.

```
type T struct {
                a int
            }
func (tv T) Mv(a int) int { return 0 }        // valore di ritorno
func (tp *T) Mp(f float32) float32 { return 1 }    // puntatore di ritorno
var t T
```

L'espressione

```
T.Mv
```

restituisce una funzione equivalente a Mv ma con un ritorno esplicito come il suo primo argomento; essa ha forma

```
func(tv T, a int) int
```

Quella funzione può essere chiamata normalmente con un ritorno esplicito, così queste tre invocazioni sono equivalenti:

```
t.Mv(7)
T.Mv(t, 7)
f := T.Mv; f(t, 7)
```

In modo simile, l'espressione

```
(*T).Mp
```

restituisce un valore di funzione rappresentante Mp con firma

```
func(tp *T, f float32) float32
```

Per un metodo con un valore di ritorno, si può derivare una funzione con un esplicito puntatore di ritorno, così

```
(*T).Mv
```

restituisce un valore di funzione rappresentante Mv con firma

    func(tv *T, a int) int

Una funzione simile si comporta indirettamente per creare un valore da passare come ritorno al metodo sottostante; il metodo non sovrascrive il valore il cui indirizzo è passato nella funzione chiamata.
Il caso finale, una funzione con valore di ritorno a un metodo con puntatore al ritorno, non è valido perché i metodi del puntatore al ritorno non sono nel metodo set del tipo valore.
I valori della funzione derivati dai metodi sono chiamati con la sintassi di una funzione chiamata; il valore di ritorno è fornito come il primo argomento alla chiamata. Cioè, data f := T.Mv, f è invocata come f(t, 7) e non tf(7). Per costruire una funzione che lega il destinatario, usare una chiusura.
È valido derivare un valore di funzione da un metodo di un tipo interfaccia. La funzione risultante prende un valore di ritorno esplicito di quel tipo interfaccia.

4.10.15 Le conversioni

Le conversioni sono espressioni della forma T(x) dove T è un tipo e x è un'espressione che può essere convertita al tipo T.

    Conversione = Tipo "(" Espressione ")" .

Se il tipo inizia con un operatore esso deve essere tra parentesi:

```
*Point(p)          // come *(Point(p))
(*Point)(p)        // p è convertito a (*Point)
<-chan int(c)      // come <-(chan int(c))
(<-chan int)(c)    // c è convertito a (<-chan int)
```

Un valore x può essere convertito al tipo T in qualsiasi di questi casi:

- x è assegnabile a T;
- il tipo di x e T hanno tipi sottostanti identici;
- il tipo di x e T sono tipi puntatori senza nome e i loro tipi base puntatore hanno tipi sottostanti identici;
- il tipo di x e T sono entrambi tipi interi o floating point;
- il tipo di x e T sono entrambi tipi complessi;
- x è un intero o ha tipo [] byte o [] int e T è un tipo stringa;
- x è una stringa e T è [] byte o [] int.

Regole specifiche si applicano a conversioni tra tipi numerici e/o a/da un tipo stringa. Queste conversioni possono cambiare la rappresentazione di x e incorrere in un rallentamento del tempo di esecuzione. Tutte le altre conversioni cambiano solo il tipo ma non la rappresentazione di x.

4.10.15.1 Conversioni tra tipi numerici

1. Quando si converte tra tipi numerici, se il valore è un intero con segno il suo segno è esteso a una precisione infinita implicita altrimenti è esteso a zero. È poi troncato per essere adattato nella misura del tipo del risultato. Per esempio, se v := unint16(0x10F0), poi uint32(int8(v)) == 0xFFFFFFF0. La conversione sempre restituisce un valore valido; non esiste indicazione di overflow.
2. Quando si converte un numero floating point in un intero, la frazione è scartata (troncamento verso zero).
3. Quando si converte un numero intero o floating point in un tipo floating point, o un numero complesso in un altro tipo complesso, il valore risultato è arrotondato alla precisione specificata dal tipo destinazione. Per esempio, il valore di una variabile x di tipo float32 può essere memorizzata usando una precisione aggiuntiva oltre quella di un numero a 32 bit di IEEE-754,

ma float32(x) rappresenta il risultato dell'arrotondamento del valore di x alla precisione a 32 bit. In modo simile, x+0.1 può usare più di 32 bit di precisione, ma float32(x+0.1) non può.
In tutte le conversioni coinvolgenti valori floating point o complessi, se il tipo risultato non può rappresentare il valore la conversione ha successo ma il valore risultato è dipendente dall'implementazione.

4.10.15.2 Conversioni a e da un tipo stringa

1. Convertendo un valore intero con o senza segno in un tipo stringa si ottiene una stringa contenente la rappresentazione UTF-8 degli interi. I valori fuori l'intervallo di un simbolo Unicode valido sono convertiti in "\uFFFD".
```
string('a')            // "a"
string(-1)             // "\ufffd" == "\xef\xbf\xbd "
string(0xf8)           // "\u00f8" == "ø" == "\xc3\xb8"
type MyString string
MyString(0x65e5)       // "\u65e5" == "日" == "\xe6\x97\xa5"
```
2. Convertendo un valore di tipo [] byte (o l'equivalente []uint8) in un tipo stringa si ottiene una stringa i cui byte successivi sono gli elementi della slice. Se il valore della slice è nil, il risultato è la stringa vuota.
```
string([]byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'})      // "hellø"
```
3. Convertendo un valore di tipo []int in un tipo stringa si ottiene una stringa che è la concatenazione di interi singoli convertiti a stringhe. Se il valore della slice è nil, il risultato è la stringa vuota.
```
string([]int{0x767d, 0x9d6c, 0x7fd4})      // "\u767d\u9d6c\u7fd4" == "白鵬翔"
```
4. Convertendo un valore di un tipo stringa in []byte (o []uint8) si ottiene una slice i cui elementi successivi sono i byte della stringa. Se la stringa è vuota, il risultato è []byte(nil).
```
[]byte("hellø")        // []byte{'h', 'e', 'l', 'l', '\xc3', '\xb8'}
```
5. Convertendo un valore di un tipo stringa in []int si ottiene una slice contenente il singolo simbolo Unicode della stringa. Se la stringa è vuota, il risultato è []int(nil).
```
[]int(MyString("白鵬翔"))     // []int{0x767d, 0x9d6c, 0x7fd4}
```

Non esiste un meccanismo linguistico per convertire da puntatori e interi. Il package unsafe implementa questa funzionalità sotto circostanze restrittive.

4.10.16 Espressioni costanti

Le espressioni costanti possono contenere solo operandi costanti e sono valutate durante il periodo della compilazione. Costanti non tipizzate booleane, numeriche e stringhe possono essere usate come operandi dovunque è valido usare un operando di tipo booleano, numerico o stringa, rispettivamente. Eccetto per le operazioni di spostamento, se gli operandi di un'operazione binaria sono una costante intera non tipizzata e una costante floating point non tipizzata, la costante intera è convertita in una costante floating point non tipizzata (rilevante per / e per %). In modo simile, le costanti intere non tipizzate o floating point possono essere usate come operandi dovunque è valido usare un operando di tipo complesso; la costante intera o floating point è convertita in una costante complessa con parte immaginaria zero.
Applicando un operatore a costanti non tipizzate si ottiene una costante non tipizzata dello stesso tipo (cioè, una costante booleana, intera, floating point, complessa o stringa), eccetto per operatori di confronto, il cui risultato è in una costante di tipo bool.
Le variabili immaginarie sono costanti complesse non tipizzate (con parte reale zero) e possono essere combinate in operazioni binarie con costanti intere non tipizzate e floating point; il risultato è

una costante complessa non tipizzata. Le costanti complesse sono sempre costruite da espressioni costanti coinvolgenti variabili immaginarie o da costanti derivate da loro, o da chiamate della funzione di built-in complex.

```
const Σ = 1 - 0.707i
const Δ = Σ + 2.0e-4 - 1/1i
const Φ = iota * 1i
const iΓ = complex(0, Γ)
```

Le espressioni costanti sono sempre valutate esattamente; i valori intermedi e le costanti stesse possono richiedere una precisione significativamente più grande piuttosto che supportate da qualsiasi tipo prefissato nel linguaggio. Le seguenti sono dichiarazioni valide:

```
const Huge = 1 << 100
const Four int8 = Huge >> 98
```

I valori di costanti tipizzate devono essere sempre accuratamente rappresentabili come valori del tipo costante. Le seguenti espressioni costanti non sono valide:

```
uint(-1)        // -1 non può essere rappresentato come un uint
int(3.14)       // 3.14 non può essere rappresentato come un int
int64(Huge)     // 1<<100 non può essere rappresentato come un int64
Four * 300      // 300 non può essere rappresentato come un int8
Four * 100      // 400 non può essere rappresentato come un int8
```

La maschera usata dall'operatore unario in complemento bit a bit ^ si combina con la regola per le non costanti: la maschera è tutto 1 per costanti senza segno e -1 per costanti con segno e non tipizzate.

```
^1          // costante intera non tipizzata, uguale a -2
uint8(^1)   // errore, come uint8(-2), fuori dell'intervallo
^uint8(1)   // costante tipizzata uint8, come 0xFF ^ uint8(1) = uint8(0xFE)
int8(^1)    // come int8(-2)
^int8(1)    // come -1 ^ int8(1) = -2
```

4.10.17 Ordine di valutazione

Quando si valutano gli elementi di un'assegnazione o di un'espressione, tutte le chiamate a funzioni, le chiamate a metodi e le operazioni di comunicazione sono valutate nell'ordine lessicale da sinistra a destra.

Per esempio, nell'assegnazione

```
y[f()], ok = g(h(), i() + x[j()], <-c), k()
```

le chiamate di funzioni e la comunicazione si verificano nell'ordine f(), h(), i(), j(), <-c, g() e k(). Comunque, l'ordine di questi eventi confrontati con la valutazione e l'indicizzazione di x e la valutazione di y non è specificato.

Le operazioni floating point all'interno di una singola espressione sono valutate secondo l'associatività degli operatori. Le parentesi esplicite hanno influenza sulla valutazione comportando l'accantonamento dell'associatività di default. Nell'espressione x+(y+z) la somma y+z è eseguita prima della somma a x.

4.11 Istruzioni

Controllo dell'esecuzione delle istruzioni.

Istruzione = Dichiarazione | IstruzioneEtichettata | IstruzioneSemplice | IstruzioneGo | IstruzioneReturn | IstruzioneBreak | IstruzioneContinue | IstruzioneGoto | IstruzioneFallthrough | Blocco | IstruzioneIf | IstruzioneSwitch | IstruzioneSelect | IstruzioneFor | IstruzioneDefer .
IstruzioneSemplice = IstruzioneVuota | IstruzioneEspressione | IstruzioneSend | IstruzioneIncrementoDecremento | Assegnazione | DichiarazioneVariabileShort .

4.11.1 Istruzioni vuote

L'istruzione vuota fa niente.

IstruzioneVuota = .

4.11.2 Istruzioni etichettate

L'utilizzo di un'istruzione etichettata può essere lo scopo di un'istruzione goto, break o continue.

IstruzioneEtichettata = Etichetta ":" Istruzione .
Etichetta = identificatore .

Esempio:

```
Error: log.Panic("error encountered")
```

4.11.3 Istruzioni espressioni

Le chiamate a funzioni, chiamate a metodi ed operazioni ricevere possono apparire nel contesto istruzione.

IstruzioneEspressione = Espressione .

Esempi:

```
h(x+y)
f.Close()
<-ch
```

4.11.4 Istruzioni send

Un'istruzione send invia un valore su un canale. L'espressione del canale deve essere di tipo canale e il tipo del valore deve essere assegnabile al tipo di elemento del canale.

IstruzioneSend = Canale "<-" Espressione .
Canale = Espressione .

Entrambi il canale e il valore dell'espressione sono valutati prima che la comunicazione inizi. La comunicazione si blocca finché l'istruzione send può procedere, a quel punto il valore è trasmesso sul canale. Un'istruzione send su un canale non bufferizzato può procedere se un ricevitore è pronto. Un'istruzione send su un canale bufferizzato può procedere se c'è spazio nel buffer.

```
ch <- 3
```

Inviando un nil nel canale si causa un panico nel tempo di esecuzione.

4.11.5 Istruzioni incremento-decremento

Le istruzioni “++” e “--” incrementano o decrementano i loro operandi della costante non tipizzata 1. Come con un'assegnazione, l'operando deve essere indirizzabile o essere un'espressione di indice mappa.

IstruzioneIncrementoDecremento = Espressione ("++" | "--") .

Le seguenti istruzioni di assegnazione sono semanticamente equivalenti:

```
Assegnazione istruzione incremento-decremento
x++         x += 1
x--  x -= 1
```

4.11.6 Assegnazioni

Assegnazione = ListaEspressioni operazione_assegnazione ListaEspressioni .
operazione_assegnazione = [operazione_somma | operazione_moltiplicazione] "=" .

Ogni operando della parte sinistra deve essere indirizzabile, o un'espressione di indice mappa, o l'identificatore blank.

```
x = 1
*p = f()
a[i] = 23
k = <-ch
```

Un'operazione di assegnazione x op = y dove op è un'operazione aritmetica binaria è equivalente a x = x op y ma valuta x solo una volta. Il costrutto op = è un singolo token. Nelle operazioni di assegnazioni, entrambi le liste di espressioni della parte destra e della parte sinistra devono contenere esattamente una singola espressione valutata.

```
a[i] <<= 2
i &^= 1<<n
```

Un'assegnazione di tupla associa gli elementi individuali di un'operazione più preziosa a una lista di variabili. Ci sono due forme. Nella prima, l'operando della parte destra è un'espressione più preziosa simile a una valutazione di funzione, o un'operazione di canale o di mappa, o un tipo asserzione. Il numero degli operandi sulla parte sinistra deve coincidere con il numero dei valori. Per esempio, se f è una funzione che ritorna due valori,

```
x, y = f()
```

assegna il primo valore a x e il secondo a y. L'identificatore blank fornisce un modo per ignorare i valori ritornati da un'espressione più preziosa:

```
x, _ = f()          // ignora il secondo valore ritornato da f()
```

Nella seconda forma, il numero degli operandi sulla sinistra deve eguagliare il numero di espressioni sulla destra, ognuno di quelli deve essere singolo valutato, e l'ennesima espressione sulla destra è assegnata all'ennesimo operando sulla sinistra. Le espressioni sulla destra sono valutate prima dell'assegnazione di qualsiasi degli operandi sulla sinistra, altrimenti l'ordine di valutazione non è specificato oltre le regole usuali.

```
a, b = b, a         // scambia a e b
```

Nelle assegnazioni, ogni valore deve essere assegnabile al tipo dell'operando a cui è assegnato. Se una costante non tipizzata è assegnata a una variabile di tipo interfaccia, la costante è convertita al tipo bool, int, float64, complex128 o string rispettivamente, a seconda se il valore è una costante booleana, intera, floating point, complessa o stringa.

4.11.7 Istruzioni If

Le istruzioni if specificano l'esecuzione condizionale di due rami secondo il valore di un'espressione booleana. Se l'espressione valuta true, il ramo if è eseguito, altrimenti, se presente, è eseguito il ramo else.

IstruzioneIf = "if" [IstruzioneSemplice ";"] Espressione Blocco ["else" Istruzione] .

```
if x > max {
        x = max
    }
```

L'espressione può essere preceduta da una semplice istruzione, che si esegue prima che l'espressione sia valutata.

```
if x := f(); x < y {
        return x
    } else if x > z {
            return z
        } else {    Else {}
              return y
          }
```

4.11.8 Istruzioni switch

Le istruzioni switch forniscono un'esecuzione a più rami. Un'espressione o specificatore di tipo è confrontato ai casi all'interno lo switch per determinare quale ramo eseguire.

IstruzioneSwitch = IstruzioneEspressioneSwitch | IstruzioneTipoSwitch .

Ci sono due forme: switch di espressione e switch di tipo. In un'espressione switch, i casi contengono espressioni che sono confrontate con il valore dell'espressione switch. In un tipo switch, i casi contengono tipi che sono confrontati con il tipo di un'espressione switch speciale annotata.

4.11.8.1 Espressioni switch

In un'espressione switch, l'espressione switch è valutata e le espressioni dei casi, che non hanno bisogno di essere costanti, sono valutate da sinistra a destra e da sopra a sotto; il primo che eguaglia l'espressione switch scatta l'esecuzione delle istruzioni del caso associato; gli altri casi sono saltati. Se nessun caso coincide e c'è un caso default, le istruzioni del caso di default sono eseguite. Ci può essere al massimo un caso di default e può apparire dovunque nell'istruzione switch. Un'espressione switch mancante è equivalente all'espressione true.

IstruzioneEspressioneSwitch = "switch" [IstruzioneSemplice ";"] [Espressione] "{"
  { ClausolaCasoEspressione } "}" .
ClausolaCasoEspressione = CasoSwitchEspressione ":" { Dichiarazione ";" } .
CasoSwitchEspressione = "case" ListaEspressioni | "default" .

In una clausola di un caso o di default, l'ultima istruzione può solo essere un'istruzione fallthrough (§ Istruzione fallthrough) per indicare che il controllo scorrerebbe dalla fine di questa clausola alla prima istruzione della prossima clausola. Altrimenti il controllo scorre alla fine dell'istruzione switch.

L'espressione può essere preceduta da un'istruzione semplice, che esegue prima che l'espressione sia valutata:

```
switch tag {
        default: s3()
        case 0, 1, 2, 3: s1()
        case 4, 5, 6, 7: s2()
```

```
            }
witch x := f(); {     // espressione switch mancante significa "true"
                      case x < 0: return -x
                      default: return x
                  }
switch {
            case x < y: f1()
            case x < z: f2()
            case x == 4: f3()
        }
```

4.11.8.2 Tipi switch

Un tipo switch confronta tipi piuttosto che valori. È altrimenti simile a un'espressione switch. È contrassegnato da una speciale espressione switch che ha la forma di un tipo asserzione usando la parola riservata type piuttosto che il tipo attuale. I casi poi coincidono con i tipi variabile in previsione del tipo dinamico dell'espressione nel tipo asserzione.

```
IstruzioneTipoSwitch = "switch" [ IstruzioneSemplice ";" ] GuardiaTipoSwitch "{"
   { ClausolaTipoCaso } "}" .
GuardiaTipoSwitch = [ identificatore ":=" ] EspressionePrincipale "." "(" "type" ")" .
ClausolaTipoCaso = CasoTipoSwitch ":" { Istruzione ";" } .
CasoTipoSwitch = "case" ListaTipi | "default" .
ListaTipi = Tipo { "," Tipo } .
```

Il GuardiaTipoSwitch può includere una dichiarazione di variabile short. Quando quella forma è usata, la variabile è dichiarata in ogni clausola. Nelle clausole con un caso elencante esattamente un tipo, la variabile ha quel tipo; altrimenti, la variabile ha il tipo dell'espressione in GuardiaTipoSwitch.

Il tipo in un caso può essere nil (§ Identificatori prefissati); quel caso è usato quando l'espressione nel GuardiaTipoSwitch ha un valore interfaccia nil.

Data un'espressione x di tipo interface{}, si ha il seguente tipo switch:

```
switch i := x.(type) {
                    case nil:
                         printString("x is nil")
                    case int:
                         printInt(i)       // i is an int
                    case float64:
                         printFloat64(i)        // i is a float64
                    case func(int) float64:
                         printFunction(i)       // i is a function
                    case bool, string:
                         printString("type is bool or string")  // i is an interface{}
                    default: default:
                         printString("don't know the type")
                }
```

che può essere riscritto:

```
v := x              // x è valutato esattamente una volta
if v == nil {
            printString("x is nil")
           } else if i, is_int := v.(int); is_int {
            printInt(i)       // i è un int
```

```
            } else if i, is_float64 := v.(float64); is_float64 {
              printFloat64(i)        // i è un float64
            } else if i, is_func := v.(func(int) float64); is_func {
              printFunction(i)       // i è una funzione
            } else {
                    i1, is_bool := v.(bool)
                    i2, is_string := v.(string)
                    if is_bool || is_string {
                            i := v
                            printString("type is bool or string")   // i è un interface{}
                    } else {
                            i := v
                            printString("don't know the type")    // i è un interface{}
                    }
          }
```

La guardia del tipo switch può essere preceduta da una semplice istruzione, che esegue prima che la guardia è valutata.
L'istruzione fallthrough non è permessa nel tipo switch.

4.11.9 Istruzioni for

Un'istruzione for specifica l'esecuzione ripetuta di un blocco. L'iterazione è controllata da una condizione, una clausola for, o una clausola intervallo.

```
IstruzioneFor = "for" [ Condizione | ClausolaFor | ClausolaIntervallo ] Blocco .
Condizione = Espressione .
```

Nella sua più semplice forma, un'istruzione for specifica l'esecuzione ripetuta di un blocco purché una condizione booleana è valutata a true. La condizione è valutata prima di ogni iterazione. Se la condizione è assente, essa è equivalente a true.

```
for a < b {
        a *= 2
      }
```

Un'istruzione for con una ClausolaFor è anche controllata dalla sua condizione, ma in aggiunta può essere specificata un'istruzione iniziale e una post, come un'istruzione di assegnazione, di incremento o di decremento. L'istruzione iniziale può essere una dichiarazione di una variabile short, ma l'istruzione post non la deve essere.

```
ClausolaFor = [ IstruzioneIniziale ] ";" [ Condizione ] ";" [ IstruzionePost ] .
IstruzioneIniziale = IstruzioneSemplice .
IstruzionePost = IstruzioneSemplice .
for i := 0; i < 10; i++ {
                f(i)
              }
```

Se non vuota, l'istruzione iniziale è eseguita una sola volta prima della valutazione della condizione per la prima iterazione; l'istruzione post è eseguita dopo ogni esecuzione del blocco (e solo se il blocco è stato eseguito). Qualsiasi elemento della ClausolaFor può essere vuoto ma i punti e virgola sono richiesti a meno che ci sia solo una condizione. Se la condizione è assente, essa è equivalente a true.

```
for cond { S() }     // è come for ; cond ; { S() }
for { S() }          // è come for true { S() }
```

Un'istruzione for con una clausola intervallo itera attraverso tutti i valori di un array, di una slice, di una stringa o di una mappa, o i valori ricevuti sul canale. Per ogni valore essa assegna valori di

iterazione alle variabili di iterazione corrispondenti e poi esegue il blocco.

ClausolaIntervallo = Espressione ["," Espressione] ("=" | ":=") "range" Espressione .

L'espressione sulla destra nella clausola intervallo è chiamata l'espressione intervallo, che può essere un array, un puntatore a un array, una slice, una stringa, una mappa o un canale. Come con un'assegnazione, gli operandi sulla sinistra devono essere indirizzabili o essere espressioni di indice mappa; essi indicano variabili di iterazione. Se l'espressione intervallo è un canale, solo una variabile di iterazione è permessa, altrimenti ci può essere una o due. L'espressione intervallo è valutata una volta prima dell'inizio del ciclo. Le chiamate a funzioni sulla sinistra sono valutate una volta per iterazione. Per ogni iterazione, i valori dell'iterazione sono prodotti come segue:

Espressione intervallo primo valore e secondo valore (se la seconda variabile è presente)
array o slice a [n]E, *[n]E, o []E indice i int a[i] E
string s tipo stringa indice i int, vedi sotto int
mappa m map[K]V chiave k K m[k] V
canale c chan E element e E

1. Per un valore array o slice, l'iterazione dei valori indice sono prodotti in ordine crescente iniziando all'elemento di indice 0.
2. Per un valore stringa, la clausola intervallo itera sul simbolo Unicode nella stringa iniziando al byte di indice 0. Nelle iterazioni successive, il valore indice sarà l'indice del primo byte dei successivi simboli codificati UTF-8, e il secondo valore, di tipo int, sarà il valore del corrispondente simbolo. Se l'iterazione incontra una sequenza non valida UTF-8, il secondo valore sarà 0xFFFD, la sostituzione del carattere Unicode, e la prossima iterazione avanzerà un singolo byte nella stringa.
3. L'ordine di iterazione su mappe non è specificato. Se gli ingressi della mappa che non sono ancora state raggiunti sono cancellati durante l'iterazione, i valori della corrispondente iterazione non sarà prodotta. Se gli ingressi della mappa sono inseriti durante l'iterazione, il comportamento è dipendente dall'implementazione, ma i valori di iterazione per ogni ingresso sarà prodotto al massimo una volta.
4. Per canali, i valori di iterazione prodotti sono i successivi valori mandati sul canale finché il canale è chiuso; non produce il valore zero mandato prima che il canale sia chiuso (§ close e closed).

I valori di iterazione sono assegnati alle rispettive variabili di iterazione come in un'istruzione di assegnazione. Le variabili di iterazione possono essere dichiarate dalla clausola intervallo (: =). In questo caso i loro tipi sono mandati ai tipi dei rispettivi valori di iterazione e il loro campo di applicazione si conclude alla fine dell'istruzione for; essi sono riusati in ogni iterazione. Se le variabili di iterazione sono dichiarate fuori l'istruzione for, dopo l'esecuzione i loro valori saranno quelli dell'ultima iterazione.

```
var a [10]string
m := map[string]int{"mon":0, "tue":1, "wed":2, "thu":3, "fri":4, "sat":5, "sun":6}
for i, s := range a {
                    // il tipo di i è int
                    // il tipo di s è string
                    // s == a[i]
                    g(i, s)
                    }
var key string
var val interface {}        // tipo valore di m è assegnabile a val
for key, val = range m {
                            h(key, val)
                        }
// key == ultima chiave mappa incontrata nell'iterazione
```

```
// val == map[key]
```

4.11.10 Istruzioni Go

Un'istruzione go inizia l'esecuzione di una chiamata di una funzione o di un metodo come un thread concorrente indipendente di controllo, o goroutine, all'interno dello stesso spazio di indirizzamento.

```
IstruzioneGo = "go" Espressione .
```

L'espressione deve essere una chiamata, diversa da una normale chiamata, dell'esecuzione di un programma che non aspetta che la funzione invocata sia completata.

```
go Server()
go func(ch chan<- bool) { for { sleep(10); ch <- true; }} (c)
```

4.11.11 Istruzioni select

Un'istruzione select sceglie quale di un insieme di possibili comunicazioni procederà. Essa sembra un'istruzione switch ma con i casi tutti referenti ad operazioni di comunicazione.

```
IstruzioneSelect = "select" "{" { ClausolaComunicazione } "}" .
ClausolaComunicazione = CasoComunicazione ":" { Istruzione ";" } .
CasoComunicazione = "case" ( IstruzioneSend | IstruzioneReceive ) | "default" .
IstruzioneReceive = [ Espressione ( "=" | ":=" ) ] EspressioneReceive .
EspressioneReceive = Espressione .
```

EspressioneReceive deve essere un'operazione receive. Per tutti i casi nell'istruzione select, le espressioni del canale sono valutate nell'ordine da sopra a sotto, con qualsiasi espressione che appare sulla parte destra delle istruzioni send. Un canale può essere nil, che è equivalente a quel caso non presente nell'istruzione select eccetto se l'espressione di un'istruzione send è ancora valutata. Se qualsiasi delle operazioni risultanti possono procedere, una di quelle è scelta e la corrispondente comunicazione e istruzioni sono valutate. Altrimenti, se c'è un caso di default, quello è eseguito; se non c'è il caso di default, l'istruzione si blocca finché una delle comunicazioni può completare. Se non ci sono casi con canali non nil, l'istruzione si blocca sempre. Anche se l'istruzione si blocca, il canale e le espressioni send sono valutate solo una volta, sull'istruzione select.
Da allora tutti i canali e le espressioni send sono valutate, qualsiasi cosa che influenza quella valutazione si verificherà per tutte le comunicazioni nell'istruzione select.
Se casi multipli possono procedere, una scelta corretta pseudocasuale è eseguita per decidere quale comunicazione singola sarà presa.
Il caso receive può dichiarare una nuova variabile usando una dichiarazione di variabile short.

```
var c, c1, c2 chan int
var i1, i2 int
select {
        case i1 = <-c1:
                print("received ", i1, " from c1\n")
        case c2 <- i2:
                print("sent ", i2, " to c2\n")
        default: default:
                print("no communication\n")
    }
for {   // sequenza casuale mandata di bit a c
    select {
                case c <- 0:    // nota: nessuna istruzione, fallthrough, casi annidati
```

```
            case c <- 1:
        }
    }
select { }    // blocca sempre
```

4.11.12 Istruzioni return

Un'istruzione return termina l'esecuzione della funzione contenente e opzionalmente fornisce uno o più valori di ritorno al chiamante.

```
IstruzioneReturn = "return" [ ListaEspressioni ] .
```

In una funzione senza un tipo di ritorno, un'istruzione return non deve specificare alcun valore di ritorno.

```
func no_result() {
        return
}
```

Ci sono tre modi per ritornare valori da una funzione con un tipo di ritorno:

1. Il valore o valori di ritorno possono essere esplicitamente elencati nell'istruzione return. Ogni espressione deve essere valutata singolarmente e assegnabile all'elemento corrispondente del tipo di ritorno della funzione.

```
func simple_f() int {
        return 2
}
func complex_f1() (re float64, im float64) {
        return -7.0, -4.0
}
```

2. La lista delle espressioni nell'istruzione return può essere una singola chiamata a una funzione più preziosa. L'effetto è come se ogni valore ritornato da quella funzione fosse assegnato a una variabile temporanea con il tipo del rispettivo valore, seguito da un'istruzione return elencante queste variabili, al cui punto le regole del caso precedente si applicano.

```
func complex_f2() (re float64, im float64) {
        return complex_f1()
}
```

3. La lista delle espressioni può essere vuota se il tipo di ritorno della funzione specifica nomi per i suoi parametri di ritorno (§ Tipi funzione). I parametri di ritorno agiscono come variabili locali ordinarie e la funzione può assegnarli valori come necessario. L'istruzione return ritorna i valori di queste variabili.

```
func complex_f3() (re float64, im float64) {
        re = 7.0
        im = 4.0
        return
}
```

Senza badare a come sono dichiarati, tutti i valori di ritorno sono inizializzati ai valori zero per il loro tipo (§ Il valore zero) all'ingresso della funzione.

4.11.13 Istruzioni break

Un'istruzione break termina l'esecuzione dell'istruzione for, switch o select dall'interno.

IstruzioneBreak = “break” [Etichetta].

Se c'è un'etichetta, essa deve essere fuori di un'istruzione for, switch o select, e la cui esecuzione termina una di quelle istruzioni (§ Istruzioni for, § Istruzioni switch, § Istruzioni select).

```
L: for i < n {
        switch i {
                case 5: break L
        }
}
```

4.11.14 Istruzioni continue

Un'istruzione continue inizia la prossima iterazione del ciclo interno for dopo la sua istruzione (§ Istruzioni for).

IstruzioneContinue = "continue" [Etichetta] .

Se c'è un'etichetta, essa deve essere fuori di un'istruzione for, e la cui esecuzione avanza l'istruzione for. (§ Istruzioni for).

4.11.15 Istruzioni goto

Un'istruzione goto trasferisce il controllo all'istruzione con la corrispondente etichetta.

IstruzioneGoto = "goto" Etichetta .

```
goto Error
```

L'esecuzione dell'istruzione goto non deve provocare che qualsiasi variabile giunga nel campo di applicazione se non era già nel campo di applicazione prima del goto. Questo esempio:

```
goto L     // brutto
v := 3
L:
```

è errato perché il salto all'etichetta L salta la creazione di v.

4.11.16 Istruzioni fallthrough

Un'istruzione fallthrough trasferisce il controllo alla prima istruzione della prossima clausola del ramo in un'istruzione di espressione switch (§ Espressioni switch). Essa può essere usata solo come istruzione finale non vuota in una clausola del ramo o di default in un'istruzione di espressione switch.

IstruzioneFallthrough = "fallthrough" .

4.11.17 Istruzioni defer

Un'istruzione defer invoca una funzione la cui esecuzione è rimandata al momento che la funzione esterna rilasci il controllo.

IstruzioneDefer = "defer" Espressione .

L'espressione deve essere una chiamata di una funzione o di un metodo. Ogni volta che l'istruzione defer si esegue, i parametri alla funzione chiamata sono valutati e salvati di nuovo ma la funzione non è invocata. Le chiamate di funzioni rimandate sono eseguite in ordine LIFO immediatamente prima che la funzione esterna rilasci il controllo, e dopo i valori di ritorno se qualcuno è stato

attribuito, ma prima che ritornino al chiamante. Per esempio, se la funzione rimandata è una variabile funzione e la funzione esterna ha parametri di ritorno con nome che sono nel campo di applicazione all'interno della variabile, la funzione rimandata può accedere e modificare i parametri di ritorno prima che siano ritornati.

```
lock(l)
defer unlock(l)      // sbloccando accade prima che la funzione esterna rilasci il controllo
// stampa 3 2 1 0 prima che la funzione esterna rilasci il controllo
for i := 0; i <= 3; i++ {
        defer fmt.Print(i)
    }
// f ritorna 1
func f() (result int) {
        defer func() {
            result++
        }()
        return 0
    }
```

4.12 Funzioni built-in

Le funzioni built-in sono prefissate. Esse sono chiamate come qualsiasi altra funzione ma molte di loro accettano un tipo invece di un'espressione come primo argomento.
Le funzioni built-in non hanno tipi standard Go, così possono apparire nella chiamata di espressioni; non possono essere usate come valori di funzione.

```
ChiamataBuiltin = identificatore "(" [ ArgomentiBuiltin [ "," ] ] ")" .
ArgomentiBuiltin = Tipo [ "," ListaEspressioni ] | ListaEspressioni .
```

4.12.1 Close e closed

Per un canale c, la funzione built-in close(c) sceglie il canale come non abilitato per accettare più valori attraverso un'operazione send; inviando a o chiudendo un canale closed si provoca un panico nel tempo di esecuzione. Dopo aver chiamato close, e dopo che qualsiasi valore precedentemente inviato è stato ricevuto, le operazioni receive ritorneranno valore zero per il tipo di canale senza il bloccaggio. Dopo che almeno un valore zero è stato ricevuto, closed(c) ritorna true.

4.12.2 Lunghezza e capacità

Le funzioni built-in len e cap prendono argomenti di vari tipi e ritornano un risultato di tipo int. L'implementazione garantisce che il valore di ritorno sempre si adatta in un int.

```
Chiamata Tipo argomento Ritorno
len(s) tipo string lunghezza stringa in byte
   [n]T, *[n]T lunghezza array (== n)
   []T lunghezza slice
   map[K]T lunghezza mappa (numero di chiavi definite)
   chan T numero di elementi in coda nel canale buffer
cap(s) [n]T, *[n]T lunghezza array (== n)
   []T capacità slice
   chan T capacità canale buffer
```

La capacità di una slice è il numero di elementi per cui c'è spazio allocato nell'array sottostante. In

un qualsiasi momento le seguenti relazioni sono valide:

```
0 <= len(s) <= cap(s)
```

La lunghezza e la capacità di una slice, di una una mappa o di un canale nil sono 0.
L'espressione len(s) è una costante se s è una stringa costante. Le espressioni len(s) e cap(s) sono costanti se s è un identificatore (opzionalmente tra parentesi) o un identificatore qualificato che indica un array o un puntatore ad array. Altrimenti le invocazioni di len e cap non sono costanti.

4.12.3 Allocazione

La funzione built-in new prende un tipo T e ritorna un valore di tipo *T. La memoria è inizializzata come descritto nella sezione sui valori iniziali (§ Il valore zero).

```
new(T)
```

Per esempio

```
type S struct { a int; b float64 }
new(S)
```

dinamicamente alloca memoria per una variabile di tipo S, la inizializza (a=0, b=0.0), e ritorna un valore di tipo *S contenente l'indirizzo della memoria.

4.12.4 Creazione di slices, mappe e canali

Le slice, le mappe e i canali sono tipi di deferimento che non richiedono indicazioni extra di un'allocazione con new. La funzione built-in make prende un tipo T, che deve essere un tipo slice, mappa o canale, opzionalmente seguito da un specifico tipo lista di espressioni. Essa ritorna un valore di tipo T (non *T). La memoria è inizializzata come descritto nella sezione sui valori iniziali (§ Il valore zero).

```
Chiamata Tipo T Ritorno
make(T, n) slice        slice di tipo T con lunghezza n e capacità n
make(T, n, m) slice     slice di tipo T con lunghezza n e capacità m
make(T) map             mappa di tipo T
make(T, n) map          mappa di tipo T con spazio iniziale per n elementi
make(T) channel         canale sincrono di tipo T
make(T, n) channel      canale sincrono di tipo T, buffer di misura n
```

Gli argomenti n e m devono essere di tipo intero. Un panico nel tempo di esecuzione si verifica se n è negativo o più grande di m, o se n o m non possono essere rappresentati da un int.

```
s := make([]int, 10, 100)          // slice con len(s) == 10, cap(s) == 100
s := make([]int, 10)               // slice con len(s) == cap(s) == 10
c := make(chan int, 10)            // canale con un buffer di dimensione 10
m := make(map[string] int, 100)    // mappa con spazio iniziale per 100 elementi
```

4.12.5 Aggiunta e copia di slice

Due funzioni di built-in aiutano nelle comuni operazioni slice.
La funzione append aggiunge zero o più valori x a una slice s e ritorna la slice risultante, con lo stesso tipo di s. Ogni valore deve essere assegnabile al tipo di elemento della slice.

```
append(s S, x ...T) S        // S è assegnabile a []T
```

Se la capacità di s non è grande abbastanza per adattare i valori aggiuntivi, append alloca una nuova slice sufficientemente grande che adatti entrambi gli elementi esistenti della slice e i valori aggiuntivi. Così, la slice ritornata può riferire a un differente array sottostante.

```
s0 := []int{0, 0}
```

```
s1 := append(s0, 2)          // aggiunge un elemento singolo s1 == []int{0, 0, 2}
s2 := append(s1, 3, 5, 7)    // aggiunge elementi multipli s2 == []int{0, 0, 2, 3, 5, 7}
s3 := append(s2, s0...)      // aggiunge una slice s3 == []int{0, 0, 2, 3, 5, 7, 0, 0}
```

La funzione copy copia elementi della slice da una sorgente src a una destinazione dst e ritorna il numero di elementi copiati. Sorgente e destinazione possono sovrapporsi. Entrambi gli argomenti devono avere il tipo di elemento identico T e devono essere assegnabili a una slice di tipo []T. Il numero degli argomenti copiati è il minimo tra len(src) e len(dst). Come un caso speciale, copy anche accetta un argomento di destinazione assegnabile al tipo []byte con un argomento sorgente di un tipo stringa. Questa forma copia i byte dalla stringa nei byte della slice.

```
copy(dst, src []T) int
copy(dst []byte, src string) int
```

Esempi:

```
var a = [...]int{0, 1, 2, 3, 4, 5, 6, 7}
var s = make([]int, 6)
var b = make([]byte, 5)
n1 := copy(s, a[0:])             // n1 == 6, s == []int{0, 1, 2, 3, 4, 5}
n2 := copy(s, s[2:])             // n2 == 4, s == []int{2, 3, 4, 5, 4, 5}
n3 := copy(b, "Hello, World!")   // n3 == 5, b == []byte("Hello")
```

4.12.6 Assemblaggio e disassemblaggio di numeri complessi

Tre funzioni assemblano e disassemblano numeri complessi. La funzione built-in complex crea un valore complesso dalla parte reale e immaginaria di un floating point, mentre real e imag estraggono le parti reale e immaginaria di un valore complesso.

```
complex(realPart, imaginaryPart floatT) complexT
real(complexT) floatT
imag(complexT) floatT
```

Nelle tre funzioni sono riportati il tipo degli argomenti e il valore di ritorno corrispondente. Per complex, i due argomenti devono essere dello stesso tipo floating point e ritornano il tipo che è il tipo complesso con i corrispondenti costituenti floating point: complex64 per float32, complex128 per float64.

Le funzioni real e imag insieme formano l'inversa, così per un valore complesso z, z == complex(real(z), imag(z)).

Se gli operandi di queste funzioni sono tutti costanti, il valore di ritorno è una costante.

```
var a = complex(2, -2)               // complex128
var b = complex(1.0, -1.4)           // complex128
x := float32(math.Cos(math.Pi/2))    // float32
var c64 = complex(5, -x)             // complex64
var im = imag(b)                     // float64
var rl = real(c64)                   // float32
```

4.12.7 Trattamento panici

Le due funzioni built-in, panic e recover, assistono nel reporting, nel trattamento dei panici nel tempo di esecuzione e nelle condizioni dell'errore del programma definito.

```
func panic(interface{})
func recover() interface{}
```

Quando una funzione F chiama panic, l'esecuzione normale di F si ferma immediatamente. Qualsiasi funzione, la cui esecuzione era rimandata dall'invocazione di F, è eseguita nel modo

usuale e poi F rilascia il controllo al suo chiamante. Al chiamante, F poi si comporta come una chiamata a panic, terminando la propria esecuzione ed eseguendo le funzioni rimandate. Questa funzione continua finché tutte le funzioni nella goroutine hanno cessato l'esecuzione, in ordine inverso. A quel punto, il programma è terminato e la condizione d'errore è riportata, includendo il valore dell'argomento a panic. Questa sequenza di terminazione è chiamata panicking.

```
panic(42)
panic("unreachable")
panic(Error("cannot parse"))
```

La funzione recover permette a un programma di gestire il comportamento di una goroutine panicking. Eseguendo una chiamata recover all'interno di una funzione rimandata (ma non una qualsiasi funzione chiamata da essa) si ferma la sequenza panicking ristabilendo l'esecuzione normale, e ripristina valore d'errore passato alla chiamata di panic. Se recover è chiamata fuori la funzione rimandata essa non fermerà una sequenza panicking. In questo caso, o quando la goroutine non è panicking, o se l'argomento fornito a panic era nil, recover ritorna nil.
La funzione protect nell'esempio sotto invoca la funzione con argomento g e protegge i chiamanti da panici nel tempo di esecuzione aumentati da g.

```
func protect(g func())
{    defer func()
     {        log.Println("done")    // Println esegue normalmente anche quando c'è un panico
              if x := recover(); x != nil
              {        log.Printf("run time panic: %v", x)
              }
     }
     log.Println("start")
     g()
}
```

4.12.8 Bootstrapping

Le correnti implementazioni forniscono funzioni built-in utili durante il bootstrapping. Queste funzioni sono documentate per completezza ma non sono garantite per rimanere nel linguaggio. Esse non ritornano un risultato.

Funzione Comportamento
print: stampa tutti gli argomenti; formattazione di argomenti è specifica dell'implementazione
println: come print ma stampa spazi tra gli argomenti e una nuova linea alla fine

4.13 I Package

I programmi Go sono costruiti collegando insieme i packages. Un package in uso è costituito da uno o più file sorgenti che insieme dichiarano costanti, tipi, variabili e funzioni appartenenti al package e che sono accessibili in tutti i file dello stesso package. Quegli elementi possono essere esportati e usati in un altro package.

4.13.1 Organizzazione del file sorgente

Ogni file sorgente consiste di una clausola package che definisce il package a cui esso appartiene, seguito da un possibile insieme vuoto di importazione di dichiarazioni che dichiarano package i cui contenuti il file vuole usare, seguito da un possibile insieme vuoto di dichiarazioni di funzioni, tipi, variabili e costanti.

FileSorgente = ClausolaPackage ";" { ImportazioneDichiarazioni ";"
{ DichiarazioneAltoLivello ";" } .

4.13.2 Clausola package

Una clausola package si trova all'inizio di ogni file sorgente e definisce il package a cui il file appartiene.

ClausolaPackage = "package" NomePackage .
NomePackage = identificatore .

Il NomePackage non deve essere l'identificatore blank.

package math

Un insieme di file che condividono lo stesso NomePackage formano l'implementazione di un package. Un'implementazione può richiedere che tutti i file sorgenti di un package si trovino nella stessa directory.

4.13.3 Importazione dichiarazioni

Un'importazione di dichiarazione stabilisce che il file sorgente contenente la dichiarazione usa identificatori esportati dal package importato e abilita l'accesso ad essi. L'importazione chiama un identificatore (NomePackage) per essere usato per l'accesso e un ImportazionePercorso che specifica il package da essere importato.

ImportazioneDichiarazione = "import" (ImportazioneSpecifica | "("
{ ImportazioneSpecifica ";" } ")") .
ImportazioneSpecifica = ["." | NomePackage] ImportazionePercorso .
ImportazionePercorso = variabile_stringa .

Il NomePackage è usato negli identificatori qualificati per accedere agli identificatori esportati del package all'interno del file sorgente importato. Esso è dichiarato nel blocco file. Se il NomePackage è omesso, esso è di default l'identificatore specificato nella clausola package del package importato. Se un punto esplicito (.) appare invece di un nome, tutti gli identificatori del package esportato saranno dichiarati nel blocco file del corrente file e può essere acceduto senza un qualificatore. L'interpretazione dell'ImportazionePercorso è dipendente dall'implementazione ma è tipicamente una sottostringa di un nome file completo del package compilato e può essere relativo a un deposito di package installati. Assumiamo che abbiamo compilato un package contenente la clausola package package math, che esporta la funzione sin, e installato il package compilato nel file identificato da “lib/math”. Questa tabella illustra come sin può essere acceduta nei file che importano il package dopo i vari tipi di importazione di dichiarazione.

Importazione Dichiarazione Nome locale di Sin
import "lib/math" math.Sin
import M "lib/math" M.Sin
import . "lib/math" Sin

Un'importazione di dichiarazione dichiara una relazione di dipendenza tra package da importare e importati. Non è valido per un package importare se stesso o importare un package senza l'assegnazione di qualsiasi dei suoi identificatori esportati. Per importare un package solamente per i suoi effetti di lato (inizializzazione), usare l'identificatore blank come nome esplicito di package:

import _ "lib/math"

4.13.4 Un esempio di package

Qui di seguito è riportato un package Go completo che implementa un crivello concorrente dei numeri primi (crivello di Eratostene).

```
package main
import "fmt"
// Manda la sequenza 2, 3, 4, ... al canale 'ch'.
func generate(ch chan<- int)
{	for i := 2; ; i++
	{	ch <- i		// Manda 'i' al canale 'ch'.
	}
}
// Copia i valori dal canale 'src' al canale 'dst',
// rimuovendo quei valori divisibili per i 'primi'.
func filter(src <-chan int, dst chan<- int, prime int)
{	for i := range src		// Ciclo sui valori ricevuti da 'src'.
	{	if i%prime != 0
		{	dst <- i	// Manda 'i' al canale 'dst'.
		}
	}
}
// Il crivello dei primi: filtro daisy-chain elabora insieme.
func sieve()
{	ch := make(chan int)		// Crea un nuovo canale.
	go generate(ch)			// Inizia generate() come un sottoprocesso.
	for
	{	prime := <-ch
		fmt.Print(prime, "\n")
		ch1 := make(chan int)
		go filter(ch, ch1, prime)
		ch = ch1
	}
}
func main()
{	sieve()
}
```

4.14 Inizializzazione ed esecuzione del programma

4.14.1 Il valore zero

Quando la memoria è allocata per immagazzinare un valore, si può effettuare una dichiarazione di chiamata di make() oppure di new(), e se nessuna inizializzazione esplicita è fornita, alla memoria è data un'inizializzazione di default. Ogni elemento inizializzato di default è imposto al valore zero in base al suo tipo: false per i boolean, 0 per gli interi, 0.0 per i float, “ “ per le stringhe, e nil per i puntatori, le funzioni, le interfacce, le slice, i canali e le mappe. Questa inizializzazione è eseguita ricorsivamente, così per esempio ogni elemento di un array di strutture avrà i suoi campi zero se nessun valore è specificato.

Queste due semplici dichiarazioni sono equivalenti:

```
var i int
```

```
var i int = 0
```

Dopo

```
type T struct { i int; f float64; next *T }
t := new(T)
```

le seguenti sono valide:

```
ti == 0
tf == 0.0
t.next == nil
```

Le stesse sarebbero true anche dopo

```
var t T
```

4.14.2 Esecuzione del programma

Un package con nessuna importazione è inizializzato assegnando valori iniziali a tutte le sue variabili a livello package e poi chiamando qualsiasi funzione di livello package con il nome e la forma di

```
func init()
```

definita nella sua sorgente. Un package può contenere funzioni multiple init(), anche all'interno di un singolo file sorgente; esse si eseguono in un ordine non specificato.
All'interno di un package, le variabili di livello package sono inizializzate, e valori costanti sono determinati, in ordine dipendente dai dati: se l'inizializzatore di A dipende dal valore di B, A sarà impostato dopo B. È un errore se simili dipendenze formano un ciclo. L'analisi delle dipendenze è effettuata lessicalmente: A dipende da B se il valore di A contiene una citazione a B, o contiene un valore il cui inizializzatore nomina B, o cita una funzione che nomina B, ricorsivamente. Se due elementi non sono interdipendenti, essi saranno inizializzati nell'ordine che appaiono nella sorgente. Da allora l'analisi delle dipendenze è effettuata per package, essa può produrre risultati non specificati se l'inizializzatore di A chiama una funzione definita in un altro package che riferisce a B.
Il codice di inizializzazione può contenere istruzioni “go”, ma le funzioni che loro invocano non iniziano l'esecuzione finché l'inizializzazione dell'intero programma è completa. Quindi, tutto il codice di inizializzazione è in esecuzione in una singola goroutine.
Una funzione init() non può essere posizionata dovunque in un programma. In particolare, init() non può essere chiamata esplicitamente, neppure può un puntatore a init essere assegnato a una variabile funzione. Se un package ha importazioni, i package importati sono inizializzati prima di inizializzare il package stesso. Se package multipli importano un package P, P sarà inizializzato una sola volta.
L'importazione di package, da costruzione, garantisce che non ci può essere alcuna dipendenza ciclica nell'inizializzazione.
Un programma completo è creato dal linking di un package, singolo e non importato, chiamato il main package con tutti i package che esso importa, transitivamente. Il main package deve avere nome package main e dichiarare una funzione main che non prende argomenti e ritorna nessun valore.

```
func main() { ... } }
```

L'esecuzione del programma comincia inizializzando il main package e poi invocando la funzione main. Quando la funzione main rilascia il controllo, il programma esce. Il programma non aspetta altre goroutine (non-main) per completare.

4.15 Panici nel tempo di esecuzione

Gli errori di esecuzione simili come un tentativo di indicizzare un array fuori dei limiti innescano un panico nel tempo di esecuzione equivalente a una chiamata di una funzione built-in panic con un valore dell'implementazione definita tipo interfaccia runtime.Error. Quel tipo definisce almeno il metodo String() string. I valori di errore esatti che rappresentano distinte condizioni di panico nel tempo di esecuzione non sono specificate, almeno per ora.

```
package runtime
type Error interface
{    String() string
        // e forse altri
}
```

4.16 Considerazioni di sistema

4.16.1 Package unsafe

Il package built-in unsafe, conosciuto al compilatore, fornisce facilitazioni per la programmazione a basso livello includendo operazioni che violano il tipo di sistema. Un package usando unsafe deve prestare attenzione per il suo tipo di sicurezza. Il package fornisce la seguente interfaccia:

```
package unsafe
type ArbitraryType int       // abbreviazione per un tipo arbitrario Go; esso non è un tipo reale
type Pointer *ArbitraryType
func Alignof(variable ArbitraryType) int
func Offsetof(selector ArbitraryType) int
func Sizeof(variable ArbitraryType) int
func Reflect(val interface{}) (typ runtime.Type, addr uintptr)
func Typeof(val interface{}) (typ interface{})
func Unreflect(typ runtime.Type, addr uintptr) interface{}
```

Qualsiasi puntatore o valore di tipo uintptr può essere convertito in un Pointer e viceversa. La funzione Sizeof prende un'espressione indicante una variabile di qualsiasi tipo e ritorna la dimensione della variabile in byte.

La funzione Offsetof prende un selettore (§ Selettori) indicante un campo struttura di qualsiasi tipo e ritorna il campo offset in byte relativo all'indirizzo della struttura. Per una struttura s con campo f:

```
uintptr(unsafe.Pointer(&s)) + uintptr(unsafe.Offsetof(sf)) == uintptr(unsafe.Pointer(&s.f))
```

Le architetture del computer possono richiedere agli indirizzi di memoria di essere allineati; cioè, gli indirizzi di una variabile devono essere un multiplo di un fattore, l'allineamento del tipo di variabile. La funzione Alignof prende un'espressione indicante una variabile di qualsiasi tipo e ritorna l'allineamento della (tipo della) variabile in byte. Per una variabile x:

```
uintptr(unsafe.Pointer(&x)) % uintptr(unsafe.Alignof(x)) == 0
```

Le chiamate a Alignof, Offsetof e Sizeof sono espressioni costanti nel tempo di compilazione di tipo int.

Le funzioni unsafe.Typeof, unsafe.Reflect e unsafe.Unreflect permettono l'accesso nel tempo di esecuzione ai tipi dinamici e ai valori immagazzinati nelle interfacce. Typeof ritorna una rappresentazione di tipo dinamico di val come un runtime.Type. Reflect alloca una copia del valore dinamico di val e ritorna entrambi il tipo e l'indirizzo della copia. Unreflect inverte Reflect, creando un valore interfaccia da un tipo e un indirizzo. Il package reflect costruito su queste primitive fornisce un modo sicuro e più conveniente per ispezionare i valori interfaccia.

4.16.2 Garanzie della dimensione e dell'allineamento

Per i tipi numerici (§ Tipi numerici), le seguenti dimensioni sono garantite:

tipo	dimensione in byte
byte, uint8, int8	1
uint16, int16	2
uint32, int32, float32	4
uint64, int64, float64, complex64	8
complex128	16

Le seguenti proprietà di allineamento minimali sono garantite:

1. Per una variabile x di qualsiasi tipo: unsafe.Alignof(x) è almeno 1.
2. Per una variabile x di tipo struttura: unsafe.Alignof(x) è la più grande di tutti i valori unsafe.Alignof(xf) per ogni campo f di x, ma almeno 1.
3. Per una variabile x di tipo array: unsafe.Alignof(x) è lo stesso come unsafe.Alignof(x[0]), ma almeno 1.

5 Costrutti significativi

Sebbene Go abbia costrutti simili al C, ne sono stati introdotti dei nuovi. Alcuni dei costrutti più rilevanti sono elencati di seguito.

5.1 Tokens

I tokens formano il vocabolario del linguaggio Go. Ci sono quattro classi: identificatori, parole chiave, operatori e delimitatori, e valori. Lo spazio bianco, formato da spazi (U+0020), tabulazioni orizzontali (U+0009), ritorni a capo (U+000D), e nuove righe (U+000A), viene ignorato eccetto quando separa tokens che altrimenti si combinerebbero in un unico token. Inoltre, una nuova linea o la fine del file potrebbe causare l'inserimento di un punto e virgola. La sequenza di caratteri in ingresso è suddivisa in tokens, dove ogni token è la più lunga sequenza di caratteri che formano un token valido.

5.1.1 Identificatori

Gli identificatori sono le essenze del programma come le variabili e i tipi. Un identificatore è una sequenza di una o più lettere e cifre. Il primo carattere di un identificatore deve essere una lettera.

    identificatore = lettera { lettera | cifra_unicode }.

Esempi: un
_x9
ThisVariableIsExported
αβ

Alcuni identificatori sono prefissati.

5.1.2 Parole chiave

Le seguenti parole chiave sono riservate e non possono essere utilizzate come nomi di identificatori.

```
break default func interface select
case defer go map struct
chan else goto package switch
const fallthrough if range type
continue for import return var
```

5.1.3 Operatori e delimitatori

Le seguenti sequenze di caratteri rappresentano operatori, delimitatori, e altri tokens particolari:

```
+    &     +=    &=     &&    ==    !=    ( )
-    |     -=    |=     ||    <     <=    [ ]
*    ^     *=    ^=     ←     >     >=    { }
/    <<    /=    <<=    ++    =     :=    ,    ;
%    >>    %=    >>=    --    !     …     .    :
&^   &^=
```

5.2 Punti e virgola

La grammatica formale utilizza i punti e virgola ";" come terminatori in una serie di istruzioni. I programmi Go possono omettere la maggior parte di questi punti e virgola utilizzando le seguenti due regole:

1. Quando la sequenza di ingresso è suddivisa in tokens, un punto e virgola viene automaticamente inserito nel flusso di token alla fine di una riga non vuota se il token finale della riga è
   - un identificatore
   - un valore intero, a virgola mobile, immaginario, carattere o stringa
   - una delle parole chiave `break` , `continue` , `fallthrough` , o `return`
   - uno degli operatori e delimitatori `++` , `--` , `)` , `]` , o `}`
2. Per consentire istruzioni complesse di occupare una sola riga, un punto e virgola può essere omesso prima di una chiusura ")" o "}" .

Per rispecchiare l'uso del particolare linguaggio, esempi di codice in questo documento sopprimono i punti e virgola con queste regole.

5.3 Valori stringa

Un valore stringa rappresenta una costante stringa ottenuta dalla concatenazione di una sequenza di caratteri. Ci sono due forme: i valori di stringhe raw e i valori di stringhe interpreted.
I valori di stringhe raw sono le sequenze di caratteri tra singole virgolette ' ' . Tra le virgolette, qualsiasi carattere è valido ad eccezione delle singole virgolette. Il valore di una stringa raw è la stringa composta dai caratteri non interpretati tra le virgolette, in particolare, i backslash non hanno alcun significato particolare e la stringa può estendersi su più righe.
I valori di stringhe interpreted sono sequenze di caratteri tra virgolette doppie " ". Il testo tra le virgolette, che non può estendersi su più righe, costituisce il valore della stringa interpreted, con escapes backslash interpretati come lo sono nei valori carattere (tranne che `\'` non è valido e `\"` è valido). Gli escape delle tre cifre ottali (`\` *nnn)* e delle due cifre esadecimali (`\x` *nn)* rappresentano singoli byte della stringa risultante, tutti gli altri escape rappresentano la codifica UTF-8 (possibilmente multi-byte) dei singoli caratteri. Così all'interno di un valore stringa, `\377` e `\xFF` rappresentano un singolo byte di valore `0xFF` = 255, mentre `ÿ` , `\u00FF` , `\U000000FF` e `\xc3\xbf` rappresentano i due byte `0xc3 0xbf` della codifica UTF-8 di carattere U+00FF.

```
valore_stringa = valore_stringa_raw | valore_stringa_interpreted.
valore_stringa_raw = "`" { carattere_unicode } "`" .
valore_stringa_interpreted = `"` { valore_unicode | valore_byte } `"` .
```

```
Esempi:   `abc`    // come "abc"
          `\n
          \n`      // come "\\n\n\\n"
          "\n"
          ""
          "Hello, world!\n"
          "日本語"
          "\u65e5 本\U00008a9e"
          "\xff\u00FF"
```

Tutti questi esempi rappresentano la stessa stringa:

```
"日本語"                // testo di ingresso in UTF-8
`日本語`                // testo di ingresso in UTF-8 come valore stringa raw
```

"\u65e5\u672c\u8a9e" // codice Unicode esplicito
"\U000065e5\U0000672c\U00008a9e" // codice Unicode esplicito
"\xe6\x97\xa5\xe6\x9c\xac\xe8\xaa\x9e" // bytes UTF-8 espliciti

Se il codice sorgente contiene un valore carattere formato da due simboli, come una combinazione che unisce un accento e una lettera, il risultato darà un errore se collocati in un valore carattere (non è un unico simbolo!!) e apparirà come due simboli, senza generare un errore, se collocati in un valore stringa.

5.4 Slice

Una slice è un riferimento a un segmento contiguo di un'array e contiene una sequenza numerata di elementi di tale array. Un tipo slice indica l'insieme di tutte le slice di array di questo tipo di elemento. Il valore di una slice non inizializzata è `nil` .

TipoSlice = "[" "]" TipoElemento .

Come gli array, le slice sono indicizzabili e hanno una lunghezza. La lunghezza di una slice `s` può essere scoperta utilizzando la funzione built-in len(s), a differenza degli array può cambiare durante l'esecuzione. Gli elementi possono essere indirizzati con indici interi tra 0 a `len(s)-1` (§ Indici). L'indice di slice di un dato elemento può essere inferiore rispetto all'indice dello stesso elemento dell'array sottostante.

Una slice, una volta inizializzata, è sempre associata ad un'array di base che contiene i suoi elementi. Una slice condivide pertanto la memoria con il suo array e con altre slice dello stesso array, invece, gli array distinti rappresentano sempre aree di memoria distinte.

L'array di base di una slice si può estendere oltre la fine della slice. La capacità è una misura di tale lunghezza: è la somma della lunghezza della slice e della lunghezza dell'array al di là della slice; una slice di lunghezza fino a quella capacità può essere creata tagliandone una nuova dalla slice originale (§ slice). La capacità di una slice `a` può essere scoperta usando la funzione built-in cap(a).

Un nuovo valore slice inizializzato per un dato tipo di elemento `T` è realizzato utilizzando la funzione built-in make, che prende un tipo di slice e i parametri specificando la lunghezza e, facoltativamente, la capacità:

make([]T, lunghezza)
make([]T, lunghezza, capacità)

La chiamata `make()` alloca un nuovo array nascosto a cui il valore della slice fa riferimento. Cioè, l'esecuzione

make([]T, lunghezza, capacità)

produce la stessa slice allocando un array e tagliando esso, quindi questi due esempi indicano la stessa slice:

make([]int, 50, 100)
new([100]int)[0:50]

Come gli array, le slice sono sempre unidimensionali ma possono essere composte per costruire oggetti di dimensioni superiori. Con array di array, gli array interni hanno, per costruzione, sempre la stessa lunghezza, ma con le slice di slice (o array di slice), le lunghezze possono variare dinamicamente. Inoltre, le slice interne devono essere allocate individualmente (con `make`).

Per una stringa, un'array o una slice a, l'espressione principale

a[low : high]

costruisce una sottostringa o una slice. Le espressioni dell'indice low e high selezionano quali elementi appaiono nel risultato. Il risultato ha indici che iniziano da 0 alla lunghezza uguale a high - low. Dopo aver tagliato l'array a

a := [5]int{1, 2, 3, 4, 5}
s := a[1:4]

lo slice `s` ha tipo `[]int` , lunghezza 3, capacità 4, ed elementi

```
s[0] == 2
s[1] == 3
s[2] == 4
```

Per convenienza, qualsiasi delle espressioni indice può essere omessa. Un indice low mancante è di default imposto a zero; un indice high mancante è di default imposto alla lunghezza dell'operando sliced:

```
a[2:]     // come a[2 : len(a)]
a[:3]     // come a[0 : 3]
a[:]      // come a[0 : len(a)]
```

Per gli array e le stringhe, gli indici low e high devono soddisfare 0 <= low <= high <= lunghezza; per le slice, il limite superiore è la capacità piuttosto che la lunghezza.
Se l'operando sliced è una stringa o una slice, il risultato dell'operazione slice è una stringa o una slice dello stesso tipo. Se l'operando sliced è un array, deve essere indirizzabile e il risultato dell'operazione di slice è una slice con lo stesso tipo di elemento come l'array.

5.5 Mappa

Una mappa è un gruppo non ordinato di elementi di un tipo, chiamato il tipo elemento, indicizzato da un insieme di chiavi univoche di un altro tipo, chiamato il tipo chiave. Il valore di una mappa non inizializzata è `nil` .

```
TipoMappa = "mappa" "[" TipoChiave "]" TipoElemento .
TipoChiave = Tipo .
```

Gli operatori di confronto == e != (§ operatori di confronto) devono essere completamente definiti per operandi del tipo chiave, quindi il tipo chiave non deve essere una struttura, un array o una slice. Se il tipo chiave è un tipo di interfaccia, tali operatori di confronto devono essere definiti per i valori della chiave dinamica; in caso di errore si verificherà un panico nel tempo di esecuzione.

```
map [string] int
map [*T] struct { x, y float64 }
map [string] interface {}
```

Il numero degli elementi di una mappa rappresenta la lunghezza della mappa stessa. Per una mappa `m` , la lunghezza può essere scoperta usando la funzione built-in len(m) e può cambiare durante l'esecuzione. I valori possono essere aggiunti e rimossi durante l'esecuzione mediante forme speciali di assegnazione .
Un nuovo e vuoto valore di mappa è realizzato con la funzione di built-in make, che prende il tipo di mappa e opzionalmente una parte di capacità come argomenti:

```
make(map[string] int)
make(map[string] int, 100)
```

La capacità iniziale non vincola le sue dimensioni: le mappe crescono per accogliere il numero di elementi memorizzati all'interno.
Le mappe possono essere costruite l'usuale sintassi del valore composto con coppie chiave-valore separate dalla virgola, così sono facili da costruire durante l'inizializzazione.

```
var timeZone = map[string] int
{   "UTC":  0*60*60,
    "EST": -5*60*60,
    "CST": -6*60*60,
    "MST": -7*60*60,
    "PST": -8*60*60,
}
```

L'assegnazione e il prelievo dei valori di mappa è sintatticamente lo stesso come si avrebbe fatto per gli array eccetto che l'indice non ha l'obbligo di essere intero.

```
offset := timeZone["EST"]
```

Un tentativo per prelevare un valore di mappa con una chiave che non è presente nella mappa ritornerà il valore zero rispetto al tipo degli elementi della mappa. Per esempio, se la mappa contiene interi, cercando una chiave non esistente ritornerà 0.

5.6 Canale

Un canale fornisce un meccanismo per due funzioni, in esecuzione contemporanea, per sincronizzare l'esecuzione e comunicare passando un valore di un tipo di elemento specificato. Il valore di un canale non inizializzato è `nil` .

```
TipoCanale = ( "canale" [ "<-" ] | "<-" "canale" ) TipoElemento .
```

L'operatore `<-` specifica la direzione, l'inviare e il ricevere del canale. Se nessuna direzione è data, il canale è bidirezionale. Un canale può essere limitato solo per inviare o soltanto per ricevere la conversione o l'assegnazione.

```
chan T            // può essere usato per inviare e ricevere valori di tipo T
chan<- float64    // può essere utilizzato solo per inviare float64s
<-chan int        // può essere utilizzato solo per ricevere int
```

L'operatore `<-` associa il chan più a sinistra possibile:

```
chan<- chan int      // come chan<- (chan int)
chan<- <-chan int    // come chan<- (<-chan int)
<-chan <-chan int    // come <-chan (<-chan int)
chan (<-chan int)
```

Un nuovo valore di canale inizializzato può essere realizzato utilizzando la funzione built-in make, che prende il tipo di canale e opzionalmente una capacità come argomenti:

```
make(chan int, 100)
```

La capacità, in numero di elementi, imposta la dimensione del buffer nel canale. Se la capacità è maggiore di zero, il canale è asincrono: le operazioni di comunicazione hanno esito positivo senza bloccarsi se il buffer non è pieno (invia) o non vuoto (riceve), e gli elementi vengono ricevuti nell'ordine in cui vengono inviati. Se la capacità è pari a zero o assente, la comunicazione ha successo solo quando sia il mittente che il ricevente sono pronti.
Un canale può essere chiuso e testato per la chiusura con le funzioni built-in close e closed.

5.7 Goroutines

Sono chiamate goroutines perché i termini esistenti – threads, coroutines, processi, così via – comunicano un significato inadeguato. Una goroutine ha un modello semplice: è una funzione eseguita in parallelo con altre goroutines nello stesso spazio di indirizzamento. Gli stack inizialmente sono piccoli, quindi economici, e crescono allocando (e liberando) memoria heap come richiesto.
Le goroutines sono multiplexate su thread multipli così se uno si deve bloccare, come quando si aspetta un'operazione di I/O, gli altri thread possono continuare l'esecuzione. Questo design nasconde molte delle complessità della creazione e gestione del thread.
Si può premettere una chiamata di funzione o metodo con la parola chiave go per eseguire la chiamata in una nuova goroutine. Quando la chiamata è esaurita, la goroutine finisce, silenziosamente. (L'effetto è simile alla notazione & dello shell Unix per eseguire un comando in background.)

```
go list.Sort()     // esegue list.Sort in parallelo; non aspetta.
```

Un valore di funzione può essere utile in un'invocazione di goroutine.

```
func Announce(message string, delay int64)
{    go func()
     {        time.Sleep(delay)
              fmt.Println(message)
     }()      // Nota le parentesi – devono chiamare la funzione.
}
```

In Go, i valori di funzione sono le conclusioni: l'implementazione assicura che le variabili indirizzate dalla funzione sopravvivono finché sono attive.
Questi esempi non sono tanto pratici perché le funzioni non hanno modo di segnalare il completamento della loro esecuzione. Per quello, c'è bisogno dei canali.

6 Applicazioni significative

Le applicazioni significative sono state sviluppate dal GO Dev Team. Alcune soluzioni adottate da questo team di sviluppo sono le stesse che si trovano nelle rom MIUI, ideale per chi non ha un telefono supportato da queste rom ma vuole comunque goderne (in parte) le comodità.
Per il momento sono stati sviluppati un launcher, un sms manager, meteo e clock widget, rubrica e gestione contatti. Di seguito queste applicazioni sono descritte nel dettaglio.

Illustrazione 6.1: logo Go Dev Team

6.1 GO Launcher Ex

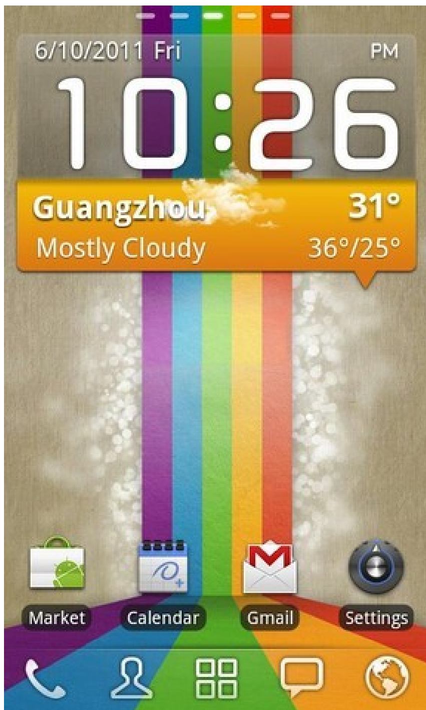

Illustrazione 6.1.1: immagine di Go Launcher Ex

GO Launcher EX è la versione estesa di GO Launcher, una delle più popolari applicazioni presenti nell'Android Market. È un'applicazione altamente personalizzabile, che si attiva quando viene premuto il tasto HOME del telefono Android.
GO Launcher EX supporta centinaia di temi, si lancia molto velocemente ed ha una grande quantità di funzioni utili.
Per alcuni versi è molto simile a quello di default che troviamo nelle rom MIUI, come del resto tutte le applicazioni GO, questa ovviamente è provvista di drawer e gode della possibilità di poter personalizzare le singole icone (pesca le icone dai temi del launcher oppure dalle cartelle che abbiamo nel telefono), ridimensionare i widget dopo averli inseriti nella home; quindi si possono allargare i widget oltre la misura per la quale sono stati progettati, effetti di transizione e tante altre opzioni di personalizzazione.
Vale la pena di spendere due parole anche sul drawer: tenendo premuto su un'icona di una applicazione si entra in modalità editing, quindi si può cambiare l’ordine delle icone, creare cartelle esattamente con la stessa gesture di iOs e disinstallare le applicazioni premendo sulla X rossa su ogni singola icona, sempre come iOs. Copiata si, ma come funzione è davvero comoda.

Abbinato a GO Launcher EX Notification avremo anche le notifiche con il contatore sulle icone per le chiamate perse e sms o email non lette.
Caratteristiche:

1. Centinaia di temi.
2. Transizione dello schermo home: animazioni estremamente armoniose con il cambiamento degli schermi.
3. Facilita l'esperienza dello scrolling.
4. Menù delle icone popup: operazioni veloci per le icone e i widget dello schermo home, sostituire icone, rinominare e disinstallare applicazioni in un click.
5. Applicazione drawer: funzioni utili come cartelle, task manager, disinstallazione veloce e storia sull'utilizzo dell'applicazione.
6. Gesture supportato: segni operativi sopra e sotto differenti definiti dall'utente, come l'apertura del menù, della scheda comunicazioni e dell'applicazione specifica.
7. Scorribilità e ridimensionamento dei widgets supportata.
8. Schermo di preview: modo veloce di sistemare la posizione dello schermo, aggiunge schermi e

imposta lo schermo home.
Funzioni di punta:
- Premere con un dito per entrare nello schermo di preview e rivedere le pagine dello schermo home.
- Cercare “go launcher theme” per ottenere i temi e selezionarli in “Themes preferences”.
- Imposta la transizione dello schermo e la velocità dell'attivazione dell'applicazione in “Display settings”.
- Forte pressione delle icone nello schermo home per attivare i menù popup.
- Forte pressione delle icone nell'applicazione drawer per rivedere la posizione, disinstallare le applicazioni e creare cartelle.

6.2 GO SMS Pro

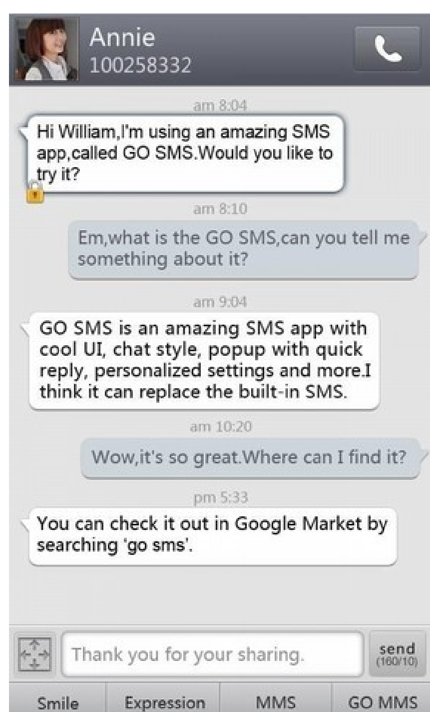

Illustrazione 6.2.1: immagine di Go Sms Pro

Questa versione appena rilasciata andrà a rimpiazzare totalmente l’attuale applicazione GO Sms.
L’interfaccia utente è stata completamente rivista. Il tema di default molto elegante ed “essenziale” dà un tocco di stile “Apple” a questa applicazione che si arricchisce di grossissime novità. A discapito del nome, l’applicazione anche se versione PRO resta comunque del tutto gratuita!
GO SMS Pro permette un enorme aumento sulle caratteristiche di applicazioni di messaggistica, molto più veloce, ha più funzioni: con UI fresche, stile lista/chat, popups, backup/restore, scheduler, codificazione di dati, blacklist, cartelle, molti temi, una grande quantità di impostazioni, include quasi tutte le funzioni possibili per il texting, ecc..

Le caratteristiche di questa applicazione sono le seguenti:
- Supporta Emoji con Plugin Emoji.
- Elenco cartelle supportate: Inbox, Outbox, MMS, cartelle create, cartelle criptate.
- Supporta GESTURE: nella pagina dell'elenco messaggi, scorrendo con il dito sullo schermo accediamo all’elenco cartelle. Le transizioni sono impostabili in Impostazioni->Impostazioni transizioni.
- Supporto ai temi DIY. Molti temi nuovi, supporta anche il tema DIY, il Pluginallpaper Maker, aspetto totalmente personalizzabile, supporta stile chat e stile lista.
- Supporto Multi Lingue indipendente all’interno dell’applicazione.
- Supporto totale SMS/MMS, anche per l'integrazione della Chat di Facebook con Pluginchat GO FB.
- Supporto foto di contatto di Facebook.
- Supporto allo stile Chat ed Elenco Chat.
- Pop-Up con risposta rapida, modalità privacy.

- Servizio GO-MMS, si possono inviare foto/musica a un amico attraverso un SMS con 2G/3G/4G o WIFI.
- L'applicazione di Stock può essere rimossa (con particolare attenzione).
- Widget 1×1 con conteggio SMS Ricevuti.
- Widget 4×2 con elenco messaggi.
- Supporto totale alle personalizzazioni utente.
- Modalità batch (aggiunge contatti, cancella conversazioni/messaggi e backup)
- Backup e restore SMS, supporta formato XML, invia file di backup per email.
- Impostazioni di backup e di restore
- Blacklist per il blocco messaggi.
- Supporto localizzazione per caratteri accentati senza usare Unicode: caratteri accentati come á í, prendendo più spazio di un carattere, fissato abilitando la modalità SMS Czech, Polish e French.
- Evita messaggi disordinati: ordinati per data o per messaggi inviati/ricevuti.
- Raggruppamento messaggi (modalità Gruppo o modalità Separato).
- Notifiche con modalità privacy e notifiche di promemoria.
- Impostazioni di sicurezza.
- Ecc....

6.3 GO Weather

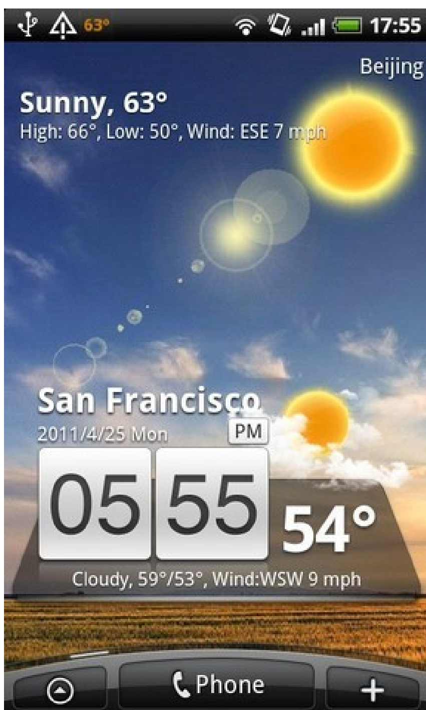

Illustrazione 6.3.1: immagine di Go Weather

Meteo, ecco quello che mancava, un classico widget con orario e meteo, un pò come quello di HTC Sense come impostazione, ma con una marcia in più. Con un tap sul widget si entra nella sezione meteo, che ha delle gradevoli animazioni sullo sfondo e fornisce dei dati sulla situazione meteorologica molto dettagliati, peccato solo che siano in inglese.
C'è anche la possibilità di scaricare le animazioni in HD, ma sono molto pesanti da scaricare, e il server pare molto lento.
Go Weather è una nuova applicazione sulle condizioni atmosferiche per telefoni Android. Si presenta con un'interfaccia utente pulita e ordinata, con un'animazione semplice che dà un'esperienza utente condivisa.
Go Weather produce le più accurate informazioni sulle condizioni atmosferiche di qualsiasi luogo in qualsiasi ora. Con l'accesso alla rete più grande di stazioni meteorologiche professionali negli USA e decine di migliaia di locazioni in tutto il mondo, Go Weather è un'applicazione sulle condizioni atmosferiche quasi-libera che include video sorprendenti ad alta definizione, portando la propria conoscenza delle condizioni atmosferiche a un più alto livello.

Caratteristiche:
- Condizioni atmosferiche mondiali: coprendo decine di migliaia di regioni e città nel mondo, fornisce informazioni sulle condizioni atmosferiche in tempo reale e previsioni fino a 5-10 giorni di distanza.
- Interfaccia visiva eccellente: video sulle condizioni atmosferiche ad alta definizione 3D splendidi, bellissimi desktop widget e wallpaper in tempo reali inclusi con la perfetta

combinazione di condizioni atmosferiche.

- Alto impatto visivo dell'occhio umano e animazioni perfezionate.
- Accesso veloce su località salvate (massimo 10).
- Dati sulle condizioni atmosferiche salvate su memoria per osservazioni offline.
- Modi per aggiungere più città: si può effettuare una ricerca, posizionarsi con il GPS oppure altri modi per aggiungere informazioni sulle condizioni atmosferiche di una città.
- Ricordi delle condizioni atmosferiche speciali: interfaccia semplice, interazione facilitata, notifiche della temperatura incluse.
- Preferenze: supporto individuale delle impostazioni come Units e Update Interval.
- Bassa potenza e basso flusso: supporto sul controllo a bassa potenza e i dati sono disponibili offline.

6.4 GO Contacts

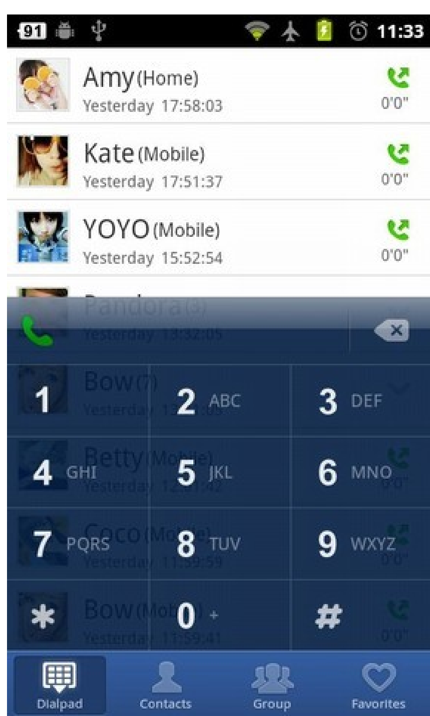

Illustrazione 6.4.1: immagine di Go Contacts

Soprattutto per chi elimina HTC Sense dal proprio telefono, si accorgerà delle limitazioni del gestore contatti originale di Android. I tempi cupi sono finiti, ora grazie a quest'ultima applicazione è tutto più facile. La composizione rapida è efficacissima come quella di Sense, si può comporre il numero direttamente, oppure mentre digitiamo vengono ricercati i contatti in rubrica in base alle lettere corrispondenti dei tasti che stiamo premendo. Altrimenti c'è la classica ricerca contatti, con tutto l'alfabeto disposto in verticale sulla destra dello schermo, dove basta passare con il dito sull'iniziale per trovare in pochissimi secondi il contatto desiderato. Favoriti, gruppi di contatti (sincronizzabili con Google Contacts) e sincronizzazione con i vari account Google, Facebook, Twitter, ecc.
GO Contacts è uno strumento potente, sostituzione ideale della famiglia Contacts e Dialer.

Caratteristiche:

1. Ricerca veloce: per qualsiasi lettera, nome, parola chiave.
2. Raggruppamento contatti: Drag&group; raggruppare SMS o email premendo un solo tasto.
3. Fusione contatti duplicati identificati dal nome o dal numero di telefono.
4. Smart dialing: Solo una parte di numeri o nomi di contatto ti ricordi? Nessun problema, premendo solo alcuni tasti, essa ti darà le combinazioni.
5. Dial veloce: IP Dial supportato.
6. Tema supportato: Dark (incluso) e Spring, Ice Blue (scaricabile separatamente).
7. Backup / ripristino: salva i contatti o li ripristina dalla SD card in modo molto sicuro.
8. Attribuzione numero: mostra la locazione registrata del numero di telefono (solo disponibile nella versione cinese, Plugin scaricabili separatamente).

6.5 GO Keyboard

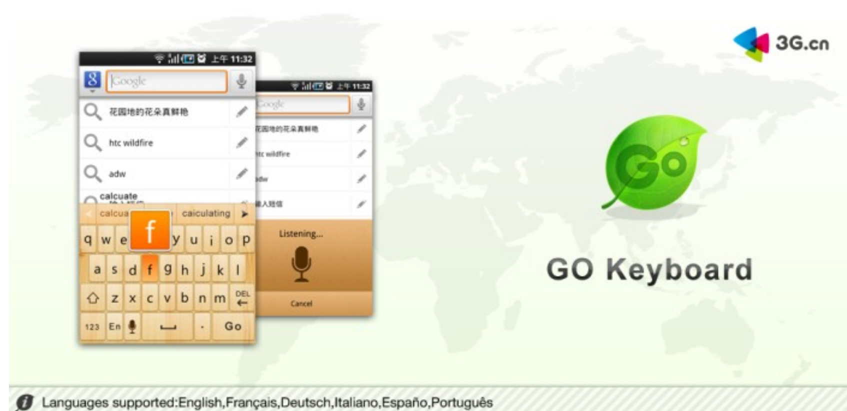

Illustrazione 6.5.1: immagine di Go Keyboard

GO Keyboard permette di scrivere a tastiera in modo molto veloce. È una scelta obbligata del telefono mobile Android.

Principali caratteristiche:

1. Ingresso multi-modo: supporta pienamente Pinyin, Stroke, English, numeri, simboli e altri metodi di ingresso.
2. Interfaccia controllo utente unica: lente di ingrandimento, anello di selezione scatola, menù curvato, trackball virtuale, ecc.
3. Correzione errore: piena correzione dell'errore all'ingresso della tastiera.
4. Aspetto fresco: scelta intelligente dell'aspetto, facilmente modifica lo sfondo della tastiera.
5. Raccolta contatti: velocità di ingresso del numero dei contatti, del nome del contatto dopo l'importazione e il numero è auto-associativo.
6. Supporta molti linguaggi e disposizioni di tastiera.
7. Contiene dizionari accurati.
8. Importazione degli SMS.

6.6 GO Score

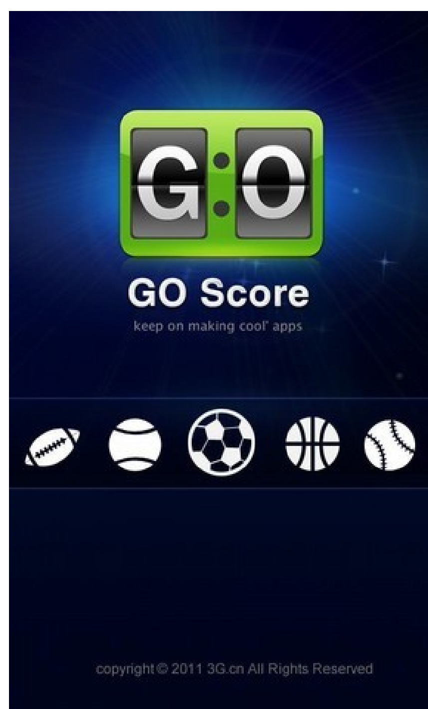

Illustrazione 6.6.1: immagine di Go Score

Arriva una nuova applicazione dal sempre apprezzato GO Team: GO Score. Si tratta di un programma, realizzato molto bene e con una grafica facile e piacevole, che permette di monitorare i dati sportivi di tutto il mondo in pochi click. GO Score è gratuito come da tradizione GO Team.
GO Score fornisce la situazione dei punteggi in tempo reale, delle quotazioni e delle squadre nelle partite di tutto il mondo.

Caratteristiche chiave:

1. Sia punteggi in tempo reale che risultati di partite passate: punteggi istantanei forniti in modo completo e professionale.
2. Panoramica dei dati delle quotazioni da sorgenti multiple: fornisce le statistiche complete sulle quotazioni e sulle società di vari sport.
3. Chat in tempo reale con altri fans di sport: sede di discussioni in rete fornisce una piattaforma per interagire e condividere le opinioni sulle partite.
4. Schede sulle posizioni e sui punteggi: fornisce le schede sulle più recenti posizioni e punteggi dei più importanti campionati di vari sport.

5. Partita da seguire: si può seguire la propria partita preferita e imposta dei segnali quando la partita inizia, finisce, quando si verificano goal e cartellini rossi (squilla o vibra), così che non si perderanno i punteggi o qualsiasi stato della partita.

Sport coperti:
* Basket：NBA, NCAA, NCAAW.
* Baseball：MLB.
* Calcio：EPL，Premiership, Champions League, Serie A, La Liga, MLS.
* Hockey：NHL，DEL，LM，ALL，ELS.
* Tennis：Wimbledon，Australian，France，US Open.

6.7 GO Wallpaper

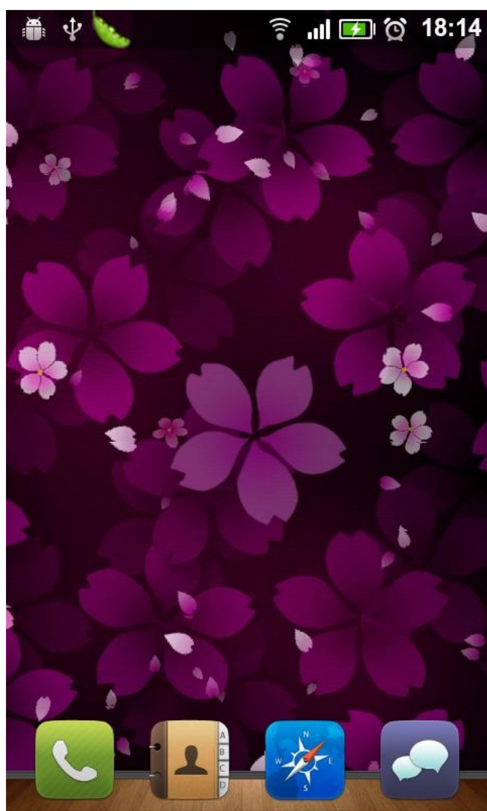

Illustrazione 6.7.1: un wallpaper di Sakura Falling

Go Wallpaper ha creato diversi temi per le applicazioni sviluppate in precedenza e diversi live wallpaper con un aspetto fantascientifico e una varietà di opzioni. Tra i vari prodotti sviluppati alcuni esempi sono “GO SMS Summer Theme”, “GO Contacts Future Theme”, “Sakura Falling Live Wallpaper”, “Hyperspace 3D Live Wallpaper”, “3D Skyrocket Live Wallpaper”, ecc.

Il primo wallpaper in tempo reale di Go Wallpaper Dev Team è “Sakura Falling Live Wallpaper” (vedi *Illustrazione 6.7.1* a fianco) che è rappresentato con un fiore cadente lentamente e leggermente, bellissimo ed elegante con uno spizzico di vento. “Sweet Heart Live Wallpaper” è un sorprendente wallpaper in tempo reale di cuori brillanti e movimentati che può essere usato per la propria innamorata riempito di dolcezza e romanticismo.

Caratteristiche:
1. Panoramica splendida: combinazione perfetta di tecnologia e arte.
2. Impostazioni su misura: incontrare i bisogni di utenti individuali.
3. Multiple scelte: dozzine di prodotti per i clienti.

Conclusioni

Le caratteristiche di questo linguaggio sono velocità, semplicità, similarità al C, concorrenza, multi-core, multi-processing, sicurezza, gestione della memoria basata sulla garbage collection.
Nonostante queste premesse e il rilascio in modalità open-source, il linguaggio Go non ha avuto le aspettative di crescita che tra i più ottimisti di casa Google prospettavano. Il mancato successo potrebbe essere dovuto a diversi motivi: la grande varietà di linguaggi presenti, l'esistenza di alcuni costrutti nuovi di difficile comprensione, un gran riutilizzo di costrutti presenti in C e in Python aggiungendone dei nuovi che rendono Go un linguaggio parzialmente nuovo.
Il capitolo "Le specifiche del linguaggio" non contiene una guida completa di Go ma solo una raccolta dei costrutti basilari e più innovativi.
Attualmente, sono state sviluppate delle applicazioni con il linguaggio Go dal GoDevTeam per telefoni "mobile" in cui è installato Android e dal GoSoftWorks (GoSatWatch, GoSkyWatch Planetarium, GoAtlantis) per iPhone e iPod touch. Finora non sono state sviluppate applicazioni rilevanti in linguaggio Go per pc.