

Alma Mater Studiorum - University of Bologna  
LM Informatica  
**SimplanPlus Compiler**  
Project of the course “Compilatori & Interpreti”

Riccardo Preite, Davide Davoli, Matteo Mele, Leonardo Pio Palumbo

A.Y. 2020/2021

**Abstract**

Questo progetto presenta la realizzazione di un compilatore e un interprete per il linguaggio **SimplanPlus**, ideato nell’ambito del corso di Compilatori e interpreti della LM Informatica A.A. 2020/21.

Più nello specifico, il linguaggio adotta un paradigma imperativo, è staticamente tipato, permette la definizione di funzioni, anche ricorsive, ma non mutuamente ricorsive e la gestione di tipi puntatori. Nelle varie fasi del progetto, descritte accuratamente nelle sezioni successive, è stata analizzata la grammatica del linguaggio e implementati svariati controlli su diversi livelli per garantire la correttezza sintattica e semantica dei programmi, con particolare attenzione all’analisi degli effetti e al controllo dei tipi. Successivamente, è stato definito un linguaggio intermedio **SVM-Assembly** che viene preso in input da un interprete. Quest’ultimo simula, tramite una memoria virtuale, il funzionamento di una macchina a pila in cui è permesso l’utilizzo di registri.

## Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
1.1	Obiettivo . . . . .	3
1.2	Il Linguaggio - SimplanPlus . . . . .	3
1.3	Compilatore . . . . .	3
1.4	Interprete . . . . .	3
1.5	Stato dell’arte . . . . .	3
<b>2</b>	<b>Grammatica</b>	<b>4</b>
2.1	Blocchi . . . . .	4
2.2	Dichiarazioni . . . . .	4
2.3	Istruzioni . . . . .	5
2.4	Espressioni . . . . .	5
2.5	Commenti e caratteri ignorati . . . . .	6
2.6	Errori grammaticali . . . . .	6
<b>3</b>	<b>Semantica</b>	<b>7</b>
3.1	Ambiente . . . . .	7
3.1.1	STentry . . . . .	7
3.1.2	Environment . . . . .	7
3.2	Errori semantici . . . . .	7
3.2.1	Variabili/Funzioni non dichiarate . . . . .	7
3.2.2	Variabili dichiarate più volte nello stesso ambiente . . . . .	8
<b>4</b>	<b>Controllo dei tipi</b>	<b>9</b>
4.1	Tipi . . . . .	9
4.1.1	Corretto uso dei puntatori . . . . .	9
4.1.2	Parametri attuali non conformi ai parametri formali . . . . .	9
<b>5</b>	<b>Analisi degli effetti</b>	<b>11</b>
5.1	Struttura ambiente . . . . .	11
5.2	Effetti . . . . .	11
5.3	Limiti . . . . .	12
<b>6</b>	<b>Compilatore</b>	<b>14</b>
6.1	Compilatore SimplanPlus . . . . .	14
<b>7</b>	<b>Interprete</b>	<b>15</b>
7.1	Interprete SimplanPlus . . . . .	15
7.1.1	Bytecode . . . . .	15
7.1.2	CPU e memoria . . . . .	15
7.1.3	Record di attivazione . . . . .	16
7.1.4	Come funziona l’interprete? . . . . .	17
<b>8</b>	<b>Esempi</b>	<b>18</b>
8.1	Esempio 1 . . . . .	18

# 1 Introduzione

## 1.1 Obiettivo

Obiettivo di questo progetto è la realizzazione di un compilatore e un interprete per uno specifico linguaggio denominato **simplanplus**, le cui specifiche sono descritte nei successivi capitoli.

## 1.2 Il Linguaggio - SimplanPlus

Le caratteristiche principali del linguaggio **SimplanPlus** sono riportate di seguito:

- sistema di tipi statico;
- linguaggio imperativo (quindi non Object-Oriented);
- gestione manuale della memoria;
- possibilità di definire funzioni, anche ricorsive, ma non mutualmente ricorsive.

## 1.3 Compilatore

L'esecuzione del compilatore di **SimplanPlus** si articola nei seguenti step, svolti esattamente nell'ordine descritto:

1. **analisi lessicale**, in cui il codice sorgente viene trasformato in *token* che contribuiscono alla creazione dell'**Abstract Syntax Tree (AST)**;
2. **analisi semantica**, in cui viene verificata la correttezza semantica del programma in input (ad esempio, che le variabili vengano utilizzate solo dopo una loro dichiarazione);
3. **type checking**, in cui viene controllato che i tipi all'interno delle espressioni, nelle assegnazioni e nelle istruzioni siano concordi a quanto atteso;
4. **analisi degli effetti**, in cui viene verificato lo stato delle variabili e dei puntatori nella memoria (inizializzato  $\perp$ , letto/scritto *rw*, cancellato *d*, errore  $\top$ ).

Successivamente, si passa alla generazione del codice **SVM-Assembly** corrispondente.

## 1.4 Interprete

L'interprete di **SimplanPlus**, **SVM**, prende in input il codice **SVM-Assembly** generato dal compilatore e illustrato nella precedente sezione, eseguendolo un'istruzione alla volta. Il programma viene eseguito in una “memoria virtuale” (simulata dal codice che implementa l'interprete) la cui dimensione è configurabile.

## 1.5 Stato dell'arte

Le tecnologie principali utilizzate per il progetto sono le seguenti:

- **Java 11**, con cui sono scritti gli interi compilatore e interprete;
- **ANTLR 4.9.2**, per implementare in maniera automatica la parte di analisi lessicale e il parser, sia per **SimplanPlus**, che per il linguaggio **SVM-Assembly**;

## 2 Grammatica

Ogni linguaggio è composto da una grammatica che definisce la struttura delle sue istruzioni, delle espressioni e dei suoi tipi. La grammatica di **SimplanPlus**, presente in `lexer/SimplanPlus.g4`, presenta una serie di regole:

- Block;
- Declaration;
- Statement;
- Expression;
- White space, comment;
- Errors.

Grazie a `lexer/SimplanPlus.g4` il lexer e il parser sono stati generati automaticamente. Ovviamente si è dovuto procedere all’implementazione finale del parser per fare in modo che possa creare i giusti nodi a partire dal contesto che riceve.

### 2.1 Blocchi

Un programma **SimplanPlus** inizia con un block che potrebbe avere una serie di dichiarazioni (variabili o funzioni) seguita da una possibile serie di istruzioni. Il blocco è considerato un’istruzione quindi potrebbe ripetersi più volte all’interno del programma.

```
block      : '{' declaration* statement* '}';
```

### 2.2 Dichiarazioni

L’utente ha la possibilità di dichiarare sia variabili, int o bool o puntatori ad int, bool o altre zone di memoria, sia funzioni di tipo int, bool o void. Durante la dichiarazione di variabili è possibile assegnare un valore tramite un’espressione sul rhs. Le funzioni permettono l’utilizzo di parametri e presentano un blocco nel quale scrivere le dichiarazioni e le istruzioni.

```
declaration : decFun  
            | decVar ;  
  
decFun      : (type | 'void') ID '(' (arg (',' arg)*)? ')' block ;  
  
decVar      : type ID ('=' exp)? ';' ;  
  
type        : 'int'  
            | 'bool'  
            | '^' type ;  
  
arg         : type ID ;
```

## 2.3 Istruzioni

Il linguaggio SimplanPlus presenta una serie di istruzioni che permettono la manipolazione di ID o chiamate di funzioni.

- (assignment) di un valore ad una variabile dichiarata;
- (deletion), viene deallocata la zona di memoria alla quale il puntatore puntava;
- (print), di una espressione (variabile, ritorno di funzione o int o bool);
- (ret) permette ad una funzione di ritornare un valore o di ritornare al chiamante in caso di funzioni void;
- (ite) definisce la costruzione di un blocco if con condizione booleana e ramo else facoltativo;
- (call) invoca la funzione corrispondente con i relativi parametri attuali;
- (block) rappresenta la creazione di un blocco annidato.

```
statement    : assignment ';'
              | deletion ';'
              | print ';'
              | ret ';'
              | ite
              | call ';'
              | block;

assignment   : lhs '=' exp;

lhs          : ID | lhs '^';

deletion     : 'delete' ID;

print        : 'print' exp;

ret          : 'return' (exp)?;

ite          : 'if' '(' exp ')' statement ('else' statement)?;

call         : ID '(' (exp(',' exp)*)? ')';
```

## 2.4 Espressioni

Le espressioni possono essere utilizzate per assegnare valori ad una variabile, in condizioni booleane o anche come argomenti di una funzione.

```
exp          : '(' exp ')' #
baseExp      : '-' exp    #negExp
              | '!' exp    #notExp
              | lhs        #
              | derExp
```

```

| 'new' type                                     #newExp
| left=exp op=('*' | '/')                      right=exp #binExp
| left=exp op=('+' | '-')                      right=exp #binExp
| left=exp op=('<' | '<=' | '>' | '>=')          right=exp #binExp
| left=exp op=('==' | '!=')                    right=exp #binExp
| left=exp op='&&'                             right=exp #binExp
| left=exp op='||'                             right=exp #binExp
| call                                           #callExp
| BOOL                                           #boolExp
| NUMBER                                         #valExp;

//Booleans
BOOL      : 'true' | 'false';

//IDs
fragment CHAR      : 'a'..'z' | 'A'..'Z' ;
ID         : CHAR (CHAR | DIGIT)* ;

//Numbers
fragment DIGIT     : '0'..'9';
NUMBER        : DIGIT+;

```

## 2.5 Commenti e caratteri ignorati

In **SimplanPlus** i commenti possono essere di due tipi: in linea (che iniziano con `//`) e a blocco (compresi tra `/*` e `*/`). Tabulazioni, Spazi bianchi e ritorni a capo sono scartati durante l'analisi lessicale.

```

WS          : (' ' | '\t' | '\n' | '\r')-> skip;
LINECOMMENTS : '//' (~('\n' | '\r'))* -> skip;
BLOCKCOMMENTS : '/*' ( ~('\n' | '\r') | '/' ~ '*' | '*' ~ '/' | BLOCKCOMMENTS)* '*/' -> skip;

```

## 2.6 Errori grammaticali

Gli errori sintattici, presenti al livello di grammatica, sono gestiti grazie a una funzionalità di ANTLR che permette di inserire codice ad alto livello (Java, nel caso di questo progetto) nel file della grammatica, il quale viene poi aggiunto in fase di auto-generazione del codice relativo al lexer e al parser.

```
/* Regole della grammatica descritte sopra */
```

```
ERR      : . { errors.add('Invalid character: ' + getText()); } -> channel(HIDDEN);
```

## 3 Semantica

### 3.1 Ambiente

In questo progetto l'ambiente viene definito come un insieme di scope, gestiti dalla classe **Environment**, che contengono una HashMap di stringhe e STentry corrispondenti alla entry di una variabile nella Symbol table. Quest'ultima è gestita dalla classe **STentry**. Environment e STentry contengono i metodi per gestire sia l'analisi semantica che l'analisi degli effetti.

#### 3.1.1 STentry

Una entry è definita dal **nesting level** in cui compare, il **tipo** che possiede e l'**offset** al quale si trova a partire dal frame pointer. Questo è tutto quello che viene utilizzato di una STentry per l'analisi semantica. Questa classe viene utilizzata dall'Environment per costruire una entry di una variabile all'occorrenza ed aggiungerla al proprio scope.

#### 3.1.2 Environment

Come detto sopra Environment si occupa di manipolare la Symbol Table, rappresentata come un parametro della classe, e possiede inoltre informazioni come il livello di annidamento attuale e l'offset di partenza (differisce nel caso sia una funzione o un blocco semplice). Ogni volta che uno scope viene creato il nesting level aumenta e una nuova, pulita, HashMap viene aggiunta in coda alla lista di Symbol Table accendendovi in maniera LIFO (Last In First Out). Ogni elemento di questo array rappresenta quindi una Symbol Table per quel nesting level dove una Symbol Table a sua volta rappresenta una mappa **String** → **STentry** dove la stringa è l'identificativo della variabile in questione. I principali compiti svolti da questa classe sono quelli di **Push & Pop** di uno scope e la ricerca di una variabile tramite il metodo **lookUp(id)**. Questi metodi vengono utilizzati nella sezione successiva per raccogliere eventuali errori semantici e farli presente all'utente alla fine della compilazione.

### 3.2 Errori semantici

La funzione *checkSemantics(Environment env)* si occupa di controllare la semantica di tutti i nodi dell'albero essendo richiamata ricorsivamente su tutti i figli di un node dove ognuno ritorna una lista, potenzialmente vuota, di errori semantici. Di seguito verranno illustrati alcuni controlli di errori più significativi.

#### 3.2.1 Variabili/Funzioni non dichiarate

Per quanto riguarda il controllo di identificatori non inizializzati basta solamente cercare se l'identificativo in questione possiede una entry nella symbol table dell'environment in questione, a qualsiasi livello.

```

@Override
public ArrayList<SemanticError> checkSemantics(Environment env) {

    //create result list
    ArrayList<SemanticError> res = new ArrayList<SemanticError>();
    entry = env.lookup(id);
    if (entry == null)
        res.add(new SemanticError("Id "+id+" not declared"));
    else
        nestinglevel = env.getNestingLevel();

    return res;
}

```

### 3.2.2 Variabili dichiarate più volte nello stesso ambiente

Analogamente se si vuole controllare che una variabile sia ridichiarata più volte allo stesso livello di annidamento basta creare una entry, e se il valore ritornato dall'aggiunta di quest'ultima sulla HashMap (SymbolTable) è diverso da null significa che esisteva già un valore appartenente a quella chiave e quindi viene aggiunto un errore semantico come di seguito. Viene lasciato solo il codice per l'identificatore di una variabile in quanto quello di una funzione è speculare.

```

@Override
public ArrayList<SemanticError> checkSemantics(Environment env) {
    /**
     * ...
     */
    STentry entry = new STentry(env.getNestingLevel(), type, new_offset);
    id.setEntry(entry);

    if (exp != null){
        res.addAll(exp.checkSemantics(env));
        // dereferenceLevel = 0 because we set the status of a variable
        id.setIdStatus(new Effect(Effect.READWRITE), 0);
    }

    if ( hm.put(id.getID(),entry) != null )
        res.add(new SemanticError("Var id '"+id+"' already declared"));
    /**
     * ...
     */
}

```



## 4 Controllo dei tipi

Il controllo sui tipi viene garantito dal metodo `Node typeCheck()` che restituisce un nodo rappresentante il tipo del nodo (istanza di `Node`) su cui si è invocato, *null* in caso quel nodo non debba avere associato un tipo o un’eccezione in caso di errori durante il processo.

Il valore *null* viene ritornato da tutti i nodi che hanno come supertipo `TypeNode`, dagli argomenti delle funzioni, dalle dichiarazioni di variabili e di funzioni.

Il tipo della maggior parte degli statements non è necessario per il controllo di tipi, dunque spesso viene impostato a `void`.

### 4.1 Tipi

I tipi implementati nel linguaggio sono:

- `bool`: rappresenta un tipo booleano `true` o `false`;
- `int`: rappresenta un tipo intero;
- `pointer`: rappresenta un tipo puntatore. Questo può puntare ad un altro puntatore oppure ad un tipo primitivo quindi booleano o intero;
- `void`: rappresenta il tipo vuoto;
- `arrowType`: tipo utilizzato per le *funzioni*, contiene la lista dei tipi dei parametri e il tipo ritornato.

Particolare attenzione viene rivolta verso il corretto uso dei puntatori e la conformità dei parametri formali con gli attuali.

#### 4.1.1 Corretto uso dei puntatori

Il corretto uso dei puntatori è definito da alcuni punti cruciali:

- Tutte le referenze di un puntatore devono essere inizializzate per poter essere scritte/lette;
- Prima di usare un puntatore cancellato questo va reinizializzato;
- Non è possibile accedere ad un puntatore cancellato;
- Il livello di dereferenziazione durante l’assegnamento deve combaciare.

Dei punti precedentemente trattati, l’unico che viene assolto durante la fase di controllo dei tipi è quello che riguarda il libello di dereferenziazione dei puntatori assegnati. Riguardo agli altri controlli sull’inizializzazione rimandiamo alla sezione dell’analisi degli effetti.

#### 4.1.2 Parametri attuali non conformi ai parametri formali

Il controllo sui parametri di una funzione avviene in due parti, la prima controlla che il numero dei parametri sia lo stesso di quello presente nella dichiarazione della funzione invocata ed inseguito viene controllato che anche i tipi combacino.

```
public TypeNode typeCheck () throws SimplanPlusException {
    List<TypeNode> p = t.getParList();
    if ( !(p.size() == parameterlist.size()) )
        throw new SimplanPlusException('Wrong number of parameters in the
            invocation of '+id);

    for (int i=0; i<parameterlist.size(); i++)
        if ( !(TypeUtils.isSubtype( (parameterlist.get(i)).typeCheck(), p.get(i)) )
            )
            throw new SimplanPlusException('Wrong type for '+ (i+1) + '-th parameter
                in the invocation of '+id);
}
```

## 5 Analisi degli effetti

Il compilatore di `SimplanPlus` effettua un controllo statico della presenza di errori dovuti alla cancellazione di puntatori o all'uso di puntatori non inizializzati. Tale proprietà è *estensionale*, dunque, per il teorema di Rice, non è decidibile.

Per questa ragione, durante la fase di sviluppo del controllore degli effetti abbiamo dovuto affrontare il problema relativo alla definizione di tale sistema di verifica. In primo luogo, abbiamo dovuto effettuare una considerazione riguardo alla tipologia di approssimazione di tale proprietà. Una approssimazione decidibile  $Q$  di una proprietà decidibile  $P$  non può coincidere con  $P$ . Deve altresì verificarsi una delle due configurazioni  $Q \subset P$  oppure  $P \subset Q$ . Nel primo caso, la proprietà  $Q$  di cui godono i programmi corretti è più stringente di quella che si cerca di approssimare; esempio di questo tipo di approssimazioni sono i *type checker* convenzionali, come ad esempio quello del C. Altrimenti, il controllore ammette che alcuni programmi che non godono della proprietà soggetta di verifica siano accettati; un simile caso è quello dei linter che analizzano i programmi alla ricerca di alcuni pattern di programmazione nocivi.

Nel nostro caso, il controllo *eager* del corretto uso dei puntatori e dell'accesso a variabili non iniziate, sebbene approssiabile nel caso di programmi sequenziali, sarebbe diventato oltremodo complesso in presenza di chiamate di funzioni. Per un risultato corretto in questo caso, avremmo dovuto adottare delle approssimazioni troppo grossolane della probabilità cercata a causa della possibilità di aliasing fra i parametri e dall'accesso a variabili esterne.

Oppure, avremmo dovuto imporre limitazioni sintattiche (semantiche) ai programmi, proibendo situazioni come quelle descritte sopra. Questo approccio avrebbe però limitato le proprietà espressive del nostro linguaggio.

Inoltre, i più diffusi linguaggi di programmazione non effettuano questo tipo di controlli o limitazioni.

Per queste considerazioni, abbiamo scelto di implementare un controllore che non fosse in grado di impedire l'esecuzione programmi errati durante la fase di compilazione, ma che fosse in grado di rilevare alcune di queste problematiche

### 5.1 Struttura ambiente

Nel contesto di un controllo di tipi, un ambiente è una funzione parziale da identificativi di variabili ad un insieme di valori che descrivono modularmente la proprietà di interesse.

L'ambiente a cui siamo ricorsi, è stato implementato per mezzo della medesima symbol table utilizzata in fase di analisi semantica e di controllo dei tipi, in modo da rappresentare naturalmente la struttura *a stack* dell'insieme degli identificativi dovuta alla presenza di blocchi all'interno del codice.

Una particolarità di tale struttura è la sua intrinseca disomogeneità: mentre agli identificatori di variabili sono associati effetti semplici o composti, agli identificatori di funzioni sono associati ambienti, rendendo le strutture degli effetti e delle funzioni mutualmente ricorsive.

$$\begin{aligned}\Sigma &::= \{[x_0 \rightarrow Eff] \dots [x_n \rightarrow Eff]\} :: \Sigma \mid nil \\ Eff &::= Eff :: Eff \mid nil \mid \Sigma \\ BEff &::= \perp \mid rw \mid d \mid \top\end{aligned}$$

### 5.2 Effetti

La sintassi degli effetti è descritta dal non terminale *Eff* nella grammatica appena rappresentata.

Gli effetti base sono:

- $\perp$  che rappresenta una locazione di memoria appena inizializzata e sulla quale non sono ancora avvenuti accessi.
- $rw$  rappresenta una locazione di memoria sulla quale sono avvenuti accessi.
- $d$  rappresenta una locazione di memoria che contiene un indirizzo al quale non è possibile accedere.
- $\top$  rappresenta una situazione di errore certo.

Ad ogni identificativo di variabile è associata una lista simile a quelle prodotte dal non terminale *Eff* che contiene, per ogni possibile livello di dereferenziazione il peggiore possibile stato per tale casella di memoria.

Agli identificativi di funzione sono associati ambienti che rappresentano l'effetto che l'applicazione di queste porta alle variabili d'ambiente.

Per il calcolo degli effetti di funzioni possibilmente ricorsive è necessario applicare il metodo del punto fisso, anche per tale ragione, è definita una relazione d'ordine totale su quei quattro valori. Tale ordine è descritto da:

$$\perp < rw < d < \top$$

Dati due effetti, è definita l'operazione *max* come intuitivamente inteso. L'operazione di sequenzializzazione è definita da:

$$Seq(x, y) := \begin{cases} max(x, y) & \text{se } x = rw \vee y = rw \\ d & \text{se } x = rw \wedge y = d \\ \top & \text{altrimenti} \end{cases}$$

Infine è definito un operatore di sequenzializzazione parallela *Par* definito come

$$Par(x, y) := max(Seq(x, y), Seq(y, x))$$

Simili funzioni sono state definite generalizzando quelle appena illustrate al caso di liste di effetti. L'operatore di composizione sequenziale è utilizzato per calcolare l'effetto risultante dall'applicazione di un effetto *y* ad uno stato *x*. L'operatore di sequenzializzazione parallela è usato per risolvere semplici situazioni di aliasing fra parametri di funzioni.

### 5.3 Limiti

Uno dei limiti del controllore di tipi implementato è l'insensibilità alla presenza di valori non dichiarati. In particolare, dal momento che la funzione di sequenzializzazione fra un valore  $\perp$  ed uno  $rw$  è  $rw$ , qualora siano presenti valori non inizializzati, il sistema di controllo degli effetti non è in gradi di rilevare l'errore.

Gli errori dovuti alla non inizializzazione dei puntatori vengono rilevati dinamicamente: ogni volta che il valore dello stack pointer viene incrementato, i valori nell'intervallo compreso fra i due valori vengono contrassegnati come illeggibili. Similmente, a seguito della liberazione di un'area dello heap tale area viene contrassegnata come illeggibile.

La macchina virtuale è in grado di sollevare un errore dinamicamente qualora venga rilevato un accesso ad un valore illeggibile sollevando un'eccezione. Dal momento che anche a seguito della contrassegnazione

di non leggibilità anche a seguito di cancellazione di aree allocate, anche errori dovuti a situazioni complesse di aliasing dovute ad esempio ad assegnazioni di puntatori dereferenziati parzialmente sono certamente rilevate dinamicamente dalla virtual machine che abbiamo implementato.

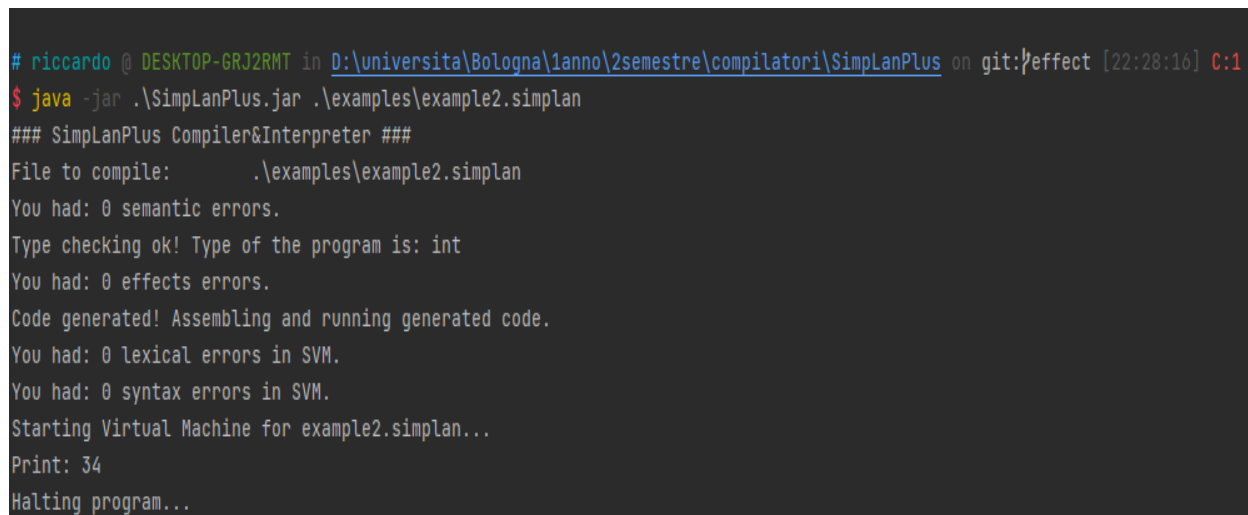
## 6 Compilatore

### 6.1 Compilatore SimplanPlus

Il compilatore può essere eseguito tramite il comando `java -jar SimplanPlus.jar [nomefile]` il quale si occuperà di:

- Controllare errori lessicali;
- Controllare errori semantici;
- Controllare errori di tipo;
- Controllare errori di effetto;
- Compilare il codice in bytecode;
- Eseguire il bytecode generato;

Quindi `compile(String fileAbsolutePath, String fileName)` si occupa di controllare tutti gli errori di tipo e nel caso non vi siano presenti genera il bytecode e lo salva nell'apposita cartella. Da qui subentra `interpreter(Instruction[] code, String filename)` che riceve il bytecode e lo esegue sulla VM.



```
# riccardo @ DESKTOP-GRJ2RMT in D:\universita\Bologna\1anno\2semestre\compilatori\SimplanPlus on git:peffect [22:28:16] C:1
$ java -jar .\SimplanPlus.jar .\examples\example2.simplan
### SimplanPlus Compiler&Interpreter ###
File to compile:      .\examples\example2.simplan
You had: 0 semantic errors.
Type checking ok! Type of the program is: int
You had: 0 effects errors.
Code generated! Assembling and running generated code.
You had: 0 lexical errors in SVM.
You had: 0 syntax errors in SVM.
Starting Virtual Machine for example2.simplan...
Print: 34
Halting program...
```

**Figure 1:** Esempio di esecuzione di un file.

## 7 Interprete

### 7.1 Interprete SimplanPlus

L'interprete di **SimplanPlus** prende in input il codice precedentemente generato dal compilatore e lo esegue un'istruzione alla volta.

#### 7.1.1 Bytecode

Per ogni nodo dell'AST è presente un metodo *String codeGeneration()* che genera il bytecode rispettando la grammatica presente nel file *SVG.g4*. Ogni istruzione è composta da:

- **Label istruzione:** è una stringa obbligatoria che identifica un'istruzione.
- **Primo argomento:** serve per specificare un registro o un'etichetta. È una stringa opzionale perché non tutte le istruzioni prevedono questo argomento (es. POP).
- **Secondo argomento:** serve per specificare un registro o un numero. È una stringa opzionale.
- **Terzo argomento:** serve per specificare un registro, un numero o un'etichetta. È una stringa opzionale.

Per ogni istruzione letta viene creato un oggetto di tipo *Instruction*, definito dalla classe *Instruction.java*.

#### 7.1.2 CPU e memoria

L'interprete simula il comportamento di una CPU e di una memoria. La CPU ha 9 registri:

- **Stack pointer (\$sp):** punta alla cima dello stack
- **Instruction pointer (\$ip):** indica la prossima istruzione da eseguire
- **Heap pointer (\$hp):** punta alla prima posizione libera dello *heap*
- **Frame pointer (\$fp):** punta all'access link corrente relativo al frame attivo
- **Access link (\$al):** registro utilizzato per attraversare la catena statica degli scope
- **Base stack pointer (\$bsp):** punta alla cella di memoria contenente il vecchio stack pointer, utile nel caso si esca da una funzione prima della fine delle istruzioni tramite un return
- **Accumulatore (\$a0):** registro utilizzato per salvare il valore calcolato da alcune espressioni
- **Registro generico (\$t1):** utilizzabile liberamente all'interno del programma
- **Return address (\$ra):** registro utilizzato per salvare l'indirizzo di ritorno una volta usciti da un frame

La CPU esegue le istruzioni passate al costruttore dall'interprete. Dopodiché, ad ogni iterazione, si accede all'istruzione indicata dal registro *\$ip* che poi viene incrementato di uno per passare ad eseguire l'istruzione successiva. L'operazione da eseguire dipende dal campo *code* dell'oggetto *Instruction*.

Per gestire i registri si utilizzano le funzioni *regStore* e *regRead* per aggiornare e leggere il valore contenuto

nel registro opportuno.

Per quanto riguarda la *memoria*, questa viene gestita come un array di *Cell*. La memoria è divisa logicamente nello stack (che cresce verso il basso e quindi indici della memoria minori) e lo heap (che parte da indice 0 e cresce verso l'alto e quindi verso indici maggiori).

La classe *Cell* rappresenta una singola cella di memoria. I campi presenti in questa classe sono:

- *Integer val*: rappresenta il valore salvato nella cella di memoria
- *boolean pointed*: indica se la cella della memoria è puntata oppure è vuota e quindi può essere scritta

Sono stati implementati i seguenti metodi per facilitare la gestione delle celle di memoria:

- *void free()*: libera una cella di memoria
- *Integer read()*: ritorna il dato contenuto nella cella di memoria
- *boolean isPointed()*: ritorna un true o false se la cella è in uso oppure no
- *void allocate()*: setta a true il campo *isPointed* e quindi segna come in uso la cella di memoria
- *Integer write(int v)*: aggiorna la cella con *v*

Nella classe *Memory* sono presenti i seguenti metodi per la gestione della memoria:

- *int read(int n)* che ritorna il contenuto della memoria nella cella *n*
- *void write(int add, int val)* che scrive il valore *val* all'indirizzo *add*
- *void free(int add)* che chiama la *free* su quella cella, mettendo il campo *pointed* a false e il campo *val* a null
- *void cleanMemory(int start, int end)* che re-inizializza tutte le celle comprese tra start ed end

### 7.1.3 Record di attivazione

Lo *stack* in memoria è diviso in record di attivazione (*ra*). Quando si entra in un nuovo blocco, si pusha il nuovo *ra*. All'uscita dal blocco viene fatta una operazione di pop del *ra* attuale. Il record di attivazione è composto come segue:

- **Old frame pointer**: contiene l'indirizzo del frame pointer prima del push del nuovo *ra*
- **Old stack pointer**: contiene l'indirizzo dello stack pointer prima del push del nuovo *ra*
- **Return address**: contiene il valore dell' *\$ip* e quindi l'istruzione da eseguire una volta terminata l'esecuzione attuale
- **Access link**: serve per accedere alle variabili dichiarate in scope esterni
- **Spazio di memoria riservato alle variabili**



#### 7.1.4 Come funziona l'interprete?

Il funzionamento dell'interprete possiamo vederlo in *SVM.java*. Il costruttore prende in input il codice passato all'interprete e lo assegna al campo `code`. La classe SVM ha i seguenti campi privati:

- **Instruction[] code**: andrà a contenere la sequenza ordinata di istruzioni che l'interprete dovrà eseguire
- **Memory memory**: rappresenta la memoria dell'interprete che è logicamente divisa in *heap* e *stack*
- **I registri**: **ip** (inizializzato a 0), **sp** (inizializzato a *memory\_size*), **hp** (inizializzato a 0), **fp** (inizializzato a *memory\_size1*), **ra** (inizializzato a null), **al** (inizializzato a null), **bsp** (inizializzato a *memory\_size*)
- **int[] a = new int[10]**: insieme dei registri temporanei

Il metodo *cpu()* della classe SVM esegue le istruzioni all'interno di *code* mediante un ciclo *while(true)* fino a quando lo *heap* supera lo *stack* dando un errore di *Out of memory*. Ad ogni iterazione viene letta l'istruzione da eseguire, vengono salvati gli argomenti che possiede e incrementato l'*ip*. Tramite un grosso switch si gestisce il comportamento in base all'istruzione in esecuzione.

## 8 Esempi

### 8.1 Esempio 1