Corso di Compilatori e Interpreti

Relazione del progetto d'esame

Compilatore SimpLanPlus

Autori:

Balugani Lorenzo	(lorenzo.balugani2@studio.unibo.it)	- Matr 0001039239
Cosenza Alessandra	(alessandra.cosenza@studio.unibo.it)	- Matr 0001052401
Mazzocato Luca	(luca.mazzocato@studio.unibo.it)	- Matr 0001046637

Anno Accademico 2021-2022

0 Indice

1 Introduzione a SimpLanPlus

SLP (SimpLanPlus) è un linguaggio imperativo che consente ai parametri delle funzioni di essere passati sia per variabile che per valore (indicato dal token "var"), e che non ammette definizioni annidate di funzioni e mutua ricorsione, pur consentendo la ricorsione tradizionale. Le variabili possono essere o di tipo intero oppure booleano, e debbono essere definite all'inizio di un blocco di codice. Non è – inoltre – possibile accedere all'interno di una funzione a variabili globali, ma è possibile per una funzione chiamarne un'altra – a patto che questa sia stata definita precedentemente.

Oltre al codice sorgente del compilatore/interprete, codici d'esempio e alla qui presente relazione, all'interno dell'archivio consegnato è presente un file .jar, che consente di provare rapidamente il progetto senza bisogno di setup particolari. È sufficiente aprire un terminale e inserire il comando "java -jar slp.jar [filename]", inserendo al posto della sezione tra le graffe il nome del file da compilare ed eseguire. Sono inoltre presenti comandi e funzionalità extra, la cui descrizione e utilizzo è contenuta nel capitolo 6 di questa relazione.

1.1 Predisposizione dell'ambiente di sviluppo

Il team ha deciso di impiegare come IDE IntelliJ IDEA Ultimate e di utilizzare GitHub per il versioning, al fine di coordinare al meglio l'attività lavorativa. Installata l'estensione di ANTLR per l'ambiente di sviluppo, il gruppo ha studiato la sintassi di SLP sperimentando con diversi programmi e osservando i vari alberi sintattici risultanti.

Su modello del progetto fornito da esempio, il package del compilatore è stato così suddiviso:

- **Ast**, contiene le implementazioni dei nodi dell'albero sintattico, del visitor e della symbol table;
- **Compiler**, contiene il runner del progetto (la classe Main) e l'handler degli errori lessico/sintattici;
- **Interpreter**, contiene la classe dell'interprete, il parser per il file assembly e le definizioni dei nodi che compongono l'AST del linguaggio intermedio;
- **Parser**, contiene il file della grammatica e le varie classi generate da ANTLR sulla base di essa;
- **Utils**, contiene le classi per la gestione della Symbol Table, gli effetti, le label e la gestione degli errori.

Al fine di rispettare i requisiti del linguaggio, la grammatica è stata modificata introducendo un elemento "program", che rappresenta il nodo di massimo livello all'interno di un programma. Questo può contenere 0 o più dichiarazioni (di funzioni o di variabili) e 0 o più statements, e deve essere seguito da un end of file (EOF). Per rimuovere la possibilità di avere definizioni annidate di funzioni, l'elemento block è stato modificato per consentire solo dichiarazioni di variabili. Questa modifica consente anche di rilevare codice esterno alle parentesi graffe più esterne, segnalando l'errore.

Non sono state necessarie ulteriori modifiche per impedire la mutua ricorsione, in quanto senza un costrutto in grado di definire prototipi questa non è possibile e non verrà svolta una visita preliminare dall'analizzatore semantico per raccogliere gli id di tutte le funzioni.

1.2 Contenuto della relazione

La qui presente relazione è divisa in capitoli, e ogni capitolo (fatta eccezione per l'introduzione) rappresenta uno dei quattro esercizi dell'esame: analisi lessicale, costruzione della symbol table, analisi semantica, creazione (ed esecuzione) del codice intermedio e feature addizionali. In ogni capitolo, sarà presente un'introduzione all'argomento, note su come l'esercizio è stato superato ed esempi che mostrano il progetto al lavoro. I file contenenti i codici sorgenti presentati nella sezione esercizi sono tutti contenuti all'interno della cartella "Examples".

2 Esercizio 1: Analisi lessicale

Una volta effettuate le modifiche precedentemente descritte alla grammatica iniziale, è stato utilizzato ANTLR per auto-generare le classi di parser e lexer, e si è predisposto il runner per leggere dal file di input (inizialmente definito all'interno del main), estrarne i token e effettuare il parsing partendo dall'elemento "program".

Sono stati poi definiti tutti gli elementi della grammatica sotto forma di nodi, in modo da avere ognuno di essi come classe che implementa l'interfaccia comune "Node". Ultimato il lavoro sui nodi, ci si è poi soffermati sull'implementare il visitor per SLP: questa classe, contenuta nel package "ast", contiene gli override dei metodi predefiniti della classe generica generata da ANTLR. Questi vengono chiamati quando si esegue la funzione "visit()" su un elemento trovato dal parser, e consente di costruire l'albero della sintassi con una strategia DFS.

Verificato il funzionamento del parser, si è poi implementata la funzione "print" di ogni nodo, in modo da ottenere una rappresentazione grafica dell'albero prodotto.

È stato inoltre implementato un error handler per errori lessicali e sintattici, che esegue la "cattura" di eccezioni legate all'analisi lessicale e sintattica e le registra all'interno di un ArrayList di stringhe, per poi inserirle all'interno di un file ".log" dallo stesso nome del file sorgente. Questa classe ("SimpLanPlusErrorHandler") estende la classe base di ANTLR "BaseErrorListener", eseguendo l'override del metodo "syntaxError". Nel caso siano presenti errori, la compilazione viene interrotta e vengono presentati all'utente gli errori riscontrati.

2.1 Esempi

Sono di seguito riportati 3 diversi esempi di codice, uno per categoria di errore (lessicale, sintattico, semantico), con di lato riportato l'output del compilatore.

Codice SLP	Output

`{`
`  int b = 42;`
`  int a_1 = 3;`
`  print(b);`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[!] An error occurred at line 3, character 9 :token recognition error at: '_'
	[!] An error occurred at line 3, character 10 :no viable alternative at input 'inta1'

`{`
`  int c = 1;`
`  print(c);`
`}`
`c = 2;`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[!] An error occurred at line 5, character 0 :mismatched input 'c' expecting <EOF>

`{`
`  int c;`
`  int fun1(int b){`
`  int c;`
`  a=1; b=1;`
`  }`
`  fun1(1, c);`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[!] Variable a not declared at line 5.

Nel primo esempio viene mostrato il comportamento del compilatore in presenza di un simbolo ("_") non presente all'interno della grammatica (errore lessicale). Il messaggio d'errore, infatti, indica un token sconosciuto e l'impossibilità di collegarlo ad una grammatica nota.

Nel secondo esempio viene mostrato il comportamento del compilatore in presenza di codice esterno al blocco “program” (errore sintattico): il compilatore segnala la presenza del carattere estraneo “c” dal momento che era atteso l’end of file.

Nel terzo ed ultimo esempio viene mostrato il comportamento del compilatore in presenza di un assegnamento ad una variabile “a” non dichiarata all’interno di un corpo della funzione (errore semantico): all’interno del file “.log” viene segnalata la mancata dichiarazione della variabile “a”.

3 Esercizio 2: Implementazione Symbol Table

La Symbol Table è stata implementata come una lista di Hash Table, soluzione adottata anche all'interno del progetto d'esempio. La scelta è stata fatta tenendo conto che sia l'approccio scelto che quello scartato sono equivalenti, avendo entrambi vantaggi e svantaggi. Nella figura che segue, è possibile avere una rappresentazione grafica di come viene gestita la Symbol Table di un codice d'esempio:

```
1.  {
2.      int c;
3.      int fun1(){
4.          return 1;
5.      }
6.      int fun2(int a){
7.          int b = 0;
8.          {
9.              int c = 1;
10.             fun1();
11.         }
12.         return b;
13.     }
14.
15.
16.     {
17.         int a;
18.         {
19.             int b;
20.         }
21.         print fun1();
22.     }
23. }
```

La symbol table, in riga 19, sarà la seguente:

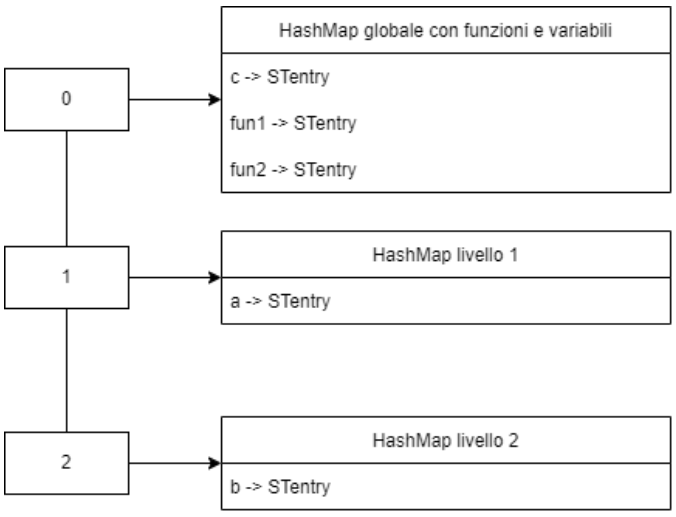

Figura 1: Esempio ST

È inoltre importante notare come, quando si entra all'interno del corpo di una funzione, l'ambiente globale viene "escluso", di fatto creando un nuovo ambiente locale che eredita da quello globale solo i simboli delle funzioni, e che rimane completamente isolato rispetto a quello esterno. Ad esempio, alla riga 9 del precedente esempio, si avrà:

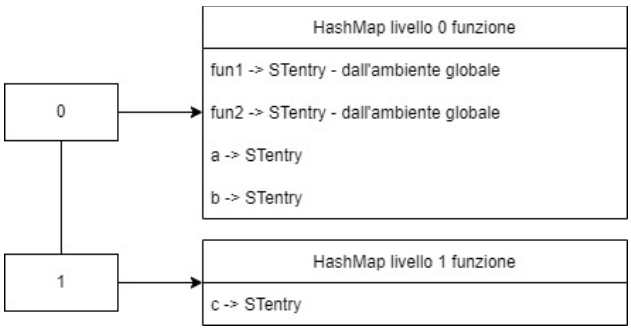

Figura 2: Esempio ST

Questa strategia consente di non permettere lo shadowing di parametri ed elementi globali (funzioni) all'interno del blocco statement della funzione, ma lo consente comunque all'interno dei sottoblocchi.

La classe "Environment" presente nel package "utils" contiene al suo interno un oggetto SymbolTableManager, che contiene e gestisce l'ArrayList di HashMap che rappresenta la SymbolTable. L'HashMap contiene, per ogni id, la sua STentry: questa contiene il nesting level di quella variabile, il tipo (memorizzato con un TypeNode), il suo offset, l'effetto e se è o meno una funzione.

Si è poi proceduto a implementare i metodi "checkSemantics" di tutti i nodi, in modo da poter verificare la validità della Symbol Table scorrendo l'albero durante una visita DFS, da svolgersi dopo la costruzione dell'albero per mano del parser. Gli errori raccolti durante la visita vengono poi mostrati all'utente (e inseriti all'interno del file ".log"), e nel caso ve ne siano la compilazione viene interrotta e il programma rigettato.

3.1 La struttura di STentry

La classe STentry è l'astrazione mediante la quale viene tenuta traccia delle caratteristiche dei simboli all'interno della symbol table. Contiene il nesting level del simbolo, il suo offset (1 per i boolean, 4 per gli integer e 0 per le funzioni), l'effetto (oggetto di classe Effect) e il suo tipo (oggetto di classe TypeNode). Contiene, inoltre, un flag booleano "isFn" che consente di indicare se il simbolo è una funzione o meno, il che consente di rilevare il tentativo di assegnamento valore ad una funzione oppure la chiamata di una variabile.

3.2 Esempi

Codice SLP	Output

`{`
`  int a;`
`  int b=2;`
`  b = b+2;`
`  print(b);`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[L] Environment is good!
	[W] Symbol a is unused in program.
	[L] Program is valid.
	[L] Assembling...
	[L] Code ready for execution!
	--- 	
	4

`{`
`  int c;`
`  {`
`  int a;`
`  {`
`  int c;`
`  }`
`  print fun1();`
`  }`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[!] Function fun1 not declared at line 8.

`{`
`  int fun(int a){`
`  int a = 2;`
`  }`
`  int fun = 1;`
`  function(1);`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[!] Variable id a already declared at line 3.
	[!] Variable id fun already declared at line 5.
	[!] Function function not declared at line 6.

Nel primo esempio è possibile vedere in azione l'analizzatore semantico, che segnala come la variabile a sia inutilizzata. Nel secondo, viene rilevato l'utilizzo di un identificatore inesistente. Nel terzo si può osservare come venga rilevato un tentativo di shadowing di un parametro di funzione, la ridichiarazione di una variabile con nome già utilizzato e la chiamata di una funzione non dichiarata.

4 Esercizio 3: Analisi Semantica

Una volta implementati i meccanismi per la gestione della Symbol Table, si è proseguito nel progetto lavorando sull'analisi semantica, ovvero verificare la correttezza dei tipi, l'uso di variabili non inizializzate e la dichiarazione di variabili non utilizzate. Per Γ,nest_level, offset $\vdash$ dec: Γ,nest_level, offset$'$ fare ciò, si è implementata per tutti i nodi dell'AST la funzione typeCheck, e le classi che ereditano da TypeNode:

- VoidTypeNode, tipo "void";
- IntTypeNode, tipo "int";
- BoolTypeNode, tipo "bool";
- FunctionTypeNode, che ha come tipo di ritorno uno dei 3 precedentemente indicati e contiene al suo interno un arraylist di ArgTypeNode;
- ArgTypeNode, che ha un tipo tra i primi 3 ed un flag booleano per indicare il passaggio con riferimento.

4.1 Premesse

4.1.1 Prototipi

L'ambiente "Γ" è una funzione definita come:

$$\Gamma: ID \rightarrow \text{Type, offset, effect, isFn, nest\_level}$$

- **Dichiarazioni**

$$\Gamma\text{,nest\_level, offset} \vdash dec: \Gamma'\text{,nest\_level, offset}'$$

- **Statement**

$$\Gamma\text{,nest\_level, offset} \vdash stm: \Gamma'\text{,nest\_level, offset, type}$$

- **Argomenti**

$$\Gamma\text{,nest\_level, offset} \vdash arg: \Gamma'\text{,nest\_level, offset}'$$

- **Espressioni**

$$\Gamma \vdash exp: \Gamma'\text{,type}$$

4.1.2 Funzioni di supporto

Funzione Top

Dato in input una pila di ambienti $\Gamma: \Gamma_1 \times \Gamma_2 \times ... \times \Gamma_n$ restituisce l'ultimo ambiente inserito.

Funzione FilterFn

Dato un ambiente in input, restituisce un ambiente che contiene solamente i simboli che posseggono il flag is_fn a True.

Funzione ArgTypes

Data in input una lista di parametri restituisce una lista contenente i tipi dei parametri.

Funzione Ret
Dati due tipi in input restituisce il primo se diverso da void, diversamente il secondo. Il compilatore SLP si ferma al primo return che incontra, e in mancanza di un return considera come tipo di ritorno il void.

Funzione Max
Dati due effetti in input, restituisce il maggiore tra i due, sapendo che declared $<$ init $<$ used $<$ $\top$.

Funzione Pop
Data una pila di ambienti in input, la restituisce senza l'ultimo ambiente inserito (che si trova in testa).

Funzione Update

Dati due ambienti in input la funzione restituirà il primo ambiente aggiornando il valore

Funzione EffectMax
Dati due ambienti in input ne restituisce uno con gli effetti modificati in base alla seguente tabella:

Effetto 1	Effetto 2	Risultato
Declared	Declared	Declared
Declared	Initialized	Declared
Declared	Used	Declared
Initialized	Initialized	Initialized
Used	Initialized	Used
Used	Used	Used
T	Declared	T
T	Initialized	T
T	Used	T
T	T	T

Gli effetti sono tra di loro commutativi, e per ridurre la dimensione della tabella le combinazioni a effetti invertiti sono state omesse. È dimostrabile come questa funzione sia monotona crescente: Sia a un effetto $\leq$ di b e $a' \leq b' \Rightarrow f(a, a') \leq f(b, b')$.

Funzione UpdateFun

Dati due ambienti in input, restituisce il primo ambiente con gli effetti delle funzioni aggiornati a quelli del secondo.

4.2 Regole di inferenza

4.2.1 ProgramNode

$$\frac{\emptyset, 0, 8 \vdash \text{decs}: \Gamma, 0, \text{offset} \quad \Gamma, 0, \text{offset} \vdash \text{stms}: \Gamma', 0, \text{offset}, \text{t\_ret}}{\vdash \{\text{DecSeq StmSeq}\}} \text{ [ProgramNode]}$$

Regola che rappresenta il blocco programma principale, che consente di avere dichiarazioni di variabili, funzioni e statements. L'offset iniziale viene posto ad 8, in quanto all'inizio dello stack vengono memorizzati il Return Address e il Frame Pointer. Questo viene fatto in tutti gli ambienti per una questione di uniformità a discapito di ottimizzazione dello spazio.

4.2.2 DecSeqNode

$$\frac{\Gamma, \text{nest\_level}, \text{offset} \vdash \text{dec}: \Gamma', \text{nest\_level}, \text{offset}' \quad \Gamma', \text{nest\_level}, \text{offset}' \vdash \text{decs}: \Gamma'', \text{nest\_level}, \text{offset}''}{\Gamma, \text{nest\_level}, \text{offset} \vdash \text{dec decs}: \Gamma'', \text{nest\_level}, \text{offset}''} [\text{DecSeq}]$$

Regola che consente il typecheck della sequenza di dichiarazioni, dove dec è la cima della pila di dichiarazioni e decs è la coda.

4.2.3 DecVarBool / DecVarBoolInit

$$\frac{ID \notin \text{Top}(\Gamma) \quad \Gamma' = \Gamma.add(ID \rightarrow \text{bool}, \text{offset}, \text{declared}, \text{false}, \text{nest\_level})}{\Gamma, \text{nest\_level}, \text{offset} \vdash \text{bool } ID: \Gamma', \text{nest\_level}, (\text{offset}+1)} [\text{DecVarBool}]$$

$$\frac{\Gamma \vdash exp: \Gamma', T \;\; T == \text{bool} \;\; ID \notin \text{Top}(\Gamma) \quad \Gamma'' = \Gamma'.add(ID \rightarrow \text{bool}, \text{offset}, \text{init}, \text{false}, \text{nest\_level})}{\Gamma, \text{nest\_level}, \text{offset} \vdash \text{bool } ID = exp: \Gamma'', \text{nest\_level}, (\text{offset}+1)} [\text{DecVarBoolInit}]$$

Regole per il typechecking delle dichiarazioni di variabili booleane, con o senza inizializzazione.

4.2.4 DecVarInt / DecVarIntInit

$$\frac{ID \notin \text{Top}(\Gamma) \quad \Gamma' = \Gamma.add(ID \rightarrow \text{int}, \text{offset}, \text{declared}, \text{false}, \text{nest\_level})}{\Gamma, \text{nest\_level}, \text{offset} \vdash \text{int } ID: \Gamma', \text{nest\_level}, (\text{offset}+4)} [\text{DecVarInt}]$$

$$\frac{\Gamma \vdash exp: \Gamma', T \;\; T == \text{int} \;\; ID \notin \text{Top}(\Gamma) \quad \Gamma'' = \Gamma'.add(ID \rightarrow \text{int}, \text{offset}, \text{init}, \text{false}, \text{nest\_level})}{\Gamma, \text{nest\_level}, \text{offset} \vdash \text{int } ID = exp: \Gamma'', \text{nest\_level}, (\text{offset}+4)} [\text{DecVarIntInit}]$$

Regole per il typechecking delle dichiarazioni di variabili intere, con o senza inizializzazione.

4.2.5 DecFunNode

$$\frac{\begin{array}{c} ID \notin Top(\Gamma)\ \Gamma' = \Gamma.add(ID \rightarrow \text{argsTypes(args)} \rightarrow T), 0, \text{init}, \text{true}, \text{nest\_level})\ \Gamma_1' = \text{FilterFn}(\Gamma')\ T == T_{ret} \\ \Gamma_1', 0, 8 \vdash decvars: \Gamma_1'', 0, \text{offset}' \\ \Gamma_1'', \text{offset}' \vdash args: \Gamma_1''', 0, \text{offset}'' \\ \Gamma_1''', 0, \text{offset}'' \vdash \text{stms}: \Gamma_1'''', 0, \text{offset}'' \end{array}}{\Gamma, \text{nesting\_level}, \text{offset} \vdash T\ ID\ (args)\ \{\text{DecVars StmSeq}\}: \text{UpdateFun}(\Gamma_1', \Gamma_1''''), \text{nesting\_level}, \text{offset}}$$

Regola che consente il typecheck della dichiarazione di funzioni, eseguendo il typecheck di argomenti e del corpo della funzione. Il blocco DecVars può essere visto come un caso particolare di DecSeq, all'interno del quale non sono presenti dichiarazioni di funzioni.

4.2.6 ArgNodes

ArgInt

$$\frac{ID \notin \text{Top}(\Gamma)\ \Gamma.add(\text{int}, \text{offset}, \text{initialized}, \text{false}, \text{nesting\_level}) = \Gamma'}{\Gamma, \text{nesting\_level}, \text{offset} \vdash \text{int } ID : \Gamma', \text{nesting\_level}, (\text{offset}+4)} [ArgInt]$$

ArgBool

$$\frac{ID \notin \text{Top}(\Gamma)\ \Gamma.add(\text{bool}, \text{offset}, \text{initialized}, \text{false}, \text{nesting\_level}) = \Gamma'}{\Gamma, \text{nesting\_level}, \text{offset} \vdash \text{bool } ID : \Gamma', \text{nesting\_level}, (\text{offset}+1)} [ArgBool]$$

ArgVarInt

$$\frac{ID \notin \text{Top}(\Gamma)\ \Gamma.add(\text{int}, \text{offset}, \text{initialized}, \text{false}, \text{nesting\_level}) = \Gamma'}{\Gamma, \text{nesting\_level}, \text{offset} \vdash \text{var int } ID : \Gamma', \text{nesting\_level}, (\text{offset}+8)} [ArgVarInt]$$

ArgVarBool

$$\frac{ID \notin \text{Top}(\Gamma)\ \Gamma.add(\text{bool,offset,initialized,false,nesting\_level}) = \Gamma'}{\Gamma\text{,nesting\_level, offset} \vdash \text{var bool } ID : \Gamma'\text{,nesting\_level, (offset+5)}}\ [ArgVarBool]$$

Regole che consentono il typecheck degli argomenti inseriti all'interno della dichiarazione di funzione.

4.2.7 StmSeq

$$\frac{\begin{array}{c}\Gamma\text{, nest\_level, offset} \vdash \text{stm}: \Gamma'\text{,nest\_level, offset}, T'\ \Gamma'\text{,nesting\_level,offset} \vdash \text{StmSeq}: \Gamma'', \\ \text{nesting\_level,offset}, T'' \\ T_{ret} = \text{Ret}(T', T'')\end{array}}{\Gamma\text{,nest\_level,offset} \vdash \text{stm StmSeq}: \Gamma''\text{,nest\_level,offset}, T_{ret}}\ [\text{StmSeq}]$$

Regola che consente il typecheck della lista degli statement, con seq che rappresenta la cima della pila degli stamement e StmSeq il resto del corpo.

4.2.8 StmAsgn

$$\frac{\begin{array}{c}ID \in dom(\Gamma)\ \Gamma(ID) = (T_{ID}, \text{offset}_{ID}, \text{effect}_{ID}, \text{is\_fn}_{ID}, \text{nest\_level}_{ID}) \\ \text{is\_fn}_{ID} == \text{false}\ \Gamma \vdash exp: \Gamma', T_{exp}\ T_{ID} == T_{exp} \\ \Gamma'' = \Gamma''.ID = (T_{ID}, \text{offset}_{ID}, Max(\text{effect}_{ID}, \text{init}), \text{is\_fn}_{ID}, \text{nest\_level}_{ID})\end{array}}{\Gamma\text{.nest\_level,offset} \vdash ID = exp; : \Gamma''\text{,nest\_level,offset,void}}\ [\text{StmAsgn}]$$

Regola che consente il typecheck dell'assegnamento di valori ad una variabile, la quale verifica inoltre che ID non sia una funzione.

4.2.9 StmPrint

$$\frac{\Gamma \vdash \text{exp}: \Gamma', T_{\text{exp}}}{\Gamma\text{,nest\_level,offset} \vdash \text{print exp;}: \Gamma'\text{,nest\_level,offset}, T_{\text{exp}}}\ [\text{StmPrint}]$$

Regola che consente il typecheck dell'istruzione print.

4.2.10 StmRetVoid / StmRet

$$\frac{}{\Gamma\text{,nest\_level,offset} \vdash \text{return;}: \Gamma\text{,nest\_level,offset,void}}\ [\text{StmRetVoid}]$$

$$\frac{\Gamma \vdash \text{exp}: \Gamma', T_{\text{exp}}}{\Gamma\text{,nest\_level,offset} \vdash \text{return exp;}: \Gamma'\text{,nest\_level,offset}, T_{\text{exp}}}\ [\text{StmRet}]$$

Regola che consente il typecheck dell'istruzione return, con o senza espressione.

4.2.11 StmIfThenElse / StmIfThen

$$\frac{\begin{array}{c}\Gamma \vdash exp: \Gamma', T_{\text{exp}}\ T_{\text{exp}} == \text{bool} \\ \Gamma'\text{,nest\_level,offset} \vdash \text{stm}_1: \Gamma''\text{,nest\_level,offset}, T_{s1} \\ \Gamma''\text{,nest\_level,offset} \vdash \text{stm}_2: \Gamma'''\text{,nest\_level,offset}, T_{s2} \\ T_{s1} == T_{s2}\end{array}}{\Gamma\text{, nest\_level,offset} \vdash if\ \text{(exp) stm}_1\ \text{else stm}_2\text{:EffectMax}(\Gamma'', \Gamma'''),\text{nest\_level, offset}, T_{s1}}\ [\text{StmIfThenElse}]$$

$$\frac{\begin{array}{c}\Gamma \vdash exp: \Gamma', T_{\text{exp}}\ T_{\text{exp}} == \text{bool} \\ \Gamma'\text{,nest\_level,offset} \vdash \text{stm}_1: \Gamma''\text{,nest\_level,offset}, T_{s1}\end{array}}{\Gamma\text{, nest\_level,offset} \vdash if\ \text{(exp) stm}_1 : \text{EffectMax}(\Gamma', \Gamma''),\text{nest\_level, offset}, T_{s1}}\ [\text{StmIfThen}]$$

Regola che consente il typechecking delle strutture ITE/IT.

4.2.12 StmCall / StmCallEmpty

$$\frac{\begin{array}{c} ID \in Dom(\Gamma)\ \Gamma(ID).\text{is\_fn} == \text{true} \\ \text{T} = \Gamma(ID).\text{type} \quad \Gamma(\text{ID}).\text{effect}{=}Max(\Gamma(\text{ID}).\text{effect}, \text{used}) \\ \left(\Gamma^{i-1} \vdash \text{exp}_\text{i}: \Gamma^{i}, T'''_{\,i}\ T'''_i == T'_i\right)^{i=1\to m} \\ \Gamma' = \Gamma^m \end{array}}{\begin{array}{c} (Var_i \in Dom(\Gamma)\ \Gamma' = \Gamma'(\text{Var}_i).\text{effect} = Max(\Gamma'(\text{Var}_i).\text{effect}, \text{used})\ \Gamma'(\text{Var}_i).\text{is\_fn} == \text{false})^{i=1\to n} \\ \Gamma,\text{nest\_level,offset} \vdash ID(\text{exp}_1, \dots, \text{exp}_m, \text{Var}_1, \dots, \text{Var}_n);: \Gamma',\text{nest\_level,offset,T.fun\_type} \end{array}}\ [\text{StmCall}]$$

$$\frac{\begin{array}{c} ID \in Dom(\Gamma)\ \Gamma(ID).\text{is\_fn} == \text{true} \\ \text{T} = \Gamma(ID).\text{type} \quad Max(\Gamma(\text{ID}).\text{effect}, \text{used}) \end{array}}{\Gamma,\text{nest\_level,offset} \vdash ID();: \Gamma',\text{nest\_level,offset,T.fun\_type}}\ [\text{StmCallEmpty}]$$

Regole che consentono il typecheck della chiamata di funzioni, dove T contiene tutti i tipi delle espressioni e delle variabili var in modo ordinato (ad esempio, $T = \{T'_1, \dots, T'_m, VarT''_1, \dots, VarT''_n\}$ e il tipo di ritorno della funzione (T.fun_type).

4.2.13 StmBlockDecs / StmBlock

$$\frac{\begin{array}{c} \Gamma' = \Gamma \cdot [] \ \text{nest\_level}' = (\text{nest\_level} + 1) \\ \Gamma', \text{nest\_level}', 4 \vdash \text{DecVars}: \Gamma'', \text{nest\_level}', \text{offset}' \\ \Gamma'', \text{nest\_level}', \text{offset}' \vdash \text{StmSeq}: \Gamma''', \text{nest\_level}', \text{offset}', T_{stm} \\ \Gamma'''' = Pop(\Gamma''') \end{array}}{\Gamma,\text{nest\_level,offset} \vdash \{\text{DecVars StmSeq}\}: \Gamma'''',\text{nest\_level,offset}, T_{stm}}\ [\text{StmBlockDecs}]$$

$$\frac{\begin{array}{c} \Gamma' = \Gamma \cdot [] \ \text{nest\_level}' = (\text{nest\_level} + 1) \\ \Gamma',\text{nest\_level}, 4 \vdash \text{StmSeq}: \Gamma'',\text{nest\_level}, 4, T_{stm} \\ \Gamma''' = Pop(\Gamma'') \end{array}}{\Gamma,\text{nest\_level,offset} \vdash \{\text{StmSeq}\}: \Gamma''',\text{nest\_level,offset}, T_{stm}}\ [\text{StmBlock}]$$

Regole che consentono il typecheck dei blocchi, che contengano o meno dichiarazioni di variabili. Per quanto riguarda l'offset che viene utilizzato per il typecheck di decvars (nella prima regola) e stmseq (nella seconda), si utilizza il 4 dal momento che a differenza di una chiamata a funzione in un blocco non è necessario memorizzare un return address.

4.2.14 Exp

IntExp

$$\frac{}{\Gamma \vdash int: \Gamma,\text{int}}\ [\text{IntExp}]$$

BoolExp

$$\frac{}{\Gamma \vdash bool: \Gamma,\text{bool}}\ [\text{BoolExp}]$$

IDExp

$$\frac{ID \in dom(\Gamma)\ \Gamma(ID).\text{effect} \geq \text{init}\ \mathrm{T_{id}} = \Gamma(ID).\text{type} \quad \Gamma' = \Gamma(ID).\text{effect} = Max(\Gamma(\text{ID}).\text{effect}, \text{used})}{\Gamma \vdash ID: \Gamma', T_{id}}\ [\text{IDExp}]$$

BooleanOpExp

$$\frac{\Gamma \vdash \text{exp}_1: \Gamma', \text{T}'\ \Gamma' \vdash \text{exp}_2: \Gamma'', T''\ \ T' == T'' == \text{bool}}{\Gamma \vdash \text{exp}_1 \$\ \text{exp}_2: \Gamma'', \text{bool}}\ [\text{BooleanOpExp}]$$

Dove $ può essere "&&" oppure "||".

MathOpExp

$$\frac{\Gamma \vdash \text{exp}_1: \Gamma', \text{T}'\ \Gamma' \vdash \text{exp}_2: \Gamma'', T''\ \ T' == T'' == \text{int}}{\Gamma \vdash \text{exp}_1 \$\ \text{exp}_2: \Gamma'', \text{int}}\ [\text{MathOpExp}]$$

Dove $ può essere "+", "-","*" oppure "/".

ComparisonOpExp

$$\frac{\Gamma \vdash \text{exp}_1: \Gamma', \text{T}'\ \Gamma' \vdash \text{exp}_2: \Gamma'', T''\ \ T' == T'' == \text{int}}{\Gamma \vdash \text{exp}_1 \$\ \text{exp}_2: \Gamma'', \text{bool}}\ [\text{ComparisonOpExp}]$$

Dove $ può essere ">","<",">=" oppure "<=".

EqualsExp

$$\frac{\Gamma \vdash \text{exp}_1: \Gamma', \text{T}'\ \Gamma' \vdash \text{exp}_2: \Gamma'', T''\ \ T' == T''}{\Gamma \vdash \text{exp}_1 \$\ \text{exp}_2: \Gamma'', \text{bool}}\ [\text{EqualsExp}]$$

Dove $ può essere "==" oppure "!=".

NotExp

$$\frac{\Gamma \vdash \text{exp}: \Gamma', T\ T == \text{bool}}{\Gamma \vdash !\,\text{exp}: \Gamma', \text{bool}}\ [\text{NotExp}]$$

NegExp

$$\frac{\Gamma \vdash \text{exp}: \Gamma', T\ T == \text{int}}{\Gamma \vdash -\text{exp}: \Gamma', \text{int}}\ [\text{NegExp}]$$

CallExp

$$\frac{\begin{array}{c} ID \in Dom(\Gamma)\ \Gamma(ID).\text{is\_fn} == \text{true} \\ \mathrm{T} = \Gamma(ID).\text{type} \quad \Gamma(\text{ID}).\text{effect}{=}Max(\Gamma(\text{ID}).\text{effect}, \text{used}) \\ \left(\Gamma^{i-1} \vdash \text{exp}_\text{i}: \Gamma^i, T'''_i\ T'''_i == T'_i\right)^{i=1\to m} \\ \Gamma' = \Gamma^m \end{array}}{\left(Var_i \in Dom(\Gamma)\ \Gamma' = \Gamma'(\text{Var}_i).\text{effect} = Max(\Gamma'(\text{Var}_i).\text{effect}, \text{used})\ \Gamma'(\text{Var}_i).\text{is\_fn} == \text{false}\right)^{i=1\to n}}{\Gamma \vdash ID(\text{exp}_1, \dots, \text{exp}_m, \text{Var}_1, \dots, \text{Var}_n);: \Gamma', \text{T.fun\_type}}\ [\text{CallExp}]$$

$$\frac{ID \in Dom(\Gamma)\ \Gamma(ID).\text{is\_fn} == \text{true} \quad \text{T} = \Gamma(ID).\text{type} \quad Max(\Gamma(\text{ID}).\text{effect}, \text{used})}{\Gamma \vdash ID();: \Gamma', \text{T}.\,\text{fun\_type}}\ [\text{CallEmptyExp}]$$

4.3 Esempi

Codice SLP	Output

`{`
`  int a;`
`  int b;`
`  int c = 1 ;`
`  if (c > 1)`
`  { b = c ; }`
`  else`
`  { a = b ; }`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[L] Environment is good!
	[!] Variable b not initialized at line 8.

`{`
`  int a;`
`  int b;`
`  int c ;`
`  void f(int n, var int x){`
`  x = n ;`
`  }`
`}`
`f(1,a) ; f(2,b) ; f(3,c) ;`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[!] An error occurred at line 9, character 0 :mismatched input 'f' expecting <EOF>

`{`
`  int a;`
`  int b;`
`  int c = 1 ;`
`  void h(int n, var int x,`
`  var int y, var int z){`
`  if (n==0) return ;`
`  else {`
`  x = y ;`
`  h(n-1,y,z,x) ;`
`  }`
`  }`
`  h(5,a,b,c) ;`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[L] Environment is good!
	[!] Variable a not initialized at line 13.

Nel primo esempio, il compilatore segnala un errore in riga 8, rispetto all'inizializzazione di b: il problema sorge in quanto si cerca di assegnare ad a il valore di b nel branch dell'else, e questo non è accettabile in quanto b non è inizializzato.

Nel secondo esempio, viene rilevato un errore di parsing, in quanto è presente del codice all'esterno del blocco programma. Nel caso in cui si scriva il codice aggiungendo le chiamate di funzione all'interno del blocco programma, ovvero modificandolo nel modo che segue

```
{
    int a;
    int b;
    int c ;
    void f(int n, var int x){
    x = n ;
    }
    f(1,a) ; f(2,b) ; f(3,c) ;
}
```

Il compilatore segnala un simbolo inutilizzato (x, in quanto viene solo inizializzato) e un errore relativo alla variabile a, che non è stata inizializzata. In questa implementazione di SLP questo è proibito, e i parametri che vengono passati ad una funzione devono essere almeno inizializzati.

Nel terzo ed ultimo esempio viene rilevato un errore in linea 13, nel momento in cui si tenta di passare alla funzione "h" una variabile non inizializzata. Viene inoltre segnalato soltanto l'errore relativo alla variabile a ma non alla variabile b in quanto il compilatore si arresta nel momento in cui viene rilevato un errore semantico. La gestione degli errori di questo esercizio viene affidato ai singoli punti che controllano specifiche regole all'interno del codice, che all'occorrenza sollevano una TypeCheckException (definita nel package utils) che viene gestita stampando il messaggio di errore sul terminale e salvandolo all'interno del logfile.

5 Esercizio 4: Codegen e Interprete

5.1 Obiettivo

Nel seguente capitolo viene esposto il linguaggio bytecode, la gestione della memoria e l'implementazione dell'interprete. Come da specifiche il bytecode utilizza delle istruzioni per una macchina a pila.

5.1.1 Bytecode

Lista delle istruzioni bytecode utilizzate con sintassi simil-MIPS:

- 'push $x': toglie 4 al valore di $sp e salva il registro $x nei 4 byte appena liberati
- 'pop $x': salva nel registro $x la word contenuta nei primi 4 byte della pila, successivamente aggiunge 4 a $sp
- 'top $x': salva nel registro $x la word contenuta nei primi 4 byte della pila
- 'li $x n': salva nel registro $x l'integer n
- 'mov $x1 $x2': copia il valore di $x2 in $x1
- 'lw $x1 n($x2)': salva nel registro $x1 la word presente nei byte $x2 + n ... $x2 + n + 4
- 'sw $x1 n($x2)': salva nella pila la word contenuta in $x1 a partire dall'indirizzo $x2 + n
- 'lb $x1 n($x2)': salva nel registro $x1 il byte presente all'indirizzo $x2 + n
- 'sb $x1 n($x2)': salva nella pila il primo byte contenuto in $x1 all'indirizzo $x2 + n
- 'add $x1 $x2 $x3': somma i valori contenuti nei registri $x2 e $x3 e salva il risultato in $x1
- 'addi $x1 $x2 n': somma n e il valore contenuto in $x2 e salva il risultato in $x1
- 'sub $x1 $x2 $x3': sottrae $x3 da $x2 e salva il risultato in $x1
- 'subi $x1 $x2 n': sottrae n da $x2 e salva il risultato in $x1
- 'mult $x1 $x2 $x3': moltiplica $x2 e $x3 e salva il risultato in $x1
- 'multi $x1 $x2 n': moltiplica n e $x2 e salva il risultato in $x1
- 'div $x1 $x2 $x3': divide $x2 per $x3 e salva il risultato in $x1
- 'divi $x1 $x2 n': divide $x2 per n e salva il risultato in $x1
- 'lt $x1 $x2 $x3': restituisce vero in $x1 se $x2 < $x3, falso altrimenti
- 'lte $x1 $x2 $x3': restituisce vero in $x1 se $x2 <= $x3, falso altrimenti
- 'gt $x1 $x2 $x3': restituisce vero in $x1 se $x2 > $x3, falso altrimenti
- 'gte $x1 $x2 $x3': restituisce vero in $x1 se $x2 >= $x3, falso altrimenti
- 'eq $x1 $x2 $x3': restituisce vero in $x1 se $x2 == $x3, falso altrimenti
- 'neq $x1 $x2 $x3': restituisce vero in $x1 se $x2 != $x3, falso altrimenti
- 'and $x1 $x2 $x3': inserisce in $x1 l'and logico tra $x2 e $x3
- 'or $x1 $x2 $x3': inserisce in $x1 l'or logico tra $x2 e $x3
- 'not $x1 $x2': inserisce in $x1 il not bitwise di $x2 se $x2 è uguale a 0, inserisce 0 altrimenti
- 'neg $x1 $x2': inserisce in $x1 la negazione di $x2 (= -1 * $x2)
- 'print $x': stampa a terminale il valore contenuto in $x
- 'jal str': salva nel registro $ra il valore del pc e setta il pc al valore del label 'str'
- 'jr $x': setta il pc al valore di $x
- 'beq $x1 $x2 str': se $x1 è uguale a $x2 setta il pc al valore del label 'str'
- 'halt': sospende l'esecuzione del programma attivando il SLPMI se si è in modalità debug.

5.1.2 Gestione della memoria e dello Stack

Per ottenere un corretto funzionamento della memoria (gestione della pila) il bytecode ha bisogno di alcuni registri con cui lavorare, questi registri nel codice sono preceduti dal simbolo '$':

1. $a0 contiene il valore dell'ultima istruzione calcolata
2. $t0 - $t1 registri temporanei
3. $sp (stack pointer) contiene l'ultimo indirizzo di memoria utilizzato, in tutti gli indirizzi più bassi ci sarà memoria da considerarsi libera mentre in tutti gli indirizzi più alti si troverà la pila del programma
4. $fp (frame pointer) contiene l'indirizzo del record di attivazione corrente, ogni volta che viene creato un nuovo record di attivazione il vecchio fp viene salvato nello stack mentre questo registro viene aggiornto con l'indirizzo del nuovo record
5. $ra (return address) registro utilizzato esclusivamente dalla chiamata a funzione; Nel momento in cui si presenta l'istruzione di 'jal' viene salvato in questo registro l'indirizzo dell'istruzione successiva in modo da permettere di ritornare al normale flusso del programma una volta terminata la funzione

Gli indirizzi come ogni registro sono a 32 bit, mentre una singola cella di pila è della dimensione di un byte.

La pila è gestita con record di attivazione; per il blocco del programma e per ogni chiamata a funzione viene creato un nuovo record salvando sia il return address che il frame pointer mentre per ogni blocco (es. I blocchi di istruzioni degli if-then-else) viene salvato solo il frame pointer nella pila.

All'avvio l'interprete imposta i registri $sp e $fp al valore 'MEMSIZE' che indica il valore massimo della memoria, successivamente le prime istruzione che vengono eseguite da un programma servono per creare il primo record di attivazione, allocando lo spazio per le variabili e salvando il frame pointer ed il return address.

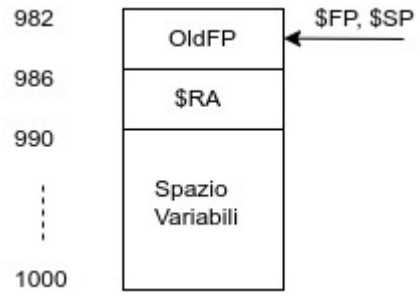

Figura 3: Stato della pila all'inizio di un programma

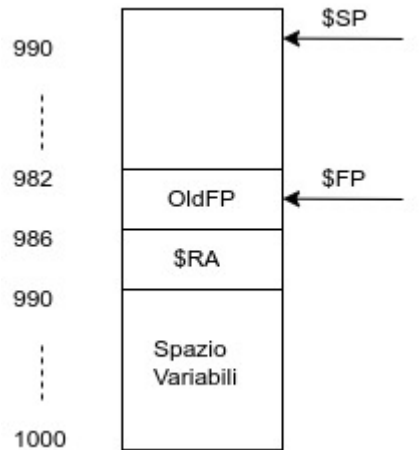

Figura 5: Stato della pila durante l'esecuzione di un programma

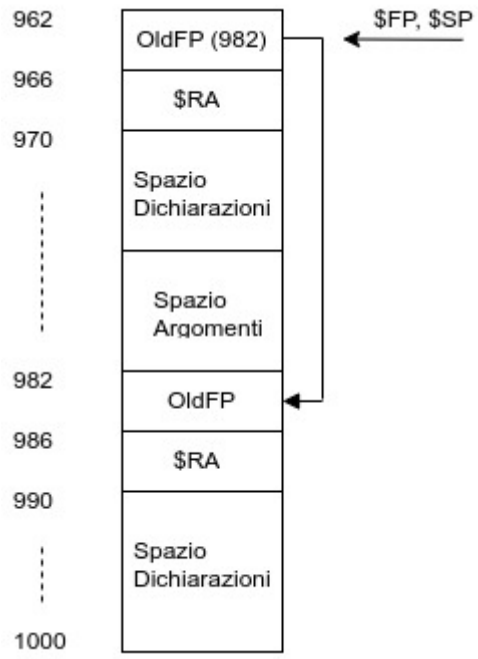

Figura 6: Stato della pila durante una chiamata a funzione

5.1.2 Visibilità dei nomi

Variabili

Le variabili sono visibili nell'activation record in cui sono state dichiarate ed in tutti i blocchi annidati tuttavia non saranno visibili all'interno di un nuovo activation record (es all'interno di una funzione).

Per poter utilizzare all'interno di una funzione una variabile esterna bisogna che la funzione accetti tra i suoi argomenti una variabile del tipo che si vuole passare e con prefisso 'var', quest'ultimo permette di effettuare un passaggio per riferimento.

Gestione del 'var'

Un argomento con il prefisso 'var' accetta in input solo variabili e non espressioni, la variabile creata ed utilizzabile all'interno della funzione sarà inizializzata al valore della variabile esterna passata alla chiamata della funzione. Al ritorno della funzione la variabile esterna sarà aggiornata all'ultimo valore assunto dalla variabile interna alla funzione.

In termini di memoria, ogni argomento con il prefisso 'var' andrà ad occupare 4 byte in più di un argomento classico, questi servono per memorizzare l'indirizzo della variabile esterna, utilizzato a sua volta al termine della funzione per rintracciare ed aggiornare la variabile esterna.

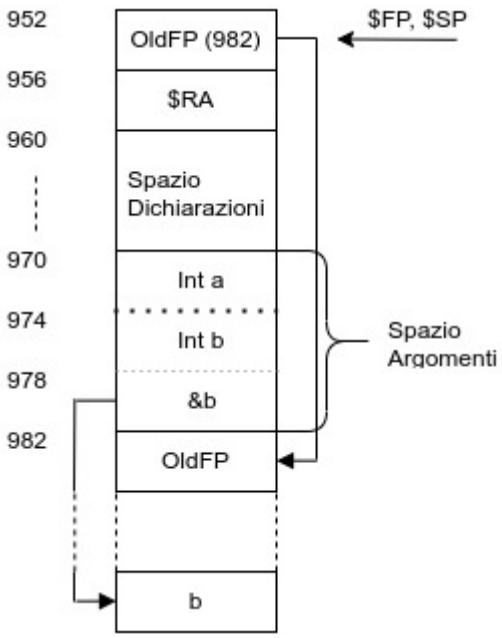

Figura 7: Esempio Activation record funzione "void fun(var int b, int a){...}"

Funzioni

Le funzioni sono dichiarabili solo nell'ambiente più esterno, tuttavia oltre ad essere visibili in qualsiasi blocco successivo sono anche visibili all'interno di se stesse, consentendo la ricorsione, ed all'interno delle funzioni dichiarate successivamente.

Generazione label

La generazione delle label viene gestita dalla classe LabelGenerator del package util, che consente di creare e memorizzare nuove label assicurando l'univocità di esse grazie ad un contatore, il cui valore viene inserito all'interno della label che viene restituita.

5.2 Esempi

Codice SLP	Output

`{`
`int x = 1;`
`void f(int n){`
`if (n == 0) { print(x) ; }`
`else { x = x * n ; f(n-1) ;`
`}`
`}`
`f(10);`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[!] Variable x not declared at line 4.
	[!] Variable x not declared at line 5.

`{`
`int u = 1 ;`
`void f(var int x, int n){`
`if (n == 0) { print(x) ; }`
`else { int y = x * n ; f(y,n-1);`
`}`
`}`
`f(u,6);`
`}`
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[L] Environment is good!
	[L] Program is valid.
	[L] Assembling...
	[L] Code ready for execution!
	---------- 	
	720

{
void f(int m, int n){
 if (m>n) { print(m+n) ;}
 else { int x = 1 ;
f(m+1,n+1) ; }
}
f(5,4);
}
	[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[L] Environment is good!
	[W] Symbol x is unused in block that starts at line 4.
	[L] Program is valid.
	[L] Assembling...
	[L] Code ready for execution!
	---------- 	
	9

Nel primo esempio, il compilatore segnala 2 istanze di variabile non dichiarata: la variabile x viene infatti dichiarata nel contesto globale del programma, ma in questa versione di SimpLanPlus non è possibile accedere a variabili globali dall'interno del corpo di una funzione. Se si modifica il codice nel seguente modo si ottiene il seguente output:

Codice SLP	Output

{
int x = 1;
void f(int n, int x){
 if (n == 0) { print(x) ; }
 else { x = x * n ; f(n-1,x)
; }
}
f(10, x);
}
	[[L] SimpLanPlus Compiler started.
	[L] Parsing...
	[L] Parse completed without issues!
	[L] Checking for semantic errors...
	[L] Environment is good!
	[L] Program is valid.
	[L] Assembling...
	[L] Code ready for execution!
	---------- 	
	3628800

Il secondo esempio viene compilato ed eseguito senza alcun problema, mentre il terzo segnala la presenza di una variabile non utilizzata.

Se si invoca la funzione f con i parametri in ordine invertito (f(4,5)) nell'ultimo esempio, l'output è il seguente:

```
[L] SimpLanPlus Compiler started.
[L] Parsing...
[L] Parse completed without issues!
[L] Checking for semantic errors...
[L] Environment is good!
[W] Symbol x is unused in block that starts at line 4.
[L] Program is valid.
[L] Assembling...
[L] Code ready for execution!
-----------------------
Error: out of memory. Please adjust the amount of allocated memory with the -m param.
```

La causa dell'errore risiede in una ricorsione infinita del codice, che consuma completamente lo spazio dello stack dell'interprete, il quale consiglia di tentare con un maggior quantitativo di memoria.

6 Extra

Durante lo sviluppo del compilatore/interprete di SLP ci si è accorti che sarebbe stato interessante aggiungere alcune funzionalità al programma, per renderlo più vicino ad un compilatore vero e proprio. Questi extra consistono in comandi aggiuntivi e meccanismi automatici che consentono di arricchire e migliorare l'esperienza di un eventuale utente del progetto, e consistono in:

- Dimensione dello stack personalizzabile;
- Caching;
- Error reporting con riferimenti precisi;
- SimpLanPlus Memory Inspector.

6.1 Dimensione dello stack

Di default, la dimensione dello stack dell'interprete SLP è di 100000 byte, e questo è stato necessario per tutti quanti gli esempi che abbiamo preparato per questa relazione. Tuttavia, è possibile che per certi programmi sia necessaria una quantità di memoria superiore. Con il parametro -m [Integer] è possibile definire una dimensione di memoria in byte personalizzata, a patto che il numero sia entro i limiti rappresentativi del tipo integer e che il sistema abbia abbastanza memoria disponibile (in caso contrario, verrà mostrato un errore all'utente).

6.2 Caching

L'unico modo per eseguire un programma SLP è quello di farlo eseguire attraverso l'interprete, ma questo richiede che il contenuto del file venga ri-compilato. Nel caso in cui il sorgente non sia cambiato dall'ultima compilazione, questo è superfluo. Per evitare ciò, SLP è dotato di un sistema di caching, che gli consente di rilevare se un file sorgente è stato modificato rispetto al suo file .asm corrispondente o meno: nel caso in cui non sia stato modificato, viene eseguito direttamente il file bytecode al posto del suo corrispettivo simplan.

Questo è possibile grazie al calcolo del checksum del file sorgente, che viene inserito come commento all'inizio del file asm: se esiste un file bytecode legato al file che si sta cercando di compilare e il checksum corrisponde, allora i file sono uguali. Per disattivare questa feature, usare il parametro -d 0.

6.3 Error reporting con riferimenti precisi

Il compilatore per SLP riporta errori precisi con riferimenti alla linea in cui si verifica l'errore: ogni nodo statement ha ad esso associato un numero di riga, ottenuto in fase di lexing, che consente di segnalare il punto nel codice in cui si verifica un certo errore, che viene inserito anche all'interno del logfile. Questa funzionalità rende inoltre possibile l'utilizzo del SLPMI.

6.4 SimpLanPlus Memory Inspector (SLPMI)

Il SLPMI è una funzionalità dell'interprete che consente di ispezionare il contenuto della memoria, dei registri e dei puntatori durante l'esecuzione del programma. Per utilizzarlo, è necessario avviare il compilatore con l'opzione -d seguita dai numeri di riga (separati da spazio) in cui inserire i

breakpoint, che possono essere solo contenere statement. Il parametro -d disattiva il caching, ed effettua la compilazione con un passaggio in più, ovvero il "setupBreaks". Tramite una visita in profondità, nei nodi statement con la riga che combacia a quella indicata nei parametri di avvio, viene abilitato il flag "isBreak", che viene controllato in fase di generazione di codice: se il flag è attivo, viene aggiunta una riga contenente l'istruzione "halt".

```
SLP Memory Inspector started.
Program Counter:66
Previous instruction: jal fun Next instruction:lw $a0 8 $t1
MEMORY STATUS: Frame pointer:9988 Stack pointer:9988
Registers: a0:-1 t1:48 t2:9996
Return address: 65

To inspect memory allocations, please type a range of memory addresses
in the format start end [!], where ! will show only the non-zero values.
Type q to resume execution.
Maximum available memory is 10000 bytes.
9990 10000
9990. 39        9991. 16        9992. 0         9993. 0         9994. 0         9995. 72        9996. 0         9997. 0         9998. 0         9999. 1
Press enter to proceed...
```

Se la modalità debug è attiva, quando l'interprete incontra un "halt" l'esecuzione viene sospesa, e il controllo viene passato alla funzione "MemoryInspector" della classe "Interpreter".