Politecnico di Bari

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica

Formal Languages and Compilers

C2Go:
Transpiler da C a Golang

Docenti:
Prof. Floriano Scioscia

Componenti:
Salvatore Bufi
Angela Di Fazio

Anno Accademico 2020–2021

Indice

Elenco delle tabelle

Elenco dei codici sorgente

Capitolo 1

Introduzione

Lo scopo di questo progetto è quello di realizzare un transpiler che permetta di tradurre un sottoset di istruzioni del linguaggio C al linguaggio Golang.

Il compilatore è stato realizzato in linguaggio C con l'ausilio del generatore automatico di scanner *Flex* e del generatore automatico di parser *Bison*.

1.1 Restrizione sul Linguaggio Sorgente

Poiché il design di un compilatore, per un intero linguaggio di programmazione, è un processo laborioso, il nostro compilatore opera solo su un sott'insieme del linguaggio C. In particolare, la restrizione del linguaggio sorgente, concordata per il nostro progetto, prevede le seguenti funzionalità e istruzioni:

Tipi di dato valori numerici interi con segno a 32 bit, valori in virgola mobile a singola precisione (float - 32 bit) e doppia precisione (double - 64 bit), stringhe gestite unicamente in relazione alle funzioni built-in `scanf` e `printf`.

Tipi di variabili variabili e vettori numerici.

Operatori aritmetici somma, sottrazione, moltiplicazione, divisione.

Operatori di confronto maggiore, minore, maggiore o uguale, minore o uguale, uguale e diverso.

Operatori logici and, or e not.

Altri operatori operatore di referenziazione *&* (previsto solo in presenza della funzione `scanf`) e operatore di assegnazione.

Commenti singola e multi linea.

Dichiarazione singola di variabili e vettori.

Dichiarazione multipla di variabili e vettori.

Dichiarazione con assegnazione (singola e multipla) esclusivamente di variabili.

Assegnazione di un'espressione ad una variabile o ad un elemento di un vettore.

Istruzione di ritorno con e senza valore.

Istruzione di diramazione costrutto `if-else`.

Istruzione di iterazione costrutto `for`.

Istruzione di input tramite funzione built-in `scanf`.

Istruzione di output tramite funzione built-in `printf`.

Chiamate di funzioni non essendo previsti i prototipi di funzioni, le dichiarazioni delle stesse devono precedere le chiamate corrispondenti; fanno eccezione le chiamate alle funzioni `scanf` e `printf`.

Espressioni fanno uso degli operatori logici, aritmetici e di confronto e prevedono la possibilità di utilizzare variabili, elementi di vettori e chiamate a funzioni.

1.2 Struttura del Progetto

Il progetto è composto da diversi file, elencati di seguito:

- `global.h` è il file contenente le variabili globali e i prototipi delle funzioni utilizzate in più file (ad esempio la definizione delle costanti per i colori dei testi e i prototipi delle funzioni per la creazione e la presentazione dei messaggi di errore).
- `scanner.l` è il file contenete le specifiche per la generazione dello scanner mediante il tool Flex. In aggiunta, contiene le definizioni delle funzioni che permettono di gestire la creazione e la presentazione dei messaggi di errore.
- `parser.y` è il file contenente le specifiche necessarie a generare il parser mediante il tool Bison. In aggiunta, contiene il main e le funzioni di supporto alla creazione e chiusura degli scope.
- `tree.h` e `tree.c` sono i file contenenti la struttura dei nodi dell'Abstract Syntax Tree e le funzioni che ne permettono la creazione e la modifica dei loro attributi.
- `symtab.h` e `symtab.c` sono i file contenti la struttura delle Symbol Table e le funzioni che permetto la gestione delle tabelle e dei loro simboli. Inoltre, contengono le strutture e le funzioni per gestire la gerarchia delle tabelle in accordo con gli scope.
- `semantic.h` e `semantic.c` sono i file contenti le funzioni relative ai controlli semantici implementati dal compilatore.
- `translate.h` e `translate.c` sono i file contenti le funzioni utilizzate durante la traduzione.
- `uthash.h` è una libreria usate per implementare hash table in C. È utilizzata come supporto per la gestione delle Symbol Table.

Capitolo 2

Descrizione del Progetto

In questo capitolo verranno descritte le fasi di un compilatore, ponendo particolare attenzione alle strutture dati utilizzate.

2.1 Analisi Lessicale

La prima fase di un compilatore consiste nell'analisi lessicale. In questa fase lo scanner analizza il testo in input e produce una sequenza di token per ciascun lessema riconosciuto. Per generare lo scanner è stato utilizzato il tool Flex (Fast LEXical analyzer generator).

Il file `scanner.l` è l'input per Flex, di seguito sono analizzate solo le parti di tale file che necessitano di un approfondimento.

2.1.1 yylineno

È stata specificata l'opzione `%option yylineno` per tenere traccia del numero di linea corrente in modo automatico, mantenendo tale valore nella variabile `yylineno`. Quest'ultima è stata utilizzata per rendere più accurata la diagnosi degli errori, indicando il numero di linea nei messaggi di errore.

2.1.2 copy_line()

In caso di errore, per poter mostrare all'utente il contenuto della linea in cui tale errore si è verificato, si è reso necessario l'utilizzo della funzione `copy_line()`. Tale funzione è richiamata in corrispondenza della regola

```
^.*                         { copy_line(); }
```

Codice Sorgente 1: `scanner.l` - chiamata alla funzione `copy_line()`

Questa funzione copia la linea, contenuta in `yytext`, all'interno della variabile globale `line`, che sarà svuotata all'inizio di ogni nuova linea. Successivamente, il contenuto di `yytext` sarà restituito al buffer di input, mediante la funzione

`yyless(0)`. In questo modo l'intera linea sarà sottoposta nuovamente alla fase di scanning cercando la corrispondenza con le regole successive.

2.1.3 Attributi dei Token

Ai token è possibile associare un attributo che ne rappresenta il valore. In particolare è stato scelto un unico tipo di attributo: un puntatore a carattere `char *s`. Tale attributo è valorizzato solo per i token relativi agli identificatori, costanti numeriche, stringhe e funzioni built-in `printf` e `scanf`. Si è scelto di memorizzare il valore delle costanti numeriche sotto forma di stringa e la loro conversione è effettuata, solo ove necessaria, nella fase di analisi semantica.

2.1.4 Stati Aggiuntivi

In aggiunta allo stato `INITIAL`, definito di default da Flex, abbiamo definito due stati esclusivi:

- `COMMENT` è utilizzato per il riconoscimento dei commenti multilinea.
- `DQUOTE` è utilizzato per il riconoscimento delle stringhe.

L'utilizzo degli stati permette di rilevare errori quali la mancata chiusura di un commento o di una stringa, nel momento in cui viene raggiunta la fine del file all'interno di tali stati. In generale, per quanto riguarda la gestione dei commenti, si è scelto di ignorare i commenti inseriti all'interno del programma sorgente mentre il contenuto delle stringhe viene passato al paser come attributo del token `STRING`.

```
"/*"                        { BEGIN COMMENT; }
<COMMENT>[^*\n]*            { }
<COMMENT>"*/"               { BEGIN INITIAL; }
<COMMENT><<EOF>>            { yyerror("unterminated comment"); BEGIN
↪  INITIAL; }
```

Codice Sorgente 2: `scanner.l` - regole per il riconoscimento dei commenti multilinea

```
\"                          { BEGIN DQUOTE; }
<DQUOTE>([^"\\\n]|\\.)*     { yylval.s = strdup(yytext); return STRING;
↪  }
<DQUOTE>\"                  { BEGIN INITIAL; }
<DQUOTE>\n |
<DQUOTE><<EOF>>             { yyerror("missing terminating \"
↪  character"); BEGIN INITIAL; }
```

Codice Sorgente 3: `scanner.l` - regole per il riconoscimento delle stringhe

2.2 Analisi Sintattica

Lo scopo della fase di analisi sintattica consiste nello stabilire se la sequenza di token ricevuta in ingresso è una frase ammessa dalla sintassi del linguaggio. Per il nostro progetto si è scelto di avvalersi del parser generator Bison.

Il file `parser.y` è l'input per Bison e contiene le produzioni che esprimono la sintassi della restrizione del linguaggio sorgente, in *Backus-Naur form*. Di seguito sono descritte solo le parti di interesse.

2.2.1 Produzioni per gli Scope

La produzione iniziale consiste nella produzione `program`:

```
program
    : { scope_enter(); } global_declaration_list
    ↪  { root = $2; scope_exit(); }
    ;
```

Codice Sorgente 4: `parser.y` - produzione `program`

Come si può notare essa è composta solo da `global_declaration_list`:

```
global_declaration_list
    : global_declaration
    | global_declaration global_declaration_list
    ;

global_declaration
    : var_declaration
    | func_declaration
    ;
```

Codice Sorgente 5: `parser.y` - produzioni per le dichiarazioni globali

Ciò determina che nello scope globale possono essere presenti solo dichiarazioni di variabili e funzioni.

Le azioni associate alla produzione `program`, oltre a salvare la root dell'Abstract Syntax Tree nella rispettiva variabile globale root, utilizzata per attraversare l'albero nella fase di generazione del codice, effettuano l'apertura e la chiusura dello scope globale mediante le funzioni `scope_enter()` e `scope_exit()`.

La funzione `scope_enter()` crea una nuova symbol table associandone un intero che identifica lo scope corrente e linkandola alla symbol table precedente. Mentre la funzione `scope_exit()` si occupa di eliminare la symbol table corrente all'uscita dallo scope. Le due funzioni si occupano inoltre di incrementare e decrementare rispettivamente il numero identificativo dello scope corrente, inizializzato a zero per lo scope globale.

Tali funzioni gestiscono gli scope dell'intero programma. Esse vengono richiamate in corrispondenza delle produzioni per la gestione degli scope:

```
compound_statement
    : scope_enter statement_list scope_exit
    ;

scope_enter
    : '{'                                          { scope_enter(); }
    ;

scope_exit
    : '}'                                          { scope_exit(); }
    ;
```

Codice Sorgente 6: `parser.y` - produzioni per la gestione degli scope

Questo permette di aprire un nuovo scope ogni qual volta è riconosciuta la produzione `scope_enter`, formata dal terminale `{`, e di chiuderlo quando viene riconosciuta la produzione `scope_exit`, formata dal terminale `}`. Evidenziamo che la produzione `compound_statement` è una produzione ricorsiva: il non terminale `statement_list` contiene una lista di statement tra cui anche `compound_statement` stesso; ciò permette di avere più scope annidati.

Due casi particolari nella gestione degli scope sono rappresentati dall'istruzione `for` e dalla dichiarazione di funzioni.
Per gestire il caso in cui presente è una dichiarazione all'interno dell'`init` statement, per l'istruzione `for` viene aperto un nuovo scope subito dopo la parentesi tonda (ad esempio nel caso `for(int i = 0; ...)`).
Per le dichiarazioni di funzioni è necessario che i parametri delle funzioni siano inserti all'intero dello scope della funzione, il quale però sarà aperto dopo la parentesi graffa `{`. Per gestire correttamente tale situazione si è utilizzata una variabile globale di supporto `param_list`, cioè un puntatore alla lista dei parametri della funzione dichiarata. All'apertura di un nuovo scope se tale variabile è valorizzata si procederà ad inserire nello scope appena creato i parametri mediante la funzione `fill_symtab`.
Un procedimento analogo è effettuato per inserire il simbolo speciale `return` che indica il tipo ritornato dalla funzione. In quest'ultimo caso verrà utilizzata la variabile globale `ret_type`, per mantere il tipo di ritorno della funzione ed inserirlo nella symbol table mediante la funzione `insert_sym`.

2.2.2 Produzioni per Printf e Scanf

Si è scelto di esplicitare separatamente le produzioni per le funzioni `scanf` e `printf` dalle chiamate a funzioni definite dall'utente per molteplici motivi:

- La nostra restrizione sul linguaggio sorgente prevede che le stringhe possono essere utilizzate solo in abbinamento a queste due funzioni.
- La nostra restrizione sul linguaggio non prevede la gestione dei puntatori e dell'operatore di referenziazione (&), necessari per gli argomenti della funzione scanf.

- Essendo entrambe funzioni built-in, non se ne prevede la definizione; pertanto, non è possibile applicare gli stessi controlli utilizzati per le altre chiamate a funzioni.

2.2.3 Produzioni per gli Errori

Per evidenziare alcuni degli errori più comuni e generare messaggi di errore più specifici sono state inserite alcune produzioni estendendo la grammatica.

```
PRINTF '(' ')' ';'                { $$ = new_error(ERROR_NODE_T);
↪  yyerror("too few arguments to function" BOLD " printf" RESET); }
SCANF '(' ')' ';'                 { $$ = new_error(ERROR_NODE_T);
↪  yyerror("too few arguments to function" BOLD " scanf" RESET); }
ID declarator_list ';'        { $$ = new_error(ERROR_NODE_T);
↪  yyerror(error_string_format("Unknown type name: " BOLD "%s" RESET,
↪  $1 )); }
```

Codice Sorgente 7: `parser.y` - produzioni per gli errori

Il numero di queste produzioni è stato ridotto al minimo per evitare di complicare troppo la grammatica del parser.

2.2.4 Precedenze degli Operatori

Al fine di evitare conflitti ed ambiguità nella grammatica è stato necessario utilizzare gli operatori di precedenza. Quest'ultimi sono stati applicati alle operazioni su operatori aritmetici, di confronto e logici.

Vogliamo porre l'attenzione su due casi in particolare:

Precedenza del meno unario la precedenza del terminale - dipende dal contesto. Il simbolo - usato come operatore unario deve avere precedenza maggiore rispetto allo stesso simbolo usato come operatore binario. Per risolvere questo problema si è usato il modificatore di precedenza `%prec` in abbinamento con un simbolo terminale fittizio `MINUS` a precedenza maggiore.

```
...
%left '+' '-'
%left '*' '/'
%rigth MINUS

expr
    : ...
    | ...
    | '-' expr %prec MINUS
    ;
```

Codice Sorgente 8: `parser.y` - precedenza del terminale -

Precedenza del costrutto if-else la grammatica da noi descritta non esplicita una precedenza per i costrutti *if-else* annidati. Questa ambiguità

viene risolta automaticamente: nei conflitti shift-reduce bison preferisce sempre l'operazione di shift a quella di reduce. Quindi, la clausola `else` sarà correttamente associata al costrutto `if` più interno. Pertanto abbiamo scelto di ignorare il warning mediante la dichiarazione `%expect 1`. In modo alternativo si sarebbe potuto procedere in maniera analoga al caso precedente, come mostrato nel seguente codice:

```
%nonassoc LOWER
%nonassoc ELSE

selection_statement
        : IF '(' if_cond ')' embedded_statement  %prec LOWER
        | IF '(' if_cond ')' embedded_statement ELSE embedded_statement
        ;
```

Codice Sorgente 9: `parser.y` - precedenza del costrutto `if-else`

2.2.5 Generazione dei Nodi dell'Abstract Syntax Tree

Durante questa fase è effettuata la creazione in memoria dell'Abstract Syntax Tree. Generalmente tra le azioni associate alle produzioni vi è la creazione del relativo nodo dell'Abstract Syntax Tree ed il collegamento con i propri nodi figli.

2.3 Abstract Sintax Tree

L'Abstract Syntax Tree (Ast) è una rappresentazione dei costrutti della grammatica sotto forma di albero, più compatta del parse tree. Questa struttura dati è indispensabile per la fase di analisi semantica e per la traduzione.

2.3.1 Struttura del Nodo

Si è scelto di avere un'unica struttura per i nodi dell'Ast, chiamata `AstNode`, per semplificare la creazione e la gestione dell'Ast. La struttura di tale nodo è visibile all'interno del file `tree.h` e comprende:

AstNode		
nodetype	node	next

Tabella 2.1: Struttura dell'`AstNode`

nodetype è il tipo del nodo dal quale dipendono le strutture valorizzate all'interno della `union node`. I tipi possibili sono 10 (`EXPR_T, IF_T, VAL_T, VAR_T, DECL_T, RETURN_T, FCALL_T, FDEF_T, FOR_T, ERROR_NODE_T`)

node è un unione che contiene diverse strutture relative a differenti tipi di nodo. I puntatori ai nodi figli del nodo corrente sono contenuti all'interno di tali strutture.

next è il puntatore al nodo fratello del nodo corrente, che sarà a sua volta di tipo `AstNode` e viene valorizzato tramite la funzione `link_AstNode`.

variable		
name	by_reference	array_dim

expression		
expr_type	l	r

funcCall	
name	args

value	
val_type	string_val

funcDef			
ret_type	name	params	code

declaration	
type	var

ifNode		
cond	body	else_body

forNode			
init	cond	update	stmt

returnNode
exp

Tabella 2.2: Strutture della `node union`

2.4 Symbol Table

La Symbol Table è una struttura dati usata dal compilatore per tenere traccia della semantica degli identificatori. La struttura scelta per la Symbol Table è una hash table per la sua efficienza e gestione semplice del codice. Per la gestione della hash table si è scelto di utilizzare la libreria di supporto `uthash.h`.

2.4.1 Operazioni sulla Symbol Table

In `symtab.c` troviamo le funzioni relative alle operazioni possibili sulla Symbol Table. Una funzione di particolare interesse è `check_usage`. Questa funzione è richiamata ogni qual volta si chiude uno scope e serve per controllare che tutte le variabili all'interno dello scope siano state utilizzate. Questo controllo è necessario in quanto Golang non permette di dichiarare variabili che restano inutilizzate.

2.4.2 Struttura dei Record

Ogni Symbol Table contiene un record per ogni identificatore, con dei campi per gli attributi. In particolare i record vengono creati durante la fase di parsing e non durante il riconoscimento dei lessemi da parte dello scanner. La struttura dei record è presente all'interno del file `symtab.h`, ed è rappresentata dalla `struct symbol`, i campi in essa contenuti sono i seguenti:

name	type	sym_type	pl	used_flag	array_flag	lineno	line

Tabella 2.3: Struttura dei Record della Symbol Table

name rappresenta il nome del simbolo, è utilizzato come chiave per la hash table.

type rappresenta per variabili e parametri il tipo mentre per le funzioni rappresenta il tipo di ritorno, così come per il simbolo speciale `return`.

sym_type è il tipo di simbolo. I tipi possibili sono `VARIABLE`, `FUNCTION`, `PARAMETER`, `F_RETURN`, utilizzati rispettivamente per variabili, funzioni, parametri e il simbolo speciale `return`. Questo simbolo viene inserto all'interno della tabella relativa allo scope di una funzione. È utilizzato, quando viene chiamata l'istruzione `return`, per controllare la coerenza del tipo ritornato da una funzione con la sua dichiarazione.

pl è un puntatore alla lista dei parametri della funzione.

used_flag è un flag che indica se una variabile viene utilizzata o meno all'interno di uno scope.

array_flag è un flag che indica se una variabile o un parametro è un array o meno.

lineno è utilizzato per salvare il numero di riga in cui il simbolo viene dichiarato, in modo da poter presentare errori più esaustivi.

line è utilizzato per salvare una copia della riga in cui il simbolo viene dichiarato, in modo da poter presentare errori più esaustivi.

2.4.3 Gestione degli Scope

Per quanto riguarda la gestione degli scope si è scelto di utilizzare una tabella separata per ogni scope del programma. La gerarchia di tali tabelle è gestista mediante uno stack, implementato come una lista linkata. La struttura della lista è presente nel file `symtab.h` e consiste di una semplice `struct symlist`, formata da:

scope	symtab	next

Tabella 2.4: Struttura `symlist`

scope un intero che rappresenta il numero dello scope.

symtab un puntatore alla tabella corrente.

next un puntatore alla tabella successiva nello stack, ovvero allo scope più esterno.

Le operazioni necessarie per la gestione degli scope comprendono:

- All'apertura di un nuovo scope: è incrementato il numero dello scope attuale (`current_scope_lvl`), è aggiunta una nuova hash table vuota come top dello stack ed è aggiornato il puntatore al top dello stack (`current_symtab`).
- Alla dichiarazione di un identificatore: viene controllato se il simbolo è già presente nello scope corrente. Se è presente, vuol dire che l'identificatore è già stato dichiarato pertanto verrà generato un errore. In caso contrario, il simbolo viene inserito nella Symbol Table corrente.
- All'uso di un identificatore: viene controllato se il simbolo è presente nello scope corrente. Se non è presente la ricerca continua con le symbol table associate agli scope via via più esterni. Se il simbolo non è presente in nessuno degli scope aperti verrà generato un errore.
- Alla chiusura di uno scope: è decrementato il numero dello scope attuale, è eliminata la tabella corrente e il puntatore `current_symtab` verrà valorizzata all'indirizzo della Symbol Table dello scope precedente.

2.5 Analisi Semantica

Lo scopo della fase di analisi semantica consiste nell'effettuare controlli sulla semantica delle istruzioni ovvero controllare che non siano presenti errori di significato.

Nel nostro transpiler i controlli semantici vengono effettuati quando viene derivata una data produzione e dopo la generazione del nodo corrispondente dell'Ast; ciò permette di sfruttare le informazioni che risiedono all'interno dei nodi dell'Ast.

È possibile visionare tutti i controlli semantici previsti nei file `semantic.c` e `semantic.h`.

2.5.1 Controlli sulle Espressioni

I controlli effettuati sulle espressioni verificano che non siano presenti divisioni per zero e che i tipi degli operandi corrispondano a quelli accettati dagli operatori.

Viene verificato inoltre che un'espressione venga assegnata ad una variabile o sia parte di una chiamata a funzione, se questo non si verifica il nostro transpiler assegnerà l'espressione al blank identifier `_`. Abbiamo implementato questa soluzione in quanto in Golang, diversamente dal C, non è prevista la possibilità di avere statement composti da sole espressioni.

Infine, le espressioni usate come condizioni nei costrutti `for` ed `if` vengono valutate e devono risultare di tipo booleano, come imposto dal linguaggio Golang.

2.5.2 Controlli sugli Array

Per quanto concerne gli array sono stati effettuati controlli sulla loro dimensione e indice. In particolare rappresentano errori semantici:

- array con dimensione e/o indice negativo.
- array con dimensione e/o indice non intero. Nello specifico non è possibile utilizzare espressioni contenenti operatori logici o di confronto, in quanto in Golang il risultato di tali espressioni è di tipo booleano.
- l'utilizzo di un array senza indicizzazione. Quest'utilizzo è permesso solo nel caso in cui l'array venga passato come parametro di una funzione.

2.5.3 Controlli sulle Assegnazioni

Per le operazioni di assegnazione vengono effettuati due tipi di controlli semantici:

- durante l'assegnazione di un'espressione a una variabile di tipo intero si controlla che il risultato dell'espressione, nel caso sia costante, consista in un numero intero o numero con valore decimale nullo. In caso contrario viene generato un errore di troncamento della costante.

- durante l'assegnazione di un'espressione a una variabile, il tipo dell'espressione deve essere concorde con quello della variabile. Le variabili che compongono l'espressione devono avere tutte lo stesso tipo della variabile a cui viene assegnata l'espressione stessa. Le costanti, invece, devono poter essere rappresentabili nel tipo della variabile (ad esempio una variabile di tipo float può contenere valori interi e non viceversa).

2.5.4 Controlli sulle Dichiarazioni di Variabili

Quando viene dichiarata una variabile viene controllato che non sia stata già dichiarata una variabile con lo stesso identificatore all'interno dello stesso scope. Per quanto riguarda le dichiarazioni con assegnazione vengono applicati anche i controlli sulle assegnazioni.

2.5.5 Controlli sulle Dichiarazioni di Funzioni

Quando viene dichiarata una funzione viene controllato che non sia già presente una dichiarazione di funzione con lo stesso identificatore. È verificato, inoltre, che il tipo di ritorno della funzione sia concorde con il tipo degli statement `return` presenti nella stessa. È prevista la possibilità di non avere uno statement `return` solo per le funzioni di tipo void.

Un caso particolare è la dichiarazione della funzione main; essa deve essere definita di tipo void e senza alcun argomento.

2.5.6 Controlli sulle Chiamate a Funzioni

La semantica delle chiamate a funzioni prevede che: esse siano state precedentemente definite, il numero degli argomenti passati alla funzione coincida con quello dei parametri attesi dalla funzione e che ogni argomento sia assegnabile al parametro relativo.

2.5.7 Controlli sulle Funzioni Printf e Scanf

Per le funzione `scanf` e `printf` viene controllato che il numero e il tipo degli argomenti sia concorde con quello previsto dalla *format string*.

La nostra restrizione del linguaggio non prevede l'uso della direttiva `include`; pertanto, si è scelto di considerare entrambe le funzioni come appartenenti alla libreria interna del compilatore.

2.6 Generazione del Codice

Per la generazione del codice è stato utilizzato un approccio classico, composto da due fasi:

1. Durante la fase di parsing viene creato l'Abstract Syntax Tree.
2. Uso dell'Abstract Syntax Tree per la generazione del codice target.

Nel nostro caso come rappresentazione intermedia è utilizzato l'Abstract Syntax Tree, in quanto risulta essere la scelta migliore per un compilatore source-to-source.

La traduzione viene effettuata solo se durante le fasi precedenti non si sono rilevati errori di alcun tipo. É possibile visionare le funzioni utilizzate nella traduzione nei file `translate.h` e `translate.c`.

La struttura del linguaggio di destinazione prevede che il primo statement sia una clausola `package` che definisce a quale package il file appartiene. Nello specifico se presente la funzione main il primo statement deve essere necessariamente `package main`. Per tenere traccia della presenza della funzione main, nel file sorgente, si è utilizzata la variabile `main_flag`. Durante la traduzione, se tale variabile è valorizzata, il primo statement del codice generato sarà `package main` altrimenti si utilizzerà come nome del package `test`.

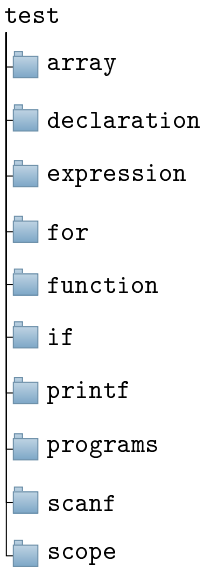
Nel linguaggio di destinazione è possibile importare le librerie solo se le funzioni ad esse appartenenti sono effettivamente impiegate nel codice. Per tenere traccia dell'eventuale presenza delle funzioni `scanf` e `printf` si è utilizzata la variabile `fmt_flag`. Nel caso in cui tale variabile è valorizzata sarà inserito l'`import` della libreria `fmt` che permette di utilizzare le funzioni equivalenti `fmt.Printf` e `fmt.Scanf`.

Il resto della traduzione è effettuato attraversando i nodi dell'Abstract Syntax Tree mediante strategia pre-order. Partendo dal nodo radice viene attraversato prima l'intero sottoalbero sinistro e infine il sottoalbero destro.

Capitolo 3

Test

I file di test sono organizzati nella cartella `test` presente all'interno della directory principale del progetto. I test sono organizzati gerarchicamente mediante una suddivisione in sottocartele che individuano l'oggetto del test. A sua volta, in ogni sottocartella, distinguiamo i test validi, dagli errori e da eventuali test che presentano warning.



3.1 Gestione degli Errori

Esistono quattro principali tipologie di errori individuabili all'interno di un codice: lessicali, sintattici, semantici e logici. Comunemente gli errori individuabili all'interno di un programma possono essere di quattro tipi:

lessicali quando lo scanner individua un lessema non associabile a nessuna parola ammessa dal linguaggio.

sintattici quando la sequenza di parole non identifica una frase ammessa dalla sintassi del linguaggio.

semantici ovvero errori di significato.

logici derivano da un'errata logica di stesura del programma da parte del programmatore.

In generale sarebbe opportuno trattare in modo differente le diverse tipologie di errore. Nel nostro caso abbiamo deciso di emulare gli errori presentati dal compilatore `gcc` e dal compilatore del linguaggio di programmazione Golang. Abbiamo distinto due tipi di errori: errori bloccanti, che non permettono la generazione della traduzione e errori non bloccanti, rappresentati dai warning, che permettono comunque di generare la traduzione del programma sorgente.

I metodi utilizzate nella gestione degli errori sono riassunti nella seguente tabella.

Tipologia di Errore	**Metodo di Gestione**	**Dettagli**
Lessicali	sempre bloccanti	è prevista una espressione regolare di fallback, che rileva le parole non ammesse dal linguaggio
Sintattici	sempre bloccanti	sono previste delle error production, per rilevare gli errori più comuni
Semantici	bloccanti e non bloccanti	in base alla semantica sono distinti gli errori dai warning
Logici	non rilevati	sono rilevati a run-time

Tabella 3.1: Metodi di Gestione degli Errori

Bibliografia

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition).* Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[2] John Levine and Levine John. *Flex & Bison.* O'Reilly Media, Inc., 1st edition, 2009.

[3] D. Thain. *Introduction to Compilers and Language Design.* LULU Press, 2019.

[4] Floriano Scioscia. Formal languages and compilers - slide del corso, 2019/2020. URL: `http://sisinflab.poliba.it/scioscia`.

[5] Gnu bison - the yacc-compatible parser generator. URL: `https://www.gnu.org/software/bison/manual`.

[6] The go programming language specification. URL: `https://golang.org/ref/spec`.