Università degli Studi di Salerno
Dipartimento di Informatica

Esame di Compilatori
Anno Accademico 2020/2021

Progetto di Compilatori

Studenti:
Emmanuel Tesauro
Luigi Cerrone

Docente:
Gennaro Costagliola

Indice

Analisi lessicale & sintattica

La quinta ed ultima esercitazione ha avuto inizio con la creazione di un analizzatore lessicale in grado di trasformare il codice sorgente, scritto in linguaggio Toy, in un flusso di Token.
L'analizzatore lessicale è stato creato utilizzando il tool JFLEX, al quale sono state fornite le specifiche lessicali composte da espressioni regolari e stati (YYINITIAL, STRING, COMMENT) formulate basandosi sulla traccia.

Una volta terminata la creazione del Lexer, si è passati alla generazione di un analizzatore sintattico, il quale ha avuto il compito di analizzare e validare il flusso di token restituito dall'analizzatore lessicale e generare un albero sintattico corrispondente al codice sorgente scritto in Toy.
Il Parser è stato generato attraverso l'utilizzo del tool Java CUP, al quale è stato fornito in input una grammatica S-Attribuita specificata nel file toy.cup.
Dopo aver testato che l'analizzatore sintattico funzionasse a dovere, è stato necessario generare un albero XML corrispondente al codice del programma in input attraverso la creazione di un primo Visitor.

Analisi semantica

In questa sezione verranno elencate tutte le regole di inferenza che sono state utilizzate durante l'analisi semantica. Infine, verranno evidenziate alcune scelte che sono state prese durante lo sviluppo e che sono state implementate durante l'analisi semantica.

Tipi primitivi

$\Gamma \vdash null : Null$
$\Gamma \vdash int : Integer$
$\Gamma \vdash float : Float$
$\Gamma \vdash string : String$
$\Gamma \vdash true : Boolean$
$\Gamma \vdash false : Boolean$

Controllo di tipo per l'ID

$$\frac{(x: \tau) \in \Gamma}{\Gamma \vdash x: \tau}$$

Controllo di tipo per l'inizializzazione per l'ID

$$\frac{\Gamma \vdash x: \tau \quad expr: \tau' \quad getType\_Operation(\tau, \tau'): \tau}{\Gamma \vdash (x := expr): \tau}$$

Controllo di tipo per l'operazione unaria

$$\frac{\Gamma \vdash arg: \tau \quad getType\_Single(op, \tau): \tau}{\Gamma \vdash (op\ arg): \tau}$$

Controllo di tipo per le operazioni binarie

$$\frac{\Gamma \vdash arg_1: \tau_1 \quad \Gamma \vdash arg_2: \tau_2 \quad getType\_Operation(op, \tau_1, \tau_2): \tau}{\Gamma \vdash (arg_1\ op\ arg_2): \tau}$$

Il metodo *getType_Operation(τ , τ΄)* prende in input i tipi dei due argomenti presi in considerazione e restituisce il tipo dell'operazione (in accordo con

quanto dichiarato nelle prime 4 righe della tabella 1) oppure 'null' se gli argomenti non fossero stati compatibili tra di loro.
I controlli per le operazioni di confronto (<, >, ≤, …), AND e OR sono riportati anch'essi nel metodo *getType_Operation* ma durante l'implementazione si è preferito separare questa parte dalla precedente per avere una maggiore leggibilità.
I metodi in questione si chiamano *getType_AndOr e getType_Boolean*.
Il metodo *getType_Single(op, τ)* è relativo alle operazioni di '!' (Not) e '-' (uminus) e restituisce un valore booleano che sta ad indicare se l'operazione può essere eseguita oppure no. Queste ultime specifiche sono riassunte all'interno della tabella 2.
Nella pratica, questo metodo non è stato implementato poiché è stato sufficiente effettuare un solo controllo e non si è ritenuto necessario creare un metodo dedicato ad esso.

Tabella 1 – getType_Operation (op, τ_1, τ_2)			
op	τ_1	τ_2	**result**
+ - * /	Integer	Integer	Integer
+ - * /	Integer	Float	Float
+ - * /	Float	Integer	Float
+ - * /	Float	Float	Float
= < > <= >= <>	Boolean	Boolean	Boolean
= < > <= >= <>	Integer	Integer	Boolean
= < > <= >= <>	Integer	Float	Boolean
= < > <= >= <>	Float	Integer	Boolean
= < > <= >= <>	Float	Float	Boolean
= < > <= >= <>	String	String	Boolean
AND OR	Boolean	Boolean	Boolean

Tabella 2 – getType_Single (op, τ_1)		
op	τ_1	**result**
uminus	Integer	Integer
uminus	Float	Float
!	Boolean	Boolean

Controlli per gli statement

Alcuni campi dei seguenti statement presentano uno '*', il quale sta ad indicare che il campo può essere nullo e non comparire nello statement senza andare ad intaccare la corretta sintassi e semantica del programma.
Di seguito, i campi opzionali verranno indicati con [...], mentre le liste verranno indicate con {...}.

If statement

$$\frac{\Gamma \vdash cond: Boolean \quad \Gamma \vdash ifBodyStat \quad \Gamma \vdash elifList^* \quad \Gamma \vdash elseBody^*}{\Gamma \vdash if\ cond\ then\ ifBodyStat\ [elifList]\ [elseBody]\ fi}$$

Elif statement

$$\frac{\Gamma \vdash cond: Boolean \quad \Gamma \vdash elifBody}{\Gamma \vdash elif\ cond\ then\ elifBody}$$

While statement

$$\frac{\Gamma \vdash \text{preCondStat}^* \quad \Gamma \vdash \text{cond: Boolean} \quad \Gamma \vdash \text{whileBodyStat}}{\Gamma \vdash \text{while [preCondStat]} \rightarrow \text{cond do whileBodyStat od}}$$

ReadIn statement

$$\frac{\Gamma \vdash idList: \{\tau_1, \dots, \tau_n\}}{\Gamma \vdash readln(idList)}$$

Write statement

$$\frac{\Gamma \vdash exprList: \{\tau_1, \dots, \tau_n\}}{\Gamma \vdash write(exprList)}$$

Assign statement

$$\frac{\Gamma \vdash idList: \{\tau_1, \dots, \tau_n\} \quad \Gamma \vdash exprList: \{\tau_1, \dots, \tau_n\}}{\Gamma \vdash idList := exprList}$$

Call procedure statement

$$\frac{\Gamma \vdash id: \{\tau_1, \dots, \tau_n\} \quad \Gamma \vdash exprList: \{\tau_1, \dots, \tau_n\}^*}{\Gamma \vdash id([exprList])}$$

Le liste per espressioni, inizializzazioni, restituzioni, variabili e procedure sono date vere se e solo se i controlli di tipo per gli elementi interni risultano essere corretti.

Inoltre, per quanto riguarda l'analisi semantica, qui di seguito verranno riportate alcune scelte implementative al fine di validare o no l'intero sorgente toy.

- La proc 'main' non deve essere obbligatoriamente presente all'interno del sorgente Toy in quanto si è pensato che nel linguaggio C questa caratteristica è ammessa;
- Il tipo di ritorno della proc 'main' è stato forzato ad essere solamente di tipo Int. I controlli che sono stati fatti riguardano gli altri tipi di ritorno, cioè Void, Float, String, Boolean e Null, dato che questi ultimi non sono ammessi in C;
- I parametri in input alla proc 'main' sono ammessi e la conversione in codice C verrà discussa nella sezione successiva;

Generazione codice intermedio

In questa sezione verranno evidenziate tutte le scelte che sono state prese durante la generazione del codice intermedio ed implementate attraverso il CVisitor.

- Sono state introdotte le librerie stdio.h, stdlib.h, stdbool.h e string.h al fine di effettuare le operazioni standard di input/output, effettuare operazioni sulle stringhe (es. strcmp, strcpy) e poter gestire i tipi booleani in quanto il linguaggio C non lo permette nativamente.

- Per quanto riguarda il passaggio di parametri al main, questo è stato implementato facendo restituire al metodo 'visit' del nodo 'ProgramNode' un valore intero calcolato quando si incontra la proc 'main' e analizzando i parametri nell'array di ParDecl. Se questo valore fosse stato zero allora il main avrebbe avuto tipo void, altrimenti sarebbe stato necessario scrivere, come vuole la sintassi del C, i parametri 'int argc, char *argv[]'. A questo punto, nel corpo del main, tutti i valori passati in input in Toy sarebbero stati assegnati alle variabili (inizializzate al momento) con le opportune conversioni (atoi, atof, ecc…). Durante l'esecuzione sarà la shell ad avvisare che il programma in questione desidera 'n' parametri in input, li farà inserire tramite un normale '*Scanner in=new Scanner(System.in)*' e li passerà allo script o al comando per la shell Windows. Questi valori verranno utilizzati nel seguente modo: '*./main par1 par2 …*'.

```
- Linguaggio Toy                                  - Linguaggio C
proc main(int x; string y; float j)int:  ->  int main(int argc, char* argv[]) {
                                                  int x = atoi(argv[1]);
                                                  char y[512] = argv[2];
                                                  float j = atof(argv[3]);
```

Trasformazione parametri dal linguaggio Toy al C

Dal momento in cui il linguaggio Toy permette di restituire valori multipli da una proc, sono state necessarie alcune operazioni per ottenere lo stesso risultato in C. La scelta che è stata fatta è stata quella di utilizzare le strutture del linguaggio C. Più nel dettaglio, quando una proc restituisce valori multipli viene creata una struct contenente tutti i valori restituiti da quella specifica proc. In questo modo è stato possibile creare una variabile di tipo struct poco prima del return nel corpo della proc a cui vengono assegnati i valori da restituire e restituirla alla funzione chiamante.

```
// Proc multAddDiff
struct struct_multAddDiff {
  int variable0;
  int variable1;
  int variable2;
  char variable3[512];
};
```

```
struct struct_multAddDiff multAddDiff() {
  ...
  int mul, add, dif;
  ...
  // Struttura di ritorno
  struct struct_multAddDiff return_multAddDiff;
  return_multAddDiff.variable0 = mul;
  return_multAddDiff.variable1 = add;
  return_multAddDiff.variable2 = diff;
  strcpy(return_multAddDiff.variable3, nome);
  return return_multAddDiff;
}
```

Dichiarazione della struttura per multAddDiff ed esempio di uso nel corpo della funzione

- Per tutte le altre funzioni, ovvero quelle che restituiscono un solo valore oppure hanno come tipo di ritorno void, non è stato necessario generare una struttura ed è stato sufficiente inserire prima del nome della funzione il tipo di ritorno oppure la parola chiave 'void'.

- Per quanto riguarda la chiamata a procedura con multipli valori di ritorno, in questo caso viene creata una variabile che ha come tipo esattamente la struttura creata precedentemente. Per convenzione, si è deciso di chiamare la variabile nel seguente modo: '*struct struct_nomeFunzione nomeFunzione_returnIndice*' dove 'Indice' è un contatore globale che indica quante chiamate a procedure sono state fatte. Utilizzando questo contatore non si hanno problemi di ridichiarazione di variabili. A questo punto verranno presi i valori contenuti all'interno di questa struttura e verranno assegnati correttamente alle variabili decise in Toy.

```
// Struttura con i valori di ritorno di multAddDiff
struct struct_multAddDiff multAddDiff_return0 = multAddDiff();
a = multAddDiff_return0.variable0;
b = multAddDiff_return0.variable1;
c = multAddDiff_return0.variable2;
strcpy(d, multAddDiff_return0.variable3);
```

Utilizzo della struttura per l'assegnazione di valori multipli di ritorno della proc multAddDiff

- Se in una 'write' viene passata una procedura, allora all'interno della printf generata nel codice C viene settato il formato corretto per visualizzare il valore restituito, ad esempio un '%d' per gli interi e booleani. Se la procedura avesse restituito più di un valore, secondo il metodo descritto nel punto precedentemente, viene creata una struttura e vengono generate tante printf quanti sono i valori restituiti.

- Il tipo string del linguaggio Toy è stato trasformato in un array di char nel linguaggio C, fissando la dimensione di 512 all'array. Inoltre, per tutte le

funzioni di assegnazione riguardanti le stringhe, è stata utilizzata la funzione 'strcpy' presente nella libreria string.h.

- Tutte le altre operazioni sulle stringhe sono state implementate utilizzando la funzione 'strcmp' presente all'interno della libreria string.h. In questo modo è stato possibile fare controlli per stabilire l'ordine lessicografico delle stringhe. Ad esempio l'istruzione '*if "Marco" <= "Francesco" '* del linguaggio Toy verrà tradotta in '*if(strcmp("Marco", "Francesco) <= 0)*' nel linguaggio C.

- Una volta generato il codice in linguaggio C, è stato utilizzando il comando '*clang-format --style=google -i nomeFile.c*' per indentare correttamente il codice. Per quanto riguarda Mac, questo comando viene eseguito richiamando lo script '*format_MAC.sh*', mentre su Windows si è scelto di eseguirlo direttamente nella classe Tester. Inoltre, è previsto una scelta per l'utente per compilare, formattare ed eseguire il codice C automaticamente. Per Mac, lo script è '*format&run_MAC.sh*' mentre su Windows, vale quanto detto precedentemente.

Emscripten

In quest'ultima sezione verranno elencati brevemente i passaggi che permettono l'esecuzione del codice C all'interno di un browser attraverso Emscripten.
Innanzitutto, dopo aver scaricato Emscripten dal sito ufficiale, si è proceduto con l'esecuzione di svariati test che hanno portato a dedurre che questo tool presenta un bug non trascurabile. Infatti, quando all'interno del codice viene eseguita una *scanf(), l'intera esecuzione* va in loop compromettendo la logica del programma. Nel caso in cui ci fossero molteplici scanf(), i valori vengono richiesti in input all'inizio dell'esecuzione, prima di qualsiasi altra cosa. Per questo motivo si è scelto di creare un programma scritto in Toy e tradotto in C ad hoc, senza l'utilizzo di questa funzione.
Per quanto riguarda il progetto, sono stati creati due script bash, di cui uno per Windows ed uno per MacOS/Linux, al fine di eseguire correttamente la compilazione del file sorgente .c e avviare Emscripten per l'esecuzione su browser (o, volendo, con NodeJS).
Di seguito vengono riportati gli script bash per MacOS e Windows con dei commenti per descrivere ogni passo degli script stessi.

```
# - activate: portare il terminal in foreground
# - source path/to/emsdk_env.sh: settare l'ambiente (altrimenti emcc e emrun non
funzionerebbero)
# - $1: nome del file in input
# - emcc source.c -o output.html: creare l'output .html
# - emrun file.html: startare il server e far partire l'esecuzione
osascript -e "
    tell application \"Terminal\"
        activate
        do script \"source /Users/manu/Desktop/emsdk/emsdk_env.sh\" in window 1
        do script \"cd /Users/manu/IdeaProjects/tesauro-cerrone_es5_scg/\" in window 1
        do script \"emcc $1 -o test_files/html_outputs/out.html\" in window 1
        do script \"emrun test_files/html_outputs/out.html\" in window 1
    end tell"
```

```
::Comando che esegue in background il file dato come argomento
call "emsdk_env.bat"
::Comando che permette di spostarsi in una specifica current directory
cd "C:\Users\cerro\IdeaProjects\tesauro-cerrone_es5_scg"
::Comando che esegue in background il comando emcc che dato un file C dato in input
::viene convertito in un file HTML "out.html"
call emcc %1 -o test_files\html_outputs_windows\out.html
::Comando che esegue in background il comando emrun che fa partire il server
::e rimanda l'utente sul browser eseguendo il file HTML "out.html"
call emrun test_files\html_outputs_windows\out.html
PAUSE
```