Università degli Studi di Padova

Dipartimento di Ingegneria Informatica

Compilatori:

Relazione di progetto

A.A. 2019/20

Davide Serratore - M.1207660

Omar Vesco - M.1197699

Introduzione

Il progetto consiste nell'analisi e progettazione di un piccolo compilatore in grado di elaborare alcune operazioni che si ispirano al linguaggio standard C. L'output ottenuto è ancora un linguaggio basato su C in modo da permettere un semplice testing del codice che verrà generato.
L'unico tipo ammesso per le variabili è quello intero ed è possibile eseguire operazioni aritmetiche, logiche e di confronto.
Di particolare interesse sono la gestione di diverse Symbol Tables e la traduzione di costrutti come il ciclo while e le istruzioni condizionali If e If-else.
L'obiettivo principale è quello di generare il codice intermedio eseguendo una sola passata sul nostro input.

Linguaggio di Programmazione

Il linguaggio permette le seguenti operazioni:

- Dichiarazione di una variabile: "**int a;**". La variabile viene inizializzata al valore 0
- Assegnazione di un numero ad una variabile già definita: "**a = 10;**". Si può inoltre assegnare un'espressione booleana o aritmetica: nei primi due casi il valore assegnato sarà 0 se falsa, 1 se vera. Le espressioni booleane possono contenere delle espressioni aritmetiche e di confronto. È possibile inoltre eseguire un'assegnazione all'interno di un'espressione (in questo caso l'end marker "**;**" è omesso)
- Dichiarazione di una variabile con un valore specifico: "**int a = 10;**"
- Stampa di una variabile: "**print(a);**"
- Condizioni if e if-else tipiche del C
- Ciclo while tipico del C

Ogni istruzione termina con il carattere "**;**"

Descrizione del Progetto

Il progetto si divide in tre principali parti:

1. Analizzatore Lessicale
2. Analizzatore Sintattico
3. Funzioni Utili

1. Analizzatore Lessicale

Nella prima parte del progetto l'obiettivo è di definire i token necessari per la traduzione del codice in input. La descrizione dei token finali utilizzati sono mostrati in tabella.

Token	Name	Attribute
int	INT	-
if	IF	0 (intero)
else	ELSE	-
while	WHILE	1 (intero)
Operatori di confronto: <, <=, ==, !=, >, >=	RELOP	'<', '<=', '==', '!=', '>', '>='
Operatori logici: &&, ||, !,	AND, OR, NOT	-
print	PRINT	-
id	ID	Stringa
number	NUMBER	Valore intero
Operatori aritmetici	'+', '-', '*', '/'	-
Parentesi graffe	'{' , '}'	-
Parentesi tonde	'(' , ')'	-

Tabella 1: Token utilizzati

Le principali modifiche effettuate durante lo sviluppo del compilatore sono:

- Gli operatori logici inizialmente erano raggruppati all'interno di un unico token **<LOGICAL_OP, Attribute>**, dove l'attributo rappresentava l'operatore. In fase di sviluppo si è preferito dividerli in differenti token per poter gestire in modo più naturale le precedenze tra i vari operatori.
- I token **IF** e **WHILE** inizialmente non prevedevano attributi. La scelta di ritornare un intero è stata fatta per semplificare leggermente la nostra grammatica e permettere il facile riconoscimento della tipologia di costrutto condizionale in esame.

2. Analizzatore Sintattico

Nell'effettuare l'analisi sintattica, abbiamo definito una CFG con relative azioni semantiche per la traduzione dell'input. Nelle prime fasi di sviluppo sono stati seguiti i codici di esempio mostrati a lezione e successivamente si è iniziato ad implementare scelte progettuali autonome.

La grammatica è risultata ambigua ed è stato necessario risolvere i conflitti introducendo dei token simbolici e quindi delle precedenze. Il primo conflitto shift/reduce è emerso nella definizione della produzione relativa all'assegnazione di variabili all'interno di un'espressione, **bexpr -> ID ASSIGN bexpr**. Per risolverlo è stato necessario introdurre il token simbolico **ASSIGN_PRECEDENCE** con precedenza inferiore al token **ENDMARKER**. In questo modo, nel caso in il lookahead fosse rappresentato da un "**;**", viene eseguito uno shift in modo da inserire **ENDMARKER** nello stack ed effettuare quindi una successiva riduzione a **stmt**. In caso contrario viene effettuata una riduzione diretta a **bexpr**, scindendo così i due tipi di assegnazione in base al contesto.

Un altro non terminale critico è **expr**. Per permettere l'inserimento di espressioni aritmetiche all'interno di espressioni booleane, è stata necessaria l'introduzione di una nuova produzione **bexpr** -> **expr**; questo ha inevitabilmente fatto emergere dei conflitti con tutte le altre produzioni relative alle operazioni aritmetiche e di confronto**.** Per risolvere il problema è stato introdotto il token simbolico **EXPR_PRECEDENCE** con precedenza inferiore ai token aritmetici e di confronto. In questo modo, quando **expr** si presenta al top dello stack, non viene eseguita subito una riduzione a **bexpr** ma, in caso di lookahead con i vari operatori aritmetici, uno shift.

Per la traduzione del ciclo while e dei costrutti condizionali **if** ed **if-else** sono stati utilizzati solo attributi ereditati.
Questi ultimi hanno bisogno di conoscere il label dello statement successivo in modo tale da poter saltare alla prossima operazione quando richiesto. Questa informazione viene propagata dal non terminale **liststmt** al padre, se non viene utilizzata, altrimenti ne genera una nuova. Per creare la prima label del programma o di un blocco viene usato il non terminale **nextlabel**. Se un **stmt** ha bisogno di recuperare il label può accedere allo stack nella posizione precedente alla riduzione attuale.
L'aggiornamento del label è possibile grazie ad un valore 0 o 1 (false o true) ritornato da **stmt** che dichiara se è un costrutto condizionale o meno.
Vista la richiesta di generare il codice intermedio con una sola passata sull'input, è stato richiesto l'utilizzo di non terminali (**print_if**, **print_while**, **print_else**) funzionali solo alla stampa dei comandi nella posizione corretta.
La struttura della grammatica raccoglie all'interno di **stmt** sia operazioni standard che costrutti condizionali: questo permette di poter riconoscere condizioni innestate ma per fare ciò è stato necessario trasferire il label successivo ai non terminali per la stampa. Questo perchè i diversi **stmt** raccolgono il label nello stack in un indice preciso.
La struttura **IF (bexpr) stmt ELSE stmt** ha creato qualche problema nella grammatica per via dei conflitti con la condizione **IF (bexpr) stmt**. I due costrutti richiedono label differenti ma parte del codice viene generato prima di verificare se il token **ELSE** è presente o meno. Il problema è stato risolto uniformando le due produzioni: questo porta, in assenza di **ELSE**, la stampa di un label vuoto ma la corretta esecuzione del codice.
Le stringhe che rappresentano le istruzioni vengono raccolte all'interno di una lista per effettuare una stampa finale, dopo aver letto l'intero file di input. Questo ha permesso di stampare le dichiarazioni delle variabili temporanee prima dell'intero codice, salvate in modo opportuno all'interno di una seconda lista.

3. Funzioni Utili

Le Symbol Table sono gestite attraverso una lista **LIFO**. Ognuna di queste non ammette variabili con lo stesso nome ed è rappresentata da una hash table. La lista viene generata anche durante la fase di compilazione permettendo al compilatore di verificare se una variabile è già stata dichiarata: in caso affermativo lancia un errore.
Per la sua implementazione abbiamo utilizzato la libreria esterna **uthash** (uthash: a hash table for C structures).
Le funzioni principali sono:

- **init_table()** : inizializza la prima Table
- **create_new_table()** : crea una nuova Table vuota e la inserisce all'interno della lista
- **delete_current_table()** : cancella e libera la memoria dell'ultima Table inserita nella lista
- **addVar(char *id)** : inserisce una nuova variabile nell'ultima Table (se non già presente) e le assegna il valore 0
- **setVar(char *id, int value)** : assegna un valore alla variabile **id**. La variabile viene cercata all'interno dell'intera lista a partire dall'ultima Table generata
- **getVar(char *id)** : ritorna il valore della variabile **id**. La variabile viene cercata all'interno dell'intera lista a partire dall'ultima Table generata

Altre funzioni utili:

- **gen_tmp(char *tmp_string)** : genera la stringa per una nuova variabile temporanea.
- **gen_label(char *label)** : genera la stringa per una nuova label
- **insert_string(char *variable_name, node_string **endlist)** : inserisce una stringa all'interno di una lista
- **print_all(node_string *mystrings, node_string *tmp_string)** : stampa in ordine la lista delle variabili temporanee e le stringhe di codice
- **free_all(node_string **mystrings)** : libera la memoria delle liste contenenti stringhe