SimpLanPlus Project Report

Progetto per il corso di Compilatori e Interpreti del corso di
Laurea Magistrale in Informatica, Università di Bologna

Chiara Manca, Melania Ghelli

A.A. 2021/22

Indice

Installazione

Guida all'Installazione e Utilizzo su Eclipse

1. Importare la cartella SimpLanPlus all'interno del workspace di Eclipse:
   (a) File > Open Projects from File System;
   (b) Cliccare su Directory e selezionare la cartella scaricata contenente il progetto; in alternativa si può cliccare su Archive e scegliere direttamente il file .zip
   (c) Cliccare su Finish.
2. Generare l'eseguibile e lanciare l'Interprete:
   (a) Tasto destro sulla cartella del progetto (quella contentente il file pom.xml);
   (b) Selezionare Show in > Terminal;
   (c) Digitare il comando:

```
mvn clean compile assembly:single
```

   (d) Creare un file .simplan oppure scegliere un file dalla cartella examples;
   (e) Tornare al terminale e digitare il seguente comando per compilare ed eseguire l'interprete:

```
java -jar target/SimpLanPlus.jar <path_file>
```

   con il path al file scelto al posto di `<path_file>`.

Guida all'Installazione e Utilizzo da terminale Linux

1. Dopo aver estratto il contenuto dell'archivio, aprire la directory SimpLanPlus (quella contentente il file pom.xml);

2. Digitare il comando:

```
mvn clean compile assembly:single
```

3. Creare un file .simplan oppure scegliere un file dalla cartella examples;

4. Tornare al terminale e digitare il seguente comando per compilare ed eseguire l'interprete:

```
java -jar target/SimpLanPlus.jar <path_file>
```

   con il path al file scelto al posto di `<path_file>`.

Introduzione

Questo report è diviso in 4 capitoli, ciascuno dei quali si concentra su una sezione diversa dell'implementazione del compilatore e interprete per il linguaggio SimpLanPlus.

La consegna prevede la risoluzione di 4 esercizi. Il primo è condensato nel capitolo 1 ed in parte nel capitolo 2. Il secondo ed il terzo esercizio invece si trovano all'interno del capitolo 3. Il quarto ed ultimo esercizio invece si trova all'interno del capitolo 4.

SimpLanPlus

SimpLanPlus è un semplice linguaggio imperativo, in cui i parametri delle funzioni possono essere passati anche per variabile, le funzioni non sono annidate (una funzione non può essere definita nel corpo di un'altra), ed ammette la ricorsione (ma non la mutua ricorsione).

Struttura del progetto

Il package principale contiene:

- il package **antlr**, che contiene i file .g4 delle grammatiche di SimpLanPlus e SVM;
- il package **java/it/ghellimanca**, che a sua volta contiene:
  - il package **ast**, che contiene tutte le classi che rappresentano gli elementi dell'Abstract Syntax Tree;
  - il package **gen**, che contiene tutti i file generati dal tool di ANTLR, come per esempio il `Lexer`, il `Parser` e tutti i file collegati a questi due, sia per la grammatica SimpLanPlus che per la grammatica di SVM;

- il package **interpreter**, che contiene tutte le classi ed eccezioni per l'intepretazione dei programmi SimpLanPlus;
- il package **semanticanalysis**, che contiene tutte le classi ed eccezioni (e warning) utili al controllo semantico dei programmi SimpLanPlus;
- il file `SimpLanPLus.java`, il main del progetto, contiene le funzioni per compilare e interpretare i programmi SimpLanPlus.

Capitolo 1

Analisi Lessicale

L'analisi lessicale è eseguita dal Lexer, generato automaticamente da ANTLR. Questo prende in input il testo del programma da eseguire e lo restituisce diviso in TOKEN.

1.1 La classe `SimpLanPlusErrorListener`

Per default, ANTLR reindirizza tutti gli errori alla standard error. Per reindirizzarli, invece, a un file `error.txt`, abbiamo esteso la BaseErrorListener di ANTLR con la nostra classe SimpLanPlusErrorListener.

In particolare, abbiamo reimplementato il metodo `syntaxError()`, che, nonostante il nome, permette di identificare sia gli errori sintattici che quelli lessicali. Questo perché il Listener può essere utilizzato sia sui Lexer che sui Parser di ANTLR.

Il metodo implementato differenzia questi due grazie al Recognizer, che sarà il riferimento o al Parser o al Lexer. Nel caso si tratti di un errore sintattico, modifichiamo il messaggio d'errore in modo che venga sottilineato il token incriminato.

Il nostro SimpLanPlusErrorListener viene utilizzato nella funzione compile: dopo aver instanziato Lexer e Parser, aggiungiamo a questi ultimi il Listener, e, dopo aver generato l' AST dalla visita del ParseTree, controlliamo se sono stati trovati degli errori. Se è così, li scriviamo nel file degli errori.

1.2 Errori Lessicali

Un errore lessicale è un errore causato dall'uso di parole o caratteri non validi secondo la grammatica del linguaggio.

Gli errori lessicali sono spesso causati da errori di battitura o da un uso scorretto di caratteri come parentesi, virgole, punti e virgola, parentesi graffe, apici, e così via.

1.2.1 Token non riconosciuto

Il seguente esempio è contenuto nel file `example1.simplan`.

```
${
    int x = 1;
    if (x >= 0)
        print x;
    else
        return;
}
```

Questo esempio contiene il carattere $ che non è riconosciuto dalla grammatica. Di seguito l'errore inserito nel file `error.txt` dopo l'esecuzione dell'esempio.

```
Lexical errors:
line 1:0 token recognition error at: '$'
```

Capitolo 2

Analisi Sintattica

L'analisi sintattica è eseguita dal Parser, generato automaticamente da ANTLR. Questo prende in input la stream di TOKEN generati dal Lexer e genera dei Context di ANTLR, che usa per creare l'Abstract Syntax Tree (AST).

2.1 La classe `SimpLanPlusVisitorImpl`

L'AST è creato grazie alla SimpLanPlusVisitorImpl, una classe che estende la classe astratta SimpLanPlusBaseVisitor generata da ANTLR.

Abbiamo implementato le funzioni `visit` di questa in modo che, partendo dalla `visitProgram` che prende in input il ProgramContext restituito dal Parser, generino ricorsivamente l'albero sintattico con delle classi che rappresentano i nodi di questo.

Tutte queste classi estendono la interfaccia madre `Node` e rappresentano i vari elementi della grammatica SimpLanPlus.

2.2 Errori sintattici

Un errore sintattico è un errore causato dalla violazione delle regole di sintassi definite nella grammatica del linguaggio di programmazione.

Gli errori sintattici sono causati spesso da errori come la mancanza di parentesi, la mancanza di virgole, l'uso di operatori errati, la dichiarazione di variabili mancanti o malformate, l'uso di istruzioni in un ordine non valido, e così via.

Come fatto presente nel capitolo precedente, gli errori sintattici sono stati gestiti insieme a quelli lessicali con il nostro listener SimpLanPlusErrorListener. Dopo aver generato l' AST dalla visita del ParseTree, controlliamo se sono stati trovati degli errori. Se è così, li scriviamo nel file degli errori `error.txt`.

2.2.1 Dichiarazione di funzione nidificata

Il seguente esempio è contenuto nel file `example2.simplan`.

```
{
    void f() {
        int b = 20;
        void g(int x){
            if (x == 0) {
                return;
            }
            else {
                g(x - 1);
            }
        }
        g(2);
    }
}
```

Questo esempio contiene la dichiarazione della funzione g nidificata all'interno della dichiarazione della funzione f.

Genererà un errore perchè la grammatica SimpLanPlus non consente di dichiarare funzioni dentro il corpo di altre funzioni.

Di seguito l'errore inserito nel file `error.txt` dopo l'esecuzione dell'esempio.

```
Syntactic errors:
line 10:8 extraneous input 'void' expecting {'{', '}', 'int', 'bool', 'print', 'return
        void g(int x){
        ^^^^
line 21:0 extraneous input '}' expecting <EOF>
    \}
    ^
```

2.2.2 Id variabile mancante

Il seguente esempio è contenuto nel file example3.simplan.

```
{
    int = 1;
    if (x >= 0)
        print x;
    else
        return;
}
```

Questo esempio contiene un errore sintattico: manca l'id nella dichiarazione di variabile intera.

Di seguito l'errore inserito nel file error.txt dopo l'esecuzione dell'esempio.

```
Syntactic errors:
line 6:8 no viable alternative at input 'int='
    int = 1;
```

2.2.3 Errore lessicale con conseguenze sintattiche

Il seguente esempio è contenuto nel file example4.simplan.

```
{
    int # = 1;
    if (x >= 0)
        print x;
    else
        return;
}
```

Questo esempio contiene sia un errore lessicale che sintattico: il token # non è riconosciuto dalla grammatica e, di conseguenza, fa nascere un errore sintattico dovuto alla mancanza dell'id nella dichiarazione della variabile.

Di seguito l'errore inserito nel file error.txt dopo l'esecuzione dell'esempio.

```
Lexical errors:
line 6:8 token recognition error at: '#'
Syntactic errors:
line 6:10 no viable alternative at input 'int='
    int # = 1;
          ^
```

Capitolo 3

Analisi Semantica

Prende in input un Abstract Syntax Tree (AST) e verifica la presenza di errori semantici.

In questa fase vengono effettuate in ordine la Scope Check, la Type Check e la Effect Analysis. Prima di implementare il relativo codice sono state scritte le Regole di Inferenza per ognuna delle tre analisi. Per la lista completa delle regole si veda l'appendice A.

L'analisi semantica viene effettuata dai metodi `checkSemantics()`, che si occupa dei controlli di scoping e sugli effetti, e `typeCheck()`. Questi metodi vengono implementati da ogni nodo dell'AST, e chiamati a partire dal nodo radice seguendo una visita in profondità. La chiamata ai due metodi a partire dalla radice viene effettuata in modo sequenziale dalla classe principale, `SimpLanPlus`. Per i controlli di Analisi Semantica dunque vengono effettuate in totale due visite dell'albero.

Durante la visita dell'albero il compilatore memorizza variabili e funzioni in un'apposita tabella, la Symbol Table, dove assieme ai nomi si tiene traccia del nesting level e dell'offset relativi alla posizione dove vengono dichiarate.

Questa Symbol Table è stata realizzata come Stack di Hash Table. Abbiamo scelto questo approccio in quanto il codice per gli accessi ci sembrava più intuitivo.

3.1 Le classi `Environment` e `STEntry`

La classe Environment rappresenta l'ambiente, il quale contiene assieme alla Symbol Table anche i parametri globali che vengono usati per conoscere in ogni momento della computazione il livello di profondità della tabella e l'offset correnti. Durante la visita gli accessi alla tabella vengono fatti attraverso questa classe.

All'interno della tabella le variabili e le funzioni vengono memorizzate come coppie `(id -> STEntry)`, come mostrato dalla figura 3.1. Queste ultime sono strutture dati implementate nell'omonima classe che memorizzano, per ciascun nome, delle informazioni necessarie per le varie fasi dell'analisi semantica, che si possono vedere alla figura 3.2. Nelle prossime sezioni si vedrà meglio quali di questi campi vengono utilizzati ai fini dell'analisi e in che modo.

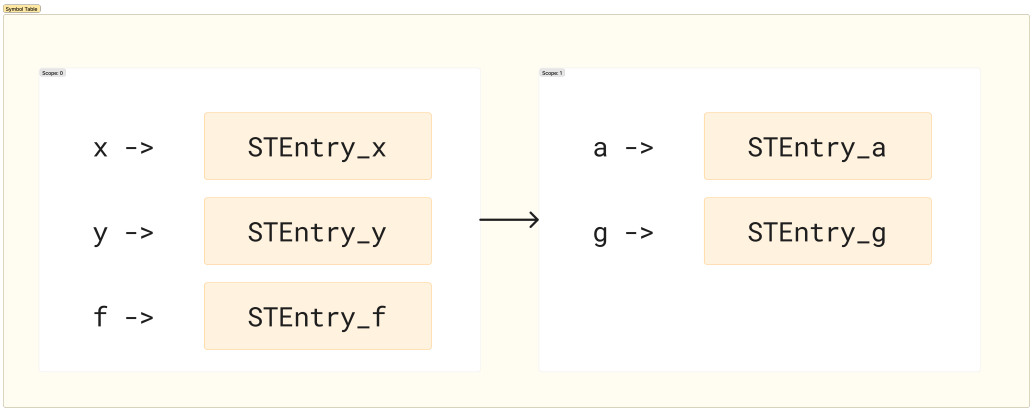

Figura 3.1: Aspetto della Symbol Table ad un generico punto dell'esecuzione di un programma

	type	nesting level	offset	varStatus	isInitAfterDec	funStatus	initPars
x	int	0	0	declared	false	[]	[]
y	bool	0	1	init	false	[]	[]
f	int x int -> bool	0	2	init	false	[init, init]	[true, false]

Figura 3.2: Dettaglio di uno scope all'interno di una Symbol Table

3.2 Scope Checking

Lo scopo di questi controlli è individuare dichiarazioni di variabili o funzioni multiple, e gli usi di variabili o funzioni non dichiarate.

I campi della `STEntry` utilizzati in questa fase sono:

- il **nestingLevel**; viene settato dalla classe Environment ad ogni nuova dichiarazione. Questa classe tiene un contatore globale, che viene incrementato all'apertura di ogni scope, decrementato all'uscita. Nella STEntry questo campo corrisponde al livello di annidamento dove la dichiarazione avviene all'interno del programma. Per ciascuno di questi livelli la Symbol Table memorizza le variabili all'interno della stessa Hash Table, che quindi coincide con lo scope.

- l'**offset**; anche questo viene settato dalla classe Environment, che all'apertura di ogni scope inizializza un contatore globale a 0. Questo contatore viene incrementato ogni volta che si processa una dichiarazione. Nella STEntry questo campo corrisponde alla distanza dalla prima dichiarazione dove la variabile di riferimento viene dichiarata all'interno dello scope. Questo campo viene settato durante l'Analisi Semantica per essere poi utilizzato durante la Code Generation, approfondita nella sezione 4.

Durante la visita in profondità dell'albero, la quale richiama i metodi `checkSemantics()` sui vari nodi, vengono fatti degli accessi in lettura o in scrittura alla Symbol Table. In base all'esito di questi accessi è possibile individuare la presenza di questi errori. I controlli di scope vengono dunque fatti dai metodi della classe `Environment`.

3.2.1 Variabili e funzioni dichiarate più volte nello stesso ambiente

Il controllo viene delegato al metodo `addDeclaration` della classe `Environment`.

Questa funzione viene chiamata per inserire una nuova entry nella tabella. Controlla che nello scope corrente non ci sia già un'occorrenza con lo stesso nome; in caso positivo solleva un'eccezione.

Durante la visita dell'albero, nel momento in cui si processa una dichiarazione, dunque nei due nodi `DecVarNode` e `DecFunNode`, le rispettive `checkSemantics()` accedono alla tabella per memorizzare la nuova dichiarazione. L'eventuale errore viene individuato controllando l'esito dell'accesso alla tabella, che in caso di collissione solleva la `MultipleDeclarationException`.

3.2.2 Variabili e funzioni non dichiarate

Anche in questo caso il controllo viene fatto da un metodo della classe Environment, ovvero `lookup()`.

Questo metodo viene invocato dalla `checkSemantics()` dei nodi `IdNode` dell'albero. Questi nodi vengono utilizzati per rappresentare ogni occorrenza di una variabile o di una funzione. Dal momento che però il controllo sulla loro dichiarazione deve essere fatto solo al momento di un uso, e non nelle occorrenze all'interno dei `DeclarationNode`, solo i nodi diversi da questi richiamano la funzione `checkSemantics()`.

Dunque durante la visita dell'albero, ogni volta che viene utilizzata una variabile o una funzione, quindi all'interno degli `StatementNode` ed `ExpressionNode`, viene effettuato un'accesso alla Symbol Table al fine di prelevare la `STEntry` dalla tabella. In caso la variabile o la funzione non ci siano, e dunque non siano state dichiarate, viene sollevata l'eccezione `MissingDeclarationException`.

3.3 Type Checking

L'obiettivo di questi controlli è quello di verificare la correttezza dei tipi: che si stia assegnando un valore del tipo corretto un una variabile dichiarata di un certo tipo, che nelle espressioni binarie due variabili abbiano un tipo concorde, ecc.

I tipi previsti dalla grammatina di SimpLanPlus sono `void`, usato solo per le funzioni, `int` e `bool`, usati sia per variabili che funzioni. L'interprete adotta una rappresentazione uniforme dei dati: a prescindere dal loro tipo occupano tutti 4 byte.

Il campo della `STEntry` utilizzato in questa fase è il **type**, il quale assume un valore che è istanza della classe `TypeNode`.

I `TypeNode` sono nodi dell'AST che rappresentano il tipo di un oggetto (variabile o funzione). Oltre a quelli corrispondenti ai tipi primari (`VoidTypeNode`, `IntTypeNode` e `BoolTypeNode`) sono stati introdotti altri due tipi:

- `VarTypeNode`, usato come wrapper per il tipo degli `ArgNode`, nel caso di variabili passate per riferimento. Contiene al suo interno il `TypeNode` corrispondente al tipo della variabile.
- `ArrowTypeNode`, usato per definire i tipi delle funzioni. E' rappresentato nella forma `(T1 x.. x Tn -> T)`, dove i primi n tipi corrispondono ai tipi dei parametri della funzione.

Per effettuare i controlli sui tipi viene fatta una seconda visita dell'AST, chiamando il metodo `typeCheck()` a partire dal nodo radice del programma. In caso di errori viene generalmente sollevata un'eccezione `TypeCheckingException`.

3.3.1 Correttezza dei tipi: conformità dei parametri attuali a quelli formali

Questo controllo viene effettuato da due metodi diversi del nodo `CallNode`.

1. All'interno del metodo `checkSemantics()` si verica che il numero di parametri attuali sia lo stesso di quelli formali. Se così non è la funzione solleva una `ParametersException`.

2. Nel metodo `typeCheck()` si controlla che i tipi siano effettivamente gli stessi. In caso contrario viene sollevata una `TypeCheckingException`.

3.4 Effect Analysis

Questi controlli hanno lo scopo di verificare come lo stato delle variabili cambia durante l'esecuzione di un programma, ma in modo statico. L'idea è quella di rappresentare tramite le regole di inferenza le potenziali modifiche fatte alle variabili, a prescindere da come si aggiorni il controllo del programma. Questo perché ci sono alcuni errori semantici che non è possibile individuare a livello statico; questi errori sono l'uso di variabili non inizializzate e la dichiarazione di variabili non utilizzate.

Gli effettivi controlli vengono fatti dal metodo `checkSemantics()`. Si è scelto di non utilizzare un metodo separato in quanto è necessario avere accesso alla `Symbol Table` corrente in ogni punto del programma.

I campi della `STEntry` utilizzati in questa fase sono:

- i campi **varStatus** e **isInitAfterDec**. Questi due vengono settati per le variabili, e memorizzano rispettivamente lo status corrente (se dichiarata, inizializzata o usata) e se la variabile è stata inizializzata in un momento successivo alla dichiarazione.

- i campi **funStatus** e **initPars**. Specularmente, vengono settati per le funzioni al fine di memorizzare rispettivamente lo status corrente dei parametri, e se questi parametri sono stati inizializzati in un momento successivo alla loro dichiarazione (all'interno del corpo della funzione, generalmente).

3.4.1 Classe `Effect` ed operazioni tra effetti

Questa classe è stata implementata per rappresentare gli stati - o meglio effetti - di variabili e funzioni. Contiene un solo campo dati di tipo intero, che memorizza

l'effetto corrente applicato all'oggetto a cui si riferisce. Può assumere uno dei seguenti valori:

- `DECLARED`, ha valore 0 ed è anche rappresentato dal simbolo $\bot$.
- `INIT`, ha valore 1; è associato a oggetti inizializzati
- `USED`, ha valore 2; viene associato a oggetti che sono stati usati
- `ERROR`, ha valore 3 ed è anche rappresentato dal simbolo $\top$; associato ad oggetti che sono in stato di errore, ad esempio perché usati senza essere stati inizializzati

Tra questi status è possibile effettuare delle operazioni. Ad esempio, ogni volta che lo status associato ad un oggetto viene aggiornato, l'esito dell'aggiornamento viene determinato dall'operazione sequenziale `seq`. Le operazioni definite sugli effetti in SimpLanPlus sono tre:

- `seq`, appena citata
- `bin`, usata nel caso di rami then ed else, serve per propagare l'effetto più critico
- `par`, usata nelle le chiamate di funzione per propagare le modifiche fatte agli status del codominio

Al fine di poter essere utilizzate direttamnte tra variabili nella forma `(id -> STEntry)`, queste operazioni vengono definite anche sulla classe `Environment`.

Per un dettaglio di come agiscono questi operatori si vedano le tabelle di verità 3.3.

SEQ	$\bot$	1	2	$\top$
$\bot$	$\bot$	1	$\top$	$\top$
1	1	1	2	$\top$
2	2	2	2	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

BIN	$\bot$	1	2	$\top$
$\bot$	$\bot$	$\bot$	$\bot$	$\top$
1	$\bot$	1	2	$\top$
2	$\bot$	2	2	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

PAR	$\bot$	1	2	$\top$
$\bot$	$\bot$	1	2	$\top$
1	1	1	2	$\top$
2	2	2	2	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Figura 3.3: Tabelle di verità per gli operatori binari sugli status

3.4.2 Uso di Variabili non Inizializzate

Questo controllo viene fatto in due momenti e in due porzioni di codice diverse.

1. **Durante l'uso della variabile**: l'errore avviene ogni qual volta si utilizza una variabile che si trova in stato diverso da `init`. In quei casi l'operazione `seq` setterà lo status di quella variabile ad errore.

2. **All'uscita di ciascun blocco**: prima di fare il pop dello scope corrente, si controlla se all'interno di questo ci sono variabili con status di errore. In quei casi viene sollevata l'eccezione `MissingInitializationException`.

Gestione variabili passate per riferimento: inizializzazione all'interno del corpo della funzione

Nel caso di variabili passate per riferimento il controllo dell'uso senza inizializzazione è stato critico a causa della possibilità di passare come parametri anche variabili solo dichiarate ma non inizializzate. Nel caso di variabili passate con la keyword `var` infatti è possibile inizializzarle anche nel corpo di una funzione.

Al fine di calcolare correttamente gli effetti nel corpo di una funzione è necessario settare gli status dei parametri a `init`. In caso contrario infatti (se queste fossero settate a `declared`, ovvero lo stato in cui si trovano i parametri formali al momento della dichiarazione di funzione) al primo uso del parametro verrebbe generato lo stato di errore.

Per gestire questa situazione sono stati introdotti i campi della `STEntry` `isInitAfterDec` e `initPars`. Il primo è un flag associato alle variabili, che viene settato a true ogni volta che questa passa da `declared` a `init`. Il secondo invece è un array di booleani associato alle funzioni. Al momento in cui viene processata la dichiarazione di funzione si verifica se qualcuno dei suoi parametri è `isInitAfterDec` dopo il controllo semantico del corpo. Quei parametri che hanno subito inizializzazione vengono dunque settati a true nell'array `initPars` della funzione. Questo array poi viene controllato dalla `CallNode`, che al momento di passare parametri attuali alla funzione, setta il loro status a `init` se e solo se questi lo sono anche nell'array `initPars`.

Gestione chiamate di funzione: punto fisso

Al momento in cui si performa l'analisi degli effetti sulle dichiarazioni di funzione, occorre determinare a livello statico quali saranno gli effetti (o status) dei parametri formali al termine dell'esecuzione della funzione. Questo diventa critico se si pensa, ad esempio, ad una funzione che contiene al suo interno una chiamata ricorsiva.

Per calcolare correttamente questi status si è reso necessario implementare un controllo che si rifà al metodo del punto fisso. Questo consiste nell'inizializzare gli status dei parametri alla fine dell'esecuzione al valore che hanno al momento della dichiarazione, per poi fare l'analisi degli effetti sul corpo della funzione in modo iterativo. Al termine di ogni analisi si confrontano gli status dei parametri: se sono cambiati, si esegue nuovamente un ciclo, fin tanto che gli status non si stabilizzano sugli stessi valori (i.e. raggiungono il loro punto fisso).

La porzione di codice che esegue il punto fisso si trova all'interno del metodo `checkSemantics()` del nodo `DecFunNode`.

3.4.3 Dichiarazione di Variabili non Utilizzate

Questo controllo viene eseguito all'uscita di ciascun blocco, ovvero alla chiusura dello scope corrente. Pertanto è implementato nei metodi `checkSemantics()` dei nodi `BlockNode`, `ProgramNode` e `DecFunNode`.

In ogni caso solleva un warning che avvisa l'utente del mancato utilizzo delle variabili.

3.5 Esempi dalle specifiche

3.5.1 Esempio 1

Il seguente esempio è contenuto nel file `example11.simplan`.

```
{
    int a;
    int b;
    int c = 1 ;
    if (c > 1) {
        b = c ;
    } else {
        a = b ;
    }
}
```

```
Parsing...
Parse completed without issues.
Semantic Analysis...
Effect analysis error:
Error in the effect analysis:
b was used before initialization.
```

L'esecuzione termina, come ci si aspetta, con un errore. Questo dipende dal fatto che nel ramo else, viene usata nell'assegnamento una variabile non inizializzata. Ci aspettiamo che l'analisi degli effetti rilevi e segnali come erroneo questo comportamento.

3.5.2 Esempio 2

Il seguente esempio è contenuto nel file example12.simplan.

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

```
Parsing...
Syntactic analysis:
There are errors in the file.
Look at the errors.txt file.
```

L'esempio contiene un'errore lessicale. A causa della presenza della parentesi graffa dopo l'assegnamento del corpo della funzione, le invocazioni di funzione vengono lette come codice al di fuori del blocco principale. Dal momento che per come è definita la grammatica non ammette la presenza di codice dopo la chiusura del blocco principale, restituisce un errore sintattico:

```
Syntactic errors:
line 12:4 mismatched input 'f' expecting <EOF>
    f(1,a) ; f(2,b) ; f(3,c) ;
    ^
```

Se si corregge l'esito è diverso.

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

```
Parsing...
Parse completed without issues.
Semantic Analysis...
Warnings:
Variable x was declared but never used.
Variable a was declared but never used.
Variable b was declared but never used.
Variable c was declared but never used.
Semantic analysis (hopefully) completed.
Type checking...
Type checking completed with success!
Assembling...
Program output:
```

L'esecuzione termina con successo, senza output prodotti. Il compilatore mostra dei warning relativi alle variabili dichiarate ma non utilizzate.

3.5.3 Esempio 3

Il seguente esempio è contenuto nel file example13.simplan.

```
{
    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) ;
}
```

```
Parsing...
Parse completed without issues.
Semantic Analysis...
Effect analysis error:
Error in the effect analysis:
a was used before initialization.
```

Il programma termina sollevando un'errore, rilevato dall'analisi degli effetti. L'errore è dovuto dai fatto che, sin dalla prima chiamata di funzione, si tenta di effettuare un assegnamento usando una variabile non inizializzata. Di fatto però l'errore non viene individuato per b, ma per a. Questo dipende dal fatto che l'algoritmo del punto fisso cicla, individuando come status finali quelli dove sia a che b sono in errore. Dal controllo fatto in uscita dal blocco, viene poi individuato per primo quello di a.

Capitolo 4

Code Generation e Interprete

SVM è il linguaggio bytecode che è stato definito per eseguire i programmi SimpLanPlus.

Il linguaggio, simil MIPS, è formato da una serie di istruzioni per una macchina a pila, che abbiamo supposto sia a celle di 4 byte (32 bit).

I dati in memoria, che nel nostro caso sono sia interi che booleani, hanno una rappresentazione uniforme, che corrisponde a 4 byte. I valori true e false, dunque, corrispondono rispettivamente agli interi 1 e 0.

4.1 Grammatica del bytecode

Per definire il linguaggio bytecode è stata creata una grammatica nel formato .g4, la quale specifica le istruzioni supportate dalla macchina a pila.

Ogni istruzione è definita come una regola all'interno della grammatica, e include l'opcode e gli eventuali parametri richiesti.

Le istruzioni sono strutturate in questo modo:

- l'**operazione**, una stringa che rappresenta l'operazione che verrà svolta sui valori presenti negli argomenti, per esempio: `add` che applica una somma fra registri, `sw` che scrive il valore di un registro in memoria, etc...;
- il **primo argomento**, una stringa che potrà essere un registro o un label;
- l'**offset**, un numero utilizzato per l'indirizzamento in memoria;
- il **secondo argomento**, sempre una stringa che potrà essere un registro o un label;

- il **terzo argomento**, sempre una stringa che potrà essere un registro o un label;
- l'**argomento intero**, un intero utilizzato per rappresentare valori primitivi utilizzati nelle operazioni.

Di seguito, è riportato un elenco delle istruzioni supportate dalla grammatica:

- **push**: sposta il valore dal registro passato come parametro sulla pila;
- **pop**: rimuove il valore in cima alla pila;
- **add**: somma i valori dei registri passati come secondo e terzo parametro e inserisce il risultato nel registro passato come primo parametro;
- **addi**: somma il valore passato come terzo parametro a quello del registro passato come secondo parametro e inserisce il risultato nel registro passato come primo parametro;
- **sub**: sottrae i valori dei registri passati come secondo e terzo parametro e inserisce il risultato nel registro passato come primo parametro;
- **subi**: sottrae il valore passato come terzo parametro a quello del registro passato come secondo parametro e inserisce il risultato nel registro passato come primo parametro;
- **mult**: moltiplica i valori dei registri passati come secondo e terzo parametro e inserisce il risultato nel registro passato come primo parametro;
- **multi**: moltiplica il valore passato come terzo parametro a quello del registro passato come secondo parametro e inserisce il risultato nel registro passato come primo parametro;
- **div**: divide i valori dei registri passati come secondo e terzo parametro e inserisce il risultato nel registro passato come primo parametro;
- **divi**: divide il valore passato come terzo parametro a quello del registro passato come secondo parametro e inserisce il risultato nel registro passato come primo parametro;
- **and**: esegue un'operazione logica AND sui valori dei registri passati come secondo e terzo parametro e inserisce il risultato nel registro passato come primo parametro;

- **or**: esegue un'operazione logica OR sui valori dei registri passati come secondo e terzo parametro e inserisce il risultato nel registro passato come primo parametro;
- **not**: esegue un'operazione logica NOT sul valore del registro passato come secondo parametro e inserisce il risultato nel registro passato come primo parametro;
- **lw**: carica il valore dalla memoria all'indirizzo formato dalla somma fra il registro passato come terzo parametro e l'offset passato come secondo e lo inserisce nel registro passato come primo parametro;
- **li**: carica il valore nel secondo parametro nel registro passato come primo parametro;
- **sw**: scrive il valore del registro passato come primo parametro nella memoria all'indirizzo formato dalla somma fra il registro passato come terzo parametro e l'offset passato come secondo;
- **mv**: copia il valore del registro passato come secondo parametro nel registro passato come primo;
- **b**: esegue un salto incondizionato all'etichetta passata come parametro;
- **beq**: esegue un salto all'etichetta passata come primo parametro se i valori dei registri nel secondo e terzo parametro sono uguali;
- **bleq**: esegue un salto all'etichetta passata come primo parametro se il valore del registro passato come secondo parametro è minore o uguale a quello del registro passato come primo;
- **jal**: esegue un salto all'etichetta passata come parametro e salva nel registro $ra l'indirizzo dell'istruzione sequenzialmente dopo la istruzione jal;
- **jr**: esegue un salto all'indirizzo contenuto nel registro passato come parametro;
- **label**: definisce un'etichetta;
- **halt**: ferma l'esecuzione del programma;
- **print**: stampa il valore del registro passato come parametro.

4.2 Code Generation

Le istruzioni SVM sono generate dalle funzioni `codeGeneration()` implementate da tutti nodi dell'AST al momento della compilazione.

Il risultato finale dell'esecuzione di queste funzioni è una stringa che rappresenta il codice assembly del programma che si è compilato. Questa stringa, al momento dell'esecuzione del programma, viene passata al Lexer e, in seguito, al Parser.

Come nel caso del SimpLanPlus, sono state implementate le funzioni `visit` per ogni elemento del linguaggio bytecode, che generano una lista di istruzioni.

Per gestire in modo più efficiente le istruzioni, abbiamo creato una classe apposita chiamata `InstructionNode`.

4.3 Esecuzione con Interprete

La lista delle istruzioni generata dal codice assembly viene passata all'interprete `SVMInterpreter`, che le eseguirà con le rispettive implementazioni utilizzando:

- una serie di **registri**, tra quelli più importanti:
  - $sp, lo stack pointer: contiene sempre l'indirizzo della cella in cima alla pila, viene inizializzato con la grandezza della memoria `MEMSIZE`;
  - $fp, il frame pointer: contiene l'indirizzo del record di attivazione corrente, viene inizializzato con la grandezza della memoria `MEMSIZE`;
  - $ra, il return address: nei record di attivazione delle funzioni viene settato all'indirizzo dell'istruzione successiva alla chiamata di funzione;
- la memoria a pila, inizializzata a `MEMSIZE`.

4.3.1 Gestione della pila

Con un ciclo che aggiorna l'**$ip** (instruction pointer) per scorrere la lista delle istruzioni, nella memoria a pila si vanno a creare dei record di attivazione che indicano l'ingresso in uno scope. Nella figura 4.1 viene mostrata la struttura che abbiamo utilizzato per gli AR.

I record di attivazione generici iniziano, partendo dal basso, con lo spazio delle variabili dichiarate nello scope a cui appartiene l'AR.

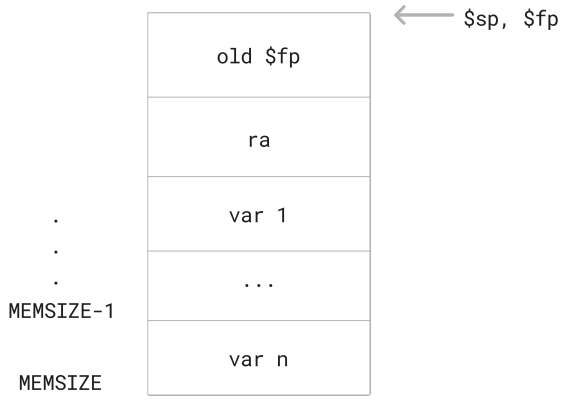

Figura 4.1: Struttura pila.

Queste sono seguite da due celle utilizzate quando c'è la necessità di scorrere la pila:

- l'**old $fp**, che memorizza l'indirizzo del frame pointer precedente al push del nuovo record;
- il **return address**, che memorizza l'indirizzo dell'istruzione successiva alla chiamata di funzione, inizializzato a 0 per i blocchi generici.

Invece, nel caso di RA di scope di funzioni (esempio in figura 4.2), le dichiarazioni di variabili sono seguite dagli argomenti della funzione.

Variabili passate per riferimento

La grammatica di SimpLanPlus permette di passare gli argomenti per riferimento grazie alla keyword `var`.

Nella pila questo tipo di argomenti occuperanno due celle: una per il valore e una che tiene traccia dell'indirizzo dove si trova il valore della dichiarazione originale. Sempre a figura 4.2 è possibile vederne un esempio.

Al momento del pop del record della funzione verrà aggiornato il valore originale a quell'indirizzo con il valore aggiornato dalle istruzioni del blocco della funzione.

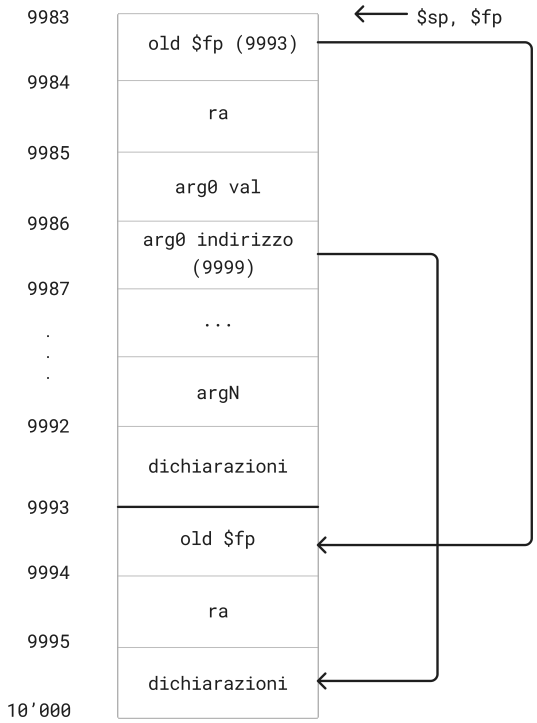

Figura 4.2: Pila con chiamata di funzione.

4.4 Esempi dalle specifiche

4.4.1 Esempio 1

Il seguente esempio è contenuto nel file `example17.simplan`.

```
{
    int x = 1;
    void f(int n) {
        if (n == 0) {
            print(x) ;
        } else {
            x = x * n ;
            f(n-1) ;
        }
    }
```

```
    f(10) ;
}
```

Questa versione del primo esempio dato nelle specifiche del progetto genera un errore semantico, in quanto il corpo della funzione cerca di utilizzare una variabile dichiarata fuori dal suo scope.

Di seguito l'errore generato.

```
Semantic analysis error:
Function f cannot access variable x because
it was declared outside function scope.
```

Se correggiamo l'esempio passando la variabile `x` per riferimento come argomento della funzione, l'esecuzione andrà a buon fine. Di seguito l'esempio corretto e il risultato della sua esecuzione.

```
{
    int a = 1;
    void f(int n, var int x) {
        if (n == 0) {
            print(x) ;
        } else {
            x = x * n ;
            f(n-1, x) ;
        }
    }
    f(10, a) ;
}
```

```
Parsing...
Parse completed without issues.
Semantic Analysis...
Semantic analysis completed.
Type checking...
Type checking completed with success!
Assembling...
Program output:
Printing: 3628800
```

4.4.2 Esempio 2

Il seguente esempio è contenuto nel file `example18.simplan`.

```
{
    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) ;
}
```

```
Parsing...
Parse completed without issues.
Semantic Analysis...
Warnings:
Variable y was declared but never used.
Semantic analysis completed.
Type checking...
Type checking completed with success!
Assembling...
Program output:
Printing: 720
```

4.4.3 Esempio 3

Il seguente esempio è contenuto nel file example19.simplan.

```
{
    void f(int m, int n){
    if (m>n) {
        print(m+n) ;
    } else {
        int x = 1 ; f(m+1,n+1) ;
    }
    }
    f(5,4) ;
}
```

```
Parsing...
Parse completed without issues.
Semantic Analysis...
Warnings:
Variable x was declared but never used.
Variable x was declared but never used.
Semantic analysis completed.
Type checking...
Type checking completed with success!
Assembling...
Program output:
Printing: 9
```

Le specifiche chiedevano di testare lo stesso codice con i parametri della chiamata di funzione invertiti, di seguito il risultato dell'esecuzione con questo cambiamento:

```
Parsing...
Parse completed without issues.
Semantic Analysis...
Warnings:
Variable x was declared but never used.
Variable x was declared but never used.
Semantic analysis completed.
Type checking...
Type checking completed with success!
Assembling...
Program output:
Error: Reached maximum memory.
```

Il programma genera un loop infinito, il controllo sull'uso della memoria del nostro interprete avvisa l'utente che è stato raggiunto il limite massimo di memoria.

Appendice A

Regole di Inferenza

A.1 Scope Checking

Expressions

Formato dei giudizi: $\Gamma \vdash$ exp: void

$$\frac{\texttt{ids(e)} \in \texttt{dom}(\Gamma)}{\Gamma \vdash \texttt{e} :}[\texttt{Exp-s}]$$

$$\frac{\Gamma \vdash \texttt{e}_1 : \quad \Gamma \vdash \texttt{e}_2 :}{\Gamma \vdash \texttt{e}_1 \ \texttt{OP} \ \texttt{e}_2 :}[\texttt{BinExp-s}]$$

$$\texttt{OP} : \ + \mid - \mid * \mid / \mid < \mid \leq \mid > \mid \geq \mid \&\& \mid \mid\mid \mid == \mid \neq$$

$$\frac{\Gamma \vdash \texttt{f}(\texttt{e}_1, \ ..., \ \texttt{e}_\texttt{n}) :}{\Gamma \vdash \texttt{f}(\texttt{e}_1, \ ..., \ \texttt{e}_\texttt{n}) :}[\texttt{CallExp-s}]$$

Statements

Formato dei giudizi: $\Gamma \vdash$ stm: void

$$\frac{\Gamma \vdash \texttt{s} : \quad \Gamma' \vdash \texttt{S} :}{\Gamma \vdash \texttt{s S} :}[\texttt{Seq-s}]$$

$$\frac{\texttt{x} \in \texttt{dom(top}(\Gamma)\texttt{)} \qquad \Gamma \vdash \texttt{e:}}{\Gamma \vdash \texttt{x=e;:}}[\texttt{Asgn-s}] \qquad \frac{\Gamma \vdash \texttt{e:}}{\Gamma \vdash \texttt{print e;:}}[\texttt{Print-s}]$$

$$\frac{}{\Gamma \vdash \texttt{return;} :}\,[\texttt{Ret1-s}] \qquad \frac{\Gamma \vdash \texttt{e} :}{\Gamma \vdash \texttt{return e;} :}\,[\texttt{Ret2-s}]$$

$$\frac{\Gamma \vdash \texttt{e} : \quad \Gamma \vdash \texttt{s} :}{\Gamma \vdash \texttt{if (e) s} \ :}\,[\texttt{Ite1-s}]$$

$$\frac{\Gamma \vdash \texttt{e} : \quad \Gamma \vdash \texttt{s}_1 : \quad \Gamma \vdash \texttt{s}_2 :}{\Gamma \vdash \texttt{if (e) s}_1 \texttt{ else s}_2 :}\,[\texttt{Ite2-s}]$$

$$\frac{\texttt{f} \in \texttt{dom}(\Gamma) \quad (\Gamma \vdash \texttt{e}_\texttt{i} :)_{1 \leq \texttt{i} \leq \texttt{n}}}{\Gamma \vdash \texttt{f(e}_1, \ \ldots, \ \texttt{e}_\texttt{n}) :}\,[\texttt{Call-s}]$$

Declarations

Formato dei giudizi: $\Gamma, n \vdash$ dec: Γ', n'

$$\frac{\Gamma \vdash \texttt{d} : \Gamma' \quad \Gamma' \vdash \texttt{D} : \Gamma''}{\Gamma \vdash \texttt{d D} : \Gamma''}\,[\texttt{DSeq-s}]$$

$$\frac{\texttt{x} \notin \texttt{dom}(\texttt{top}(\Gamma))}{\Gamma, \texttt{n} \vdash \texttt{T x;} : \Gamma[\texttt{x} \mapsto \texttt{T}, \texttt{n}], \texttt{n} + 1}\,[\texttt{DecVar-s}]$$

$$\frac{\texttt{x} \notin \texttt{dom}(\texttt{top}(\Gamma)) \quad \Gamma \vdash \texttt{e} :}{\Gamma, \texttt{n} \vdash \texttt{T x} = \texttt{e;} : \Gamma[\texttt{x} \mapsto \texttt{T}, \texttt{n}], \texttt{n} + 1}\,[\texttt{DecVarI-s}]$$

$$\frac{\texttt{f} \notin \texttt{dom}(\texttt{top}(\Gamma)) \quad \Gamma[\texttt{x}_1 \mapsto \texttt{T}_1, ..., \texttt{y}_\texttt{m} \mapsto \texttt{T}_\texttt{m}] \vdash \texttt{b} : \Gamma'}{\Gamma, \texttt{n} \vdash \texttt{T f(var T}_1 \ \texttt{x}_1, ..., \texttt{T}'_\texttt{m} \ \texttt{y}_\texttt{m}) = \texttt{b;} : \Gamma'[\texttt{f} \mapsto (\texttt{T}_1 \times ... \times \texttt{T}'_\texttt{m}) \to \texttt{T}, \texttt{n}], \texttt{n} + 1}\,[\texttt{DecFun-s}]$$

Altre

Formato dei giudizi: $\Gamma \vdash$ block: void

$$\frac{\Gamma, 0 \bullet [] \vdash \texttt{D} : \Gamma', \texttt{n} \quad \Gamma' \vdash \texttt{S} :}{\Gamma \vdash \{\texttt{D S}\} :}\,[\texttt{Block-s}]$$

Formato dei giudizi: $\Gamma \vdash$ program: void

$$\frac{\varnothing, 0 \bullet [] \vdash \texttt{D} : \Gamma, \texttt{n} \quad \Gamma \vdash \texttt{S} :}{\varnothing \vdash \{\texttt{D S}\} : \texttt{n}}\,[\texttt{Prog-s}]$$

A.2 Type Checking

Expressions

Formato dei giudizi: $\Gamma \vdash \text{exp}: \mathsf{T}$

$$\frac{\Gamma(\mathtt{x})=\mathsf{T}}{\Gamma \vdash \mathtt{x}:\mathsf{T}}[\mathtt{Var-t}] \quad \frac{}{\Gamma \vdash \mathtt{n}:\mathtt{int}}[\mathtt{Val-t}] \quad \frac{}{\Gamma \vdash \mathtt{b}:\mathtt{bool}}[\mathtt{Bool-t}]$$

$$\frac{\Gamma \vdash \mathtt{e}:\mathtt{int}}{\Gamma \vdash -\mathtt{e}:\mathtt{int}}[\mathtt{Neg-t}] \quad \frac{\Gamma \vdash \mathtt{e}:\mathtt{bool}}{\Gamma \vdash -\mathtt{e}:\mathtt{bool}}[\mathtt{Not-t}]$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathtt{e}_2 : \mathsf{T}_2 \quad \mathsf{T}_1 = \mathtt{int} = \mathsf{T}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{BIN\_OP\_INT1}\ \mathtt{e}_2 : \mathtt{int}}[\mathtt{BinInt-t}]$$

$$\mathtt{BIN\_OP\_INT1}: \quad + \mid - \mid * \mid /$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathtt{e}_2 : \mathsf{T}_2 \quad \mathsf{T}_1 = \mathtt{int} = \mathsf{T}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{BIN\_OP\_INT2}\ \mathtt{e}_2 : \mathtt{bool}}[\mathtt{BinInt2-t}]$$

$$\mathtt{BIN\_OP\_INT2}: \quad < \mid \leq \mid > \mid \geq$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathtt{e}_2 : \mathsf{T}_2 \quad \mathsf{T}_1 = \mathtt{bool} = \mathsf{T}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{BIN\_OP\_BOOL}\ \mathtt{e}_2 : \mathtt{bool}}[\mathtt{BinBool-t}]$$

$$\mathtt{BIN\_OP\_BOOL}: \quad \&\& \mid ||$$

$$\frac{\Gamma \vdash \mathtt{e}_1 : \mathsf{T}_1 \quad \Gamma \vdash \mathtt{e}_2 : \mathsf{T}_2 \quad \mathsf{T}_1 = \mathsf{T}_2}{\Gamma \vdash \mathtt{e}_1\ \mathtt{BIN\_OP}\ \mathtt{e}_2 : \mathtt{bool}}[\mathtt{Bin-t}]$$

$$\mathtt{BIN\_OP}: \quad == \mid \neq$$

$$\frac{\Gamma \vdash \mathtt{f}(\mathsf{T}'_1\ \mathtt{x}_1,\ \ldots,\ \mathsf{T}'_n\ \mathtt{x}_n) : \mathsf{T}}{\Gamma \vdash \mathtt{f}(\mathsf{T}'_1\ \mathtt{x}_1,\ \ldots,\ \mathsf{T}'_n\ \mathtt{x}_n) : \mathsf{T}}[\mathtt{CallExp-t}]$$

Statements

Formato dei giudizi: $\Gamma \vdash \text{stm}: \mathtt{T}$

$$\frac{\Gamma \vdash \mathtt{s} : \mathtt{T} \quad \Gamma' \vdash \mathtt{S} : \mathtt{T}'}{\Gamma \vdash \mathtt{s\ S} : \mathtt{T}'}[\mathtt{Seq-t}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{T} \\ \Gamma(\mathtt{x}) = \mathtt{T}' \quad \mathtt{T} = \mathtt{T}'\end{array}}{\Gamma \vdash \mathtt{x} = \mathtt{e};: \mathtt{void}}[\mathtt{Asgn-t}]$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{T}}{\Gamma \vdash \mathtt{print\ e};: \mathtt{void}}[\mathtt{Print-t}]$$

$$\frac{}{\Gamma\vdash \mathtt{return};:\mathtt{void}}[\mathtt{Ret1-t}] \quad \frac{\Gamma\vdash \mathtt{e}:\mathtt{T}}{\Gamma\vdash \mathtt{return\ e};:\mathtt{T}}[\mathtt{Ret2-t}]$$

$$\frac{\Gamma \vdash \mathtt{e} : \mathtt{bool} \quad \Gamma \vdash \mathtt{s} : \mathtt{T}}{\Gamma \vdash \mathtt{if\ (e)\ s\ } : \mathtt{T}}[\mathtt{Ite1-t}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathtt{e} : \mathtt{bool} \\ \Gamma \vdash \mathtt{s}_1 : \mathtt{T}_1 \quad \Gamma \vdash \mathtt{s}_2 : \mathtt{T}_2 \quad \mathtt{T}_1 = \mathtt{T}_2\end{array}}{\Gamma \vdash \mathtt{if\ (e)\ s}_1 \mathtt{\ else\ s}_2 : \mathtt{T}_2}[\mathtt{Ite2-t}]$$

$$\frac{\Gamma(\mathtt{f}) = \mathtt{T}_1 \times ... \times \mathtt{T}_n \to \mathtt{T} \quad (\mathtt{T}_i = \mathtt{T}'_i)_{1\leq i\leq n}}{\Gamma \vdash \mathtt{f}(\mathtt{T}'_1\ \mathtt{x}_1,\ ...,\ \mathtt{T}'_n\ \mathtt{x}_n) : \mathtt{T}}[\mathtt{Call-t}]$$

Declarations

Formato dei giudizi: $\Gamma \vdash \text{dec}: \mathtt{T}$

$$\frac{\Gamma \vdash \mathtt{d} : \mathtt{T} \quad \Gamma \vdash \mathtt{D} : \mathtt{T}'}{\Gamma \vdash \mathtt{d\ D} : \mathtt{T}'}[\mathtt{DSeq-t}]$$

$$\frac{}{\Gamma\vdash \mathtt{T\ x};:\mathtt{T}}[\mathtt{DecVar-t}] \quad \frac{\Gamma\vdash \mathtt{e}:\mathtt{T}' \qquad \mathtt{T}=\mathtt{T}'}{\Gamma\vdash \mathtt{T\ x=e};:\mathtt{T}}[\mathtt{DecVarI-t}]$$

$$\frac{\Gamma \vdash \mathtt{b} : \mathtt{T}'' \quad \mathtt{T} = \mathtt{T}''}{\Gamma \vdash \mathtt{T\ f(var\ T}_1\ \mathtt{x}_1, ..., \mathtt{T}'_m\ \mathtt{y}_m) = \mathtt{b};: \mathtt{T}}[\mathtt{DecFun-t}]$$

Altre

Formato dei giudizi: $\Gamma \vdash$ block: T

$$\frac{\Gamma, 0 \bullet [] \vdash \mathtt{D} : \quad \Gamma' \vdash \mathtt{S} : \mathtt{T} \quad \mathtt{S.equalTypesToReturn()}}{\Gamma \vdash \{\mathtt{D\ S}\} : \mathtt{T}}\,[\mathtt{Block-t}]$$

Formato dei giudizi: $\Gamma \vdash$ program: void

$$\frac{\varnothing, 0 \bullet [] \vdash \mathtt{D} : \Gamma', \mathtt{n} \quad \Gamma' \vdash \mathtt{S} : \mathtt{T} \quad \mathtt{S.equalTypesToReturn()}}{\varnothing \vdash \{\mathtt{D\ S}\} : \mathtt{T}}\,[\mathtt{Prog-t}]$$

Dove il metodo equalTypesToReturn() verifica che gli statements che compongono S, se multipli, ritornino tutti lo stesso tipo.

A.3 Effect Analysis

Expressions

Formato dei giudizi: $\Sigma \vdash$ exp: Σ'

$$\frac{\mathtt{ids(e)} = \mathtt{x_1}, ..., \mathtt{x_n}}{\Sigma \vdash \mathtt{e} : \Sigma \triangleright [\mathtt{x_1} \mapsto \mathtt{used}, ..., \mathtt{x_n} \mapsto \mathtt{used}]}\,[\mathtt{Exp-e}]$$

$$\frac{\Sigma \vdash \mathtt{e_1} : \Sigma'' \quad \Sigma'' \vdash \mathtt{e_2} : \Sigma'}{\Sigma \vdash \mathtt{e_1\ OP\ e_2} : \Sigma'}\,[\mathtt{BinExp-e}]$$

$$\mathtt{OP}: \ + \mid - \mid * \mid / \mid < \mid \leq \mid > \mid \geq \mid \&\& \mid \mid\mid \mid == \mid \neq$$

$$\frac{\Sigma \vdash \mathtt{f(u_1, ..., u_n, e_1, ..., e_m);} : \Sigma'}{\Sigma \vdash \mathtt{f(u_1, ..., u_n, e_1, ..., e_m);} : \Sigma'}\,[\mathtt{CallExp-e}]$$

Statements

Formato dei giudizi: $\Sigma \vdash$ stm: Σ'

$$\frac{\Sigma \vdash \mathtt{s} : \Sigma' \quad \Sigma' \vdash \mathtt{S} : \Sigma''}{\Sigma \vdash \mathtt{s\ S} : \Sigma''}\,[\mathtt{Seq-e}]$$

$$\frac{\Sigma \vdash \mathtt{e} : \Sigma'}{\Sigma \vdash \mathtt{x=e;} :: \Sigma' \triangleright [\mathtt{x} \mapsto \mathtt{init}]}\,[\mathtt{Asgn-e}] \qquad \frac{\Sigma \vdash \mathtt{e} : \Sigma'}{\Sigma \vdash \mathtt{print\ e;} :: \Sigma'}\,[\mathtt{Print-e}]$$

$$\frac{}{\Sigma \vdash \texttt{return;} :: \Sigma}\,[\texttt{Ret1-e}] \qquad \frac{\Sigma \vdash \texttt{e} : \Sigma'}{\Sigma \vdash \texttt{return e;} :: \Sigma'}\,[\texttt{Ret2-e}]$$

$$\frac{\Sigma \vdash \texttt{e} : \Sigma' \qquad \Sigma' \vdash \texttt{s} : \Sigma''}{\Sigma \vdash \texttt{if (e) s} \; : \Sigma''}\,[\texttt{Ite1-e}]$$

$$\frac{\Sigma \vdash \texttt{e} : \Sigma' \quad \Sigma' \vdash \texttt{s}_1 : \Sigma_1 \quad \Sigma' \vdash \texttt{s}_2 : \Sigma_2}{\Sigma \vdash \texttt{if (e) s}_1 \texttt{ else s}_2 : \texttt{BIN}(\Sigma_1, \Sigma_2)}\,[\texttt{Ite2-e}]$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{f} : \&\texttt{T}_1 \times ... \times \&\texttt{T}_\texttt{n} \times \texttt{T}_1 \times ... \times \texttt{T}_\texttt{m} \to \texttt{T} \\ \Sigma(\texttt{f}) = (\Sigma_0 \to \Sigma_1, \texttt{initPars}) \\ (\Sigma_1(\texttt{y}_\texttt{j}) \leq \texttt{used})_{1 \leq \texttt{j} \leq \texttt{m}} \qquad \Sigma[(\texttt{u}_\texttt{i} \mapsto \Sigma(\texttt{u}_\texttt{i}) \triangleright \texttt{init})_{\texttt{initPars(i)==true}}] \\ \Sigma' = \Sigma[(\texttt{z}_\texttt{j} \mapsto \Sigma(\texttt{z}_\texttt{j}) \triangleright \texttt{used})_{\texttt{z}_\texttt{j} \in \texttt{ids}(\texttt{e}_1,...,\texttt{e}_\texttt{m})}] \qquad \Sigma'' = \otimes_{\texttt{i}=1,...,\texttt{n}}[\texttt{u}_\texttt{i} \mapsto \Sigma(\texttt{u}_\texttt{i}) \triangleright \Sigma_1(\texttt{x}_\texttt{i})]\end{array}}{\Sigma \vdash \texttt{f}(\texttt{u}_1, ..., \texttt{u}_\texttt{n}, \texttt{e}_1, ..., \texttt{e}_\texttt{m}); : \texttt{update}(\Sigma', \Sigma'')}\,[\texttt{Call-e}]$$

Declarations

Formato dei giudizi: $\Sigma \vdash$ dec: Σ'

$$\frac{\Sigma \vdash \texttt{d} : \Sigma' \quad \Sigma' \vdash \texttt{D} : \Sigma''}{\Sigma \vdash \texttt{d D} : \Sigma''}\,[\texttt{DSeq-e}]$$

$$\frac{}{\Sigma \vdash \texttt{T x;} :: \Sigma[\texttt{x} \mapsto \bot]}\,[\texttt{DecVar-e}] \qquad \frac{\Sigma \vdash \texttt{e} : \Sigma'}{\Sigma \vdash \texttt{T x=e;} :: \Sigma'[\texttt{x} \mapsto \texttt{init}]}\,[\texttt{DecVarI-e}]$$

$$\frac{\begin{array}{c}\Sigma_0 = [\texttt{x}_1 \mapsto \bot, ..., \texttt{x}_\texttt{n} \mapsto \bot, \texttt{y}_1 \mapsto \bot, ..., \texttt{y}_\texttt{m} \mapsto \bot] \\ \Sigma|_{\texttt{FUN}} \bullet \Sigma_0[\texttt{f} \mapsto \Sigma_0 \to \Sigma_1] \vdash \texttt{s} : \Sigma|_{\texttt{FUN}} \bullet \Sigma_1[\texttt{f} \mapsto \Sigma_0 \to \Sigma_1], \texttt{initPars}\end{array}}{\Sigma \vdash \texttt{f}(\texttt{var x}_1, ..., \texttt{var x}_\texttt{n}, \texttt{y}_1, ..., \texttt{y}_\texttt{m}) \texttt{ s;} : \Sigma[\texttt{f} \mapsto (\Sigma_0 \to \Sigma_1, \texttt{initPars})]}\,[\texttt{DecFun-e}]$$

Altre

Formato dei giudizi: $\Sigma \vdash$ block: Σ'

$$\frac{\begin{array}{c}\Sigma \bullet [] \vdash \texttt{D} : \Sigma' \quad \Sigma' \vdash \texttt{S} : \Sigma'' \\ \Sigma'' = \Sigma''_0 \bullet \Sigma''_1 \quad (\Sigma''_1(\texttt{x}_\texttt{i}) \leq \texttt{used})^{\texttt{x}_\texttt{i} \in \texttt{dom}(\Sigma''_1)}\end{array}}{\Sigma \vdash \{\texttt{D S}\} : \Sigma''_0}\,[\texttt{Block-e}]$$

Formato dei giudizi: $\Sigma \vdash$ program: Σ'

$$\frac{\begin{array}{c}\varnothing \bullet [] \vdash \texttt{D} : \Sigma \quad \Sigma \vdash \texttt{S} : \Sigma' \\ \Sigma' = \Sigma'_0 \bullet \Sigma'_1 \quad (\Sigma'_1(\texttt{x}_\texttt{i}) \leq \texttt{used})^{\texttt{x}_\texttt{i} \in \texttt{dom}(\Sigma'_1)}\end{array}}{\varnothing \vdash \{\texttt{D S}\} : \Sigma'_0}\,[\texttt{Prog-e}]$$

Elenco delle figure