Scriviamo un Interprete in Go!

Lexer

Il lexer ha il compito di tradurre il codice sorgente di un programma in un array di token - senza attribuire a questi un significato semantico d'insieme, quindi si occupa di verificare che il token sia valido - struttura più facile da manipolare per il parser.

un esempio preso direttamente dal libro:
data la stringa in input “let x = 5 + 5;”

Il lexer produrrà il seguente array di token:
[LET, IDENTIFIER(“x”), EQUAL_SIGN, INTEGER(5), PLUS_SIGN, INTEGER(5), SEMICOLON].

I Token rappresentano ogni elemento sintattico presente, quindi gli spazi vuoti non rappresentano Token, bensì degli elementi che garantiscono maggiore leggibilità; in questo caso scrivere "let x = 5 +5" o "let x = 5 + 5" è completamente indifferente per il Lexer.

NOTA: È compito dei Lexer quello di associare ad ogni Token il numero di riga e il sorgente, in cui il Token è scritto, fornendo così una diagnostica degli errori decisamente più corretta, tuttavia questa caratteristica non viene implementata nel nostro interprete.

Struttura dei Sorgenti:

Possiamo strutturare i sorgenti, inizialmente, in due principali classi, o pseudo tali, in quanto il Go non è un linguaggio di programmazione ad oggetti class-based, ma usa le strutture e un sistemi di riferimenti ad essa per incapsulare i metodi.

Ad esempio:

```
public c        lass Ex {

                private int field;

                public void myMethod() {
                //...

                }

}
```

Diventa

```
type Ex struct {
      field int

}

func (e *Ex) myMethod() {

}
```

myMethod fa ora parte del cosidetto method set e l'identificatore "e" identifica il method receiver che deve essere un puntatore, grazie al quale possiamo accedere al field dell'istanza specifica.

Le "classi" che creiamo possono chiamarsi Lexer e Token.

Token è struttura composta da due campi, ovvero il tipo di token, che per semplicità possiamo dichiararlo come stringa e il letterale associato, anch'esso stringa. Inoltre il sorgente che contiene la "classe" Token deve definire anche delle costanti, le quali indentificano i vari token, come ad esempio operatori aritmetici-logici, keywords per la dichiarazioni di variabili, parentesi, EOF e un valore ILLEGAL per notificare un errore. Per il momento Token non presenta metodi.

per dichiarare una costante in Go si usa la keyword const, a cui associamo un identificatore, un tipo opzionale e infine un valore. Esempio: const MUL string = "*", oppure const MUL = "*". Il Go per rendere più leggera la lettura di un sorgente con numerosi constante permette di insere quest'ultime in uno blocco di definizone, consentendo di scrivere solo gli identificatori e i valori.

Esempio: const (

    SUM = "+"

    MUL = "*"

)

Lexer è una struttura che contiene 4 campi: il primo è un campo di tipo stringa che identifica il brano di codice da analizzare, il secondo e il terzo invece sono campi di tipo intero che rappresentano degli indici che consentono di scorrere la stringa di input, infine l'ultimo campo identifica il carattere che stiamo attualmente leggendo dalla stringa di input, il campo in questione è un byte, quindi in Go un intero a 8 bit, che non rappresenta tutti i caratteri UTF-8. Lexer, a differenza di Token, contiene dei metodi, questi, oltre al costruttore, sono readChar, NextToken e newToken (da notare che idiomaticamente in Go i metodi pubblici devono iniziare con la maiuscola, ovviamente è solo una convenzione).

readChar() inserisce nel campo ch (quello che identifica i caratteri della stringa di input) il Token da analizzare, incrementando anche gli indici in vista di aggiornamenti successivi del campo.

NextToken() chiama il metodo readChar() che restisuisce il Token da analizzare, si entra quindi in un costrutto switch che scorrendo le costanti della "classe" Token aggiunge ogni token che incontra chiamando il metodo newToken()

newToken() deve avere come parametri il tipo di Token, quindi il valore delle costanti della "classe" Token, e il letterale del Token, quindi il valore del campo ch. Il metodo crea tante nuovi valori di tipo Token (che suppongo debbano essere memorizzati da qualche parte).

Il compito del Lexer, come detto in precedenza, è quello di identificare i token, questi non possono essere per forza delle keyword oppure operatori, ma possono essere anche identificatori. Occorre quindi aggiungere una clausola default al costrutto switch che deve riconoscere se l'input è un normale carattere, come per esempio un nome di variabile, oppure un numero (per il momento solo interi). Se invece non corrisponde a nessuno degli elementi precedenti allora lo imposta come illegale. Abbiamo quindi bisogno dei seguenti metodi:

isLetter(l.ch), prende il valore di input e controlla se corrisponde alle lettere dell'alfabeto, cioè se è incluso tra A e Z, quest'ultime anche in minuscolo. Questo metodo restituisce un boolean.

isDgiti(l.ch), prende il valore di input e controlla se è un numero intero senza segno, quindi se è incluso tra lo 0 e il 9. Anche questo metodo restituisce un boolean.

readIdentifier() , se chiamato consente prendere i caratteri successivi se si incontra un carattere, ad

esempio se il programmatore dichiara una variabile con più lettere, ad esempio "addendo", la funzione chiama, finchè incontra lettere, readChar(), quindi aggiorna gli indici per catturare l'ultimo carattere, quindi la lettera "o", di "addendo"; infine restituisce una stringa contente l'intero nome della variabile.

readNumber(), questa funzione svolge un compito del tutto simile a readIdentifier(), ma legge dei numeri. Se il programmatore ad esempio scrive "let addendo = 560", la funzione readNumber() chiama la funzione readChar() per scorrere gli indici, finchè incontra dei numeri.

NOTA: i due precedenti metodi servono solo per impostare il letterale, occorre però indicare anche che tipo di Token sia quello appena letto. È sufficiente aggiungere le costanti definite in precedenza, per i numeri possiamo inserire la constante INT e per gli identificatori IDENTIFICATOR.

Il codice scritto finora contempla, ad eccezione dei numeri e identificatori, solo operatori composti da un singolo carattere, come per esempio le parentesi e l'operatore di assegnamento, tuttavia vi sono operatori che sono composti da più di un carattere come l'operatore di uguaglianza e di disuaglianza, ci serve quindi definire una funzione che prenda un carattere aggiuntivo, da chiamare quando le clausole "=" e "!" dello switch statement si avverano, in questo modo possiamo sapere, predendo il carattere successivo, se piuttosto che essere un operatore di assegnamento è un operatore di uguaglianza. Questa funzione possiamo chiamarla peekChar() che utilizzando l'indice readPosition, ovvero quello che punta alla posizione successiva rispetto all'attuale, restituisce il carattere successivo.

Per completare la lista di identificatori possiamo creare una mappa K-V che contiene come chiavi gli identificatori scritti secondo la sintassi del linguaggio di cui facciamo il parsing, e come valori i letterali definiti come costanti, ad esempio FUNCTION = "FUNCTION".

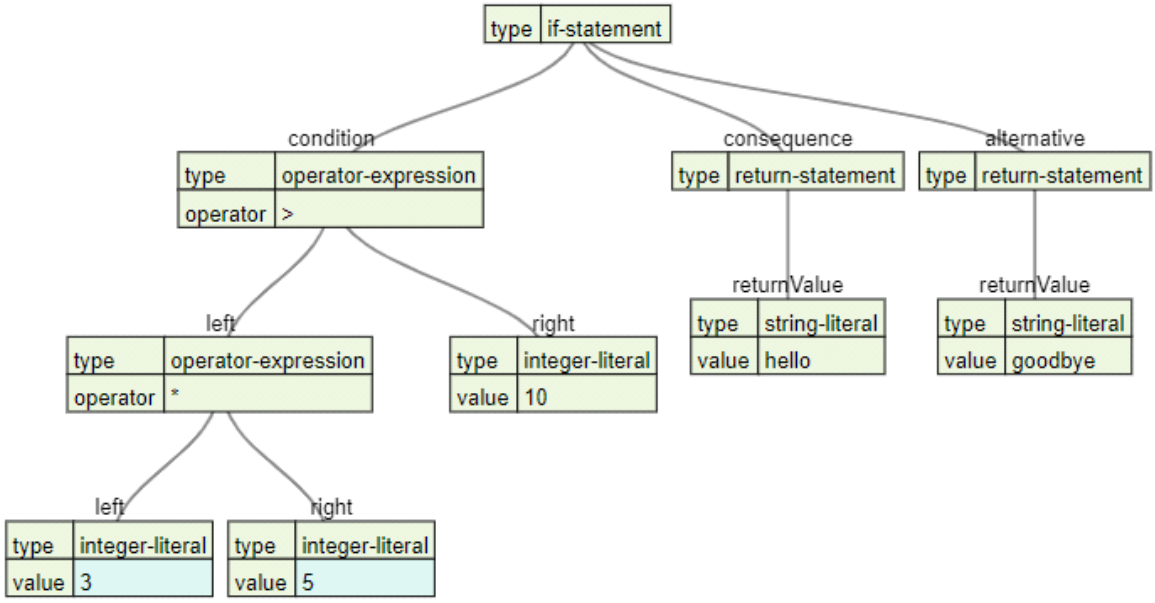
Infine, per completare il Lexer, occorre implementare un REPL(Read Eval Print Loop), ovvero un loop che legge da terminale la stringa di input, contentente il sorgente da analizzare. Una volta acquisiti i dati da terminale occorre chiamare il metodo "costruttore" New() che istanzia un nuovo Lexer e che permette di chiamare il metodo NewToken(), da inserire in un loop, che effettua il lexing dell'input finchè non s'incontra il Token EOF.

Parser

Il compito del parser è trasformare l’array di token prodotto dal lexer in una rappresentazione più strutturata ed utile alla computazione, nel caso specifico un Abstract Sintax Tree (AST), un tipo di albero.

sempre riprendendo un esempio dal libro:

la stringa di codice “if (3 * 5 > 10) { return "hello"; } else { return "goodbye"; }”, dopo essere stata tokenizzata dal lexer, viene trasformata dal parser in un oggetto di questo tipo:



Evaluator

Infine, il compito dell'evaluator è quello di prendere un AST in ingresso e restituire un risultato.
nel caso dell'AST risultante dall'operazione del paragrafo precedente il risultato della valutazione sarà la stringa "hello", essendo il risultato della moltiplicazione 3 * 5 maggiore dell'integer-literal 10 forzando quindi la scelta del ramo "consequence" del nodo di tipo if-statement.