Compilatori e Interpreti
Relazione progetto "SimpLanPlus"
A.A. 2020/2021

Gianluigi Carrozzo
Gaia Ghidoni

21 luglio 2022

Indice

1 Introduzione

In questo progetto è stata richiesta l'implementazione di un compilatore per un linguaggio di programmazione semplificato fornito dal professore, il cui nome è *SimpLanPlus*. Questo linguaggio di programmazione è una versione estesa del linguaggio di programmazione *SimpLan* visto durante le lezioni. Oltre al compilatore, è stata richiesta la definizione di un linguaggio bytecode e l'implementazione di un interprete.
L'obiettivo finale è quello di compilare ed eseguire i programmi scritti nel linguaggio *SimpLanPlus*. Si è assunto che i programmi possano essere ricorsivi, ma non mutuamente ricorsivi.
In particolare, al compilatore è stato richiesto di controllare:

- l'uso di variabili o funzioni non dichiarate;
- le variabili dichiarate più volte nello stesso ambiente;
- l'uso di variabili non inizializzate;
- il corretto uso dei puntatori;
- la correttezza dei tipi;
- gli accessi a identificatori "cancellati".

Il compilatore esegue 4 analisi nel seguente ordine:

1. *analisi lessicale*: si occupa di dividere il codice del programma in input in token e verifica la correttezza di ogni singolo token;
2. *analisi sintattica*: si occupa di verificare la correttezza sintattica del programma e costruisce l'albero di sintassi astratta (AST);
3. *analisi semantica*: si occupa di verificare la correttezza semantica del programma, ad esempio, controlla che non ci siano variabili non dichiarate o variabili dichiarate più volte nello stesso ambiente;
4. *analisi dei tipi*: si occupa di verificare la correttezza dei tipi, ad esempio, controlla che vengano utilizzati i tipi corretti negli assegnamenti o nelle chiamate di funzione;
5. *analisi degli effetti*: si occupa di verificare gli effetti delle variabili, ad esempio, controlla che non ci siano accessi a variabili non inizializzate o a puntatori "cancellati".

Se la compilazione termina con successo, viene generato il codice intermedio utilizzando il linguaggio bytecode che è stato definito nel progetto. Il codice intermedio viene poi dato in input all'interprete che esegue le istruzioni su un modello di memoria semplificato.

2 Grammatica

Per l'analisi lessicale e sintattica è stato utilizzato il tool *ANTLR*. A partire dalla grammatica `SimpLanPlus.g4`, *ANTLR* genera automaticamente il lexer e il parser. Il primo verifica la presenza di errori lessicali, invece, il secondo verifica la presenza di errori sintattici.
Per per la visualizzazione degli errori sintattici è stata utilizzata la classe `SLPErrorListener` (*SimpLanPlus* error listener), la quala permette di ottenere maggiori informazioni su un eventuale errore sintattico, come ad esempio, il numero di riga e il punto preciso nella riga dove è stato riscontrato l'errore. Per utilizzare `SLPErrorListener` è stato sufficiente rimuovere il listener di default e aggiungere al parser il nuovo error listener.

Regole lessicali
Di seguito sono riportate tutte le regole utilizzate dal lexer nell'analisi lessicale:

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

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

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

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


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

Regole sintattiche
Di seguito sono riportate tutte le regole utilizzate dal parser nell'analisi sintattica:

- Ogni programma scritto nel linguaggio *SimpLanPlus* deve iniziare con un blocco. All'interno di un blocco, la lista delle eventuali dichiarazioni deve precedere quella degli eventuali statement.

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

- Le dichiarazioni sono di due tipi: dichiarazione di funzione e dichiarazione di variabile. Nella dichiarazione di funzione bisogna specificare il tipo della funzione (int, bool o void), il nome della funzione, eventuali argomenti (variabili o puntatori) e il corpo della funzione. Le funzioni non possono essere di tipo puntatore. Nelle dichiarazione di variabile bisogna specificare obbligatoriamente il tipo della variabile (int, bool o puntatore) e il nome della variabile, mentre, opzionalmente è possibile assegnare un'espressione alla variabile.

```
declaration : decFun            #decFunL
            | decVar            #decVarL;

decFun      : (type | 'void') ID '(' (arg (',' arg)*)? ')' block;

decVar      : type ID ('=' exp)? ';' ;

type        : 'int'
            | 'bool'
            | '^' type ;

arg         : type ID;
```

- Anche gli statement possono essere di vario tipo: assegnamenti, cancellazione dei puntatori, stampa di un'espressione, `return` di un'espressione, costrutto `if-then-else`, chiamata di funzione e dichiarazione di un blocco annidato. Notare che:
  - la cancellazione è permessa solo sui puntatori inizializzati, ed elimina il riferimento alle celle di memoria nello heap, le quali vengono liberate;
  - dopo una cancellazione non si può più accedere al puntatore;
  - il `return` può essere utilizzato solo all'interno di una funzione.

```
statement   : assignment ';'    #assignmentL
            | deletion ';'      #deletionL
            | print ';'         #printL
            | ret ';'           #retL
            | ite               #iteL
            | call ';'          #callL
            | block             #blockL;
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)*)? ')';
```

- Le espressioni comprendono i valori numerici, i valori booleani e l'accesso alle variabili ed ai puntatori. Sulle espressioni è possibile utilizzare operatori aritmetici (addizione, sottrazione, moltiplicazione, divisione), operatori logici (and, or, not) e operatori di confronto ($<$, $\leq$, $>$, $\geq$, $==$, $!=$). Notare che:
  - per inizializzare un puntatore bisogna invocare la *new* tante volte quanti sono gli ^ nella dichiarazione del puntatore;

– non è consentita l'aritmetica sugli indirizzi dei puntatori.

```
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;
```

Se entrambe le analisi, lessicale e sintattica, terminano con successo, allora il compilatore passa all'analisi successiva, ovvero l'analisi semantica.

3 Analisi semantica

La fase di analisi semantica prevede l'esecuzione dei controlli su scope, tipi ed effetti delle variabili, implementati rispettivamente nei metodi *checkSemantics*, *typeCheck* e *checkEffects*. Per queste verifiche sarà fondamentale la creazione e l'utilizzo di un ambiente in cui memorizzare le informazioni su *scope* e variabili.

3.1 Ambiente

Il linguaggio SimpLanPlus permette la creazione di scope annidati, attraverso l'utilizzo di blocchi. Gli identificatori di variabili e funzioni devono essere univoci all'interno di ogni scope, ma sono permesse dichiarazioni multiple dello stesso identificatore se effettuate in scope differenti, anche annidati.

L'ambiente è stato implementato attraverso la classe `Environment`, che racchiude le funzionalità legate alla gestione della tabella dei simboli. L'ambiente verrà utilizzato per tutte le tre fasi dell'analisi semantica precedentemente elencate. La classe `Environment` fa uso della classe `STEntry` che rappresenta le informazioni associate ad un identificatore in uno scope.

Più in dettaglio `Environment` presenta i seguenti campi:

- `symbolTable`: memorizza la tabella dei simboli ed è implementata come una lista di HashMap. Ogni elemento della lista è uno scope dell'ambiente, rappresentato tramite una HashMap che associa una *entry* (valore di classe `STEntry`) ad ogni identificatore dichiarato (chiave);
- `nestingLvl`: memorizza il livello di annidamento corrente. Nel momento della creazione di un ambiente il valore di nestingLvl viene impostato a -1, e verrà incrementato ad ogni aggiunta di scope. In questo modo il blocco principale del programma possiede livello di nesting uguale a 0, mentre gli scope più interni avranno un valore positivo di nesing level;
- `offset`: memorizza il valore aggiornato dell'offset nello scope corrente, che viene inizializzato a -2 nel momento della creazione dell'ambiente (in accordo con le scelte effettuate sulla gestione della memoria).

Tra i metodi presenti nella classe `Environment`, si riportano i quattro corrispondenti alle principali operazioni che si possono svolgere sulla tabella dei simboli:

- `addScope`: metodo per l'aggiunta di un nuovo scope alla tabella dei simboli. Permette di inserire una nuova HashMap alla tabella dei simboli, incrementandone il livello di nesting;
- `removeScope`: metodo per la rimozione dell'ultimo scope della tabella dei simboli, corrispondente al livello di nesting corrente che verrà quindi decrementato;
- `addEntry`: metodo per l'aggiunta di una nuova entry nello scope corrente;
- `lookup`: metodo per la ricerca di un identificatore all'interno dell'ambiente. Gli scope vengono percorsi dall'ultimo al primo aggiunto, fino al ritrovamento dell'STEntry corrispondente alla prima occorrenza dell'identificatore, se presente.

Le principali informazioni sugli identificatori necessarie per la fase di compilazione sono memorizzate nei campi della classe `STEntry`, tra i quali si trovano:

- il campo `type` che memorizza il tipo dell'identificatore;
- il campo `nestingLvl`, contenente il livello di annidamento in cui si trova l'identificatore;
- il campo `offset` che registra la posizione dell'identificatore rispetto alle altre dichiarazioni dello scope;
- un campo per la memorizzazione dello stato dell'identificatore (necessario durante la fase di analisi degli effetti).

3.2 Analisi degli scope

La prima fase di analisi semantica viene eseguita tramite il metodo `ArrayList<SemanticError> checkSemantics(Environment env)` sulla radice dell'albero sintattico (AST), che permette la creazione dell'ambiente e il controllo degli scope andando a visitare, se presenti, tutti i figli.

In particolare, un nuovo scope viene aggiunto all'istanza di `Environment` all'entrata di ogni blocco (delimitato da parentesi graffe), e viene rimosso alla sua uscita. Ad ogni dichiarazione di un identificatore (variabile o funzione), lo scope corrente viene popolato con l'aggiunta di una nuova STEntry.

Durante questo processo vengono effettuati alcuni controlli legati alla semantica del linguaggio e alla correttezza degli scope. Se vengono individuati degli errori, questi vengono restituiti al termine dell'esecuzione della `checkSemantics` tramite la classe ausiliaria `SemanticError`, e la compilazione viene terminata.

In particolare vengono effettuati i seguenti controlli:

- **Dichiarazioni multiple di identificatori**. Quando si incontrano dichiarazioni di variabili e funzioni, viene creata una nuova STEntry che viene aggiunta allo scope corrente tramite il metodo `addEntry` di `Environment`. Questo metodo lancia una eccezione nel caso in cui lo scope corrente contenga già una entry per lo stesso id, che viene catturata dal `checkSemantics` per poter riportare un `SemanticError`;

- **Identificatori non dichiarati**. L'utilizzo di un identificatore deve essere permesso solamente se tale id è stato precedentemente dichiarato in uno scope raggiungibile dalla posizione in esame. Questo controllo viene eseguito utilizzando il metodo `lookup` di `Environment`: se l'identificativo non è presente nell'ambiente, viene riportato un errore semantico;

- **Uso corretto dei *return***. E' stato imposto che l'istruzione *return* può essere utilizzata solamente all'interno del corpo delle funzioni. Un campo ausiliare nella classe `RetNode` (classe rappresentante il nodo *return* nell'AST) permette di sapere se l'istruzione si trova all'interno di una funzione. In caso contrario viene restituito un errore semantico. Inoltre si impone che, quando presente, un *return* deve essere l'ultimo statement del blocco, in modo da evitare la presenza di codice irraggiungibile. In caso di codice non raggiungibile viene ritornato un errore semantico.

Infine, si evidenzia che durante questa fase di analisi semantica è resa possibile la ricorsione (ma non la mutua ricorsione) per le funzioni del linguaggio *SimpLanPlus*.

Il corpo di una funzione rappresenta un nuovo scope rispetto a quello della dichiarazione di funzione. Questo scope contiene i parametri formali e al suo interno è possibile accedere alle funzioni dichiarate precedentemente, ma non alle variabili degli scope più esterni. Inoltre, nel corpo della funzione non è possibile effettuare nuove dichiarazioni utilizzando l'identificativo della funzione stessa.

3.3 Analisi dei tipi

La seconda fase dell'analisi semantica riguarda il controllo sulla correttezza dei tipi, che necessita una seconda visita all'AST e l'esecuzione del metodo `typeCheck`. Questo metodo restituisce il tipo del nodo o il valore *null* nel caso in cui il tipo non sia di interesse per l'analisi del codice. Se viene individuato un errore di tipo, il metodo `typeCheck` solleverà l'eccezione `TypeErrorException` e la compilazione verrà terminata.

I tipi del linguaggio SimpLanPlus sono i seguenti:

- **Int**: per la rappresentazione del tipo intero;

- **Bool**: per la rappresentazione del tipo booleano;

- **Pointer**: per la rappresentazione del tipo puntatore. Un puntatore avrà a sua volta un tipo associato, che rappresenta il tipo del valore puntato e può quindi essere *int*, *bool* o *pointer*;

- **Void**: per la rappresentazione del tipo vuoto, che viene usato unicamente per il tipo di ritorno delle funzioni;
- **Arrow**: per la rappresentazione del tipo delle funzioni, e contiene la lista dei tipi dei parametri e il tipo di ritorno della funzione.

Più in particolare, i nodi che ritornano un tipo sono:

- le **espressioni**;
- gli **identificatori**, che ritornano il tipo dichiarato;
- i **return**, che restituiscono il tipo dell'espressione del *return*, oppure tipo *void* se non è presente nessuna espressione;
- le **chiamate di funzioni**, che ritornano il tipo di ritorno della funzione;
- gli statement **if-then-else**, che restituiscono il tipo dei rami (si impone che i rami siano dello stesso tipo);
- i **blocchi**, che possono ritornare un tipo oppure il valore *null*. In particolare, un tipo viene ritornato solamente se il blocco si trova all'interno di una funzione e contiene una istruzione che ritorna un tipo. In caso contrario viene restituito *null*.

Di seguito si evidenziano alcune delle scelte implementate per l'analisi dei tipi.

Puntatori

Come accennato precedentemente, è possibile creare puntatori semplici, ad esempio `^int x;`, ma sono anche permesse le catene di puntatori, come nell'esempio di puntatore a puntatore `^^int x;`. E' quindi necessario controllare la correttezza del tipo assegnato al valore puntato, ma anche ad ogni eventuale puntatore di una catena. Con riferimento alla variabile `x` precedente, il seguente codice solleva un errore di tipo a causa dello scorretto assegnamento tra puntatori: `^ int y; x=y;` (perchè un puntatore non è uguale ad un puntatore di puntatore).

Similmente, quando un puntatore viene utilizzato, è necessario controllare che l'utilizzo della *dereferenziazione* sia adeguata (il numero di dereferenziazione non può superare quello dichiarato).

Inoltre, il controllo dei tipi permette di verificare che l'istruzione `delete` venga applicata solamente a identificatori di puntatori. In caso contrario, l'uso è impedito e il compilatore restituisce un errore di tipo.

Funzioni

Nelle dichiarazioni di funzioni si controlla l'uguaglianza tra il tipo di ritorno dichiarato e il tipo restituito dal corpo della funzione, rappresentato da un blocco.

Quando si effettua una chiamata a funzione, invece, l'esecuzione di `typeCheck` controlla che il numero ed il tipo dei parametri attuali sia conforme con la dichiarazione della funzione stessa.

3.4 Analisi degli effetti

L'analisi degli effetti viene eseguita solo quando l'analisi degli scope e dei tipi termina senza errori. In questa analisi, ad ogni identificatore viene associato un effetto e si va a verificare che gli effetti di tutti gli identificatori siano corretti. Se viene trovato un effetto erroneo, il programma non deve essere eseguito.

I tipi effetto degli identificatori sono:

- $\perp$: significa che la variabile è stata dichiarata, ma non è stata ancora inizializzata;

- rw: significa che la variabile è stata inizializzata, quindi è possibile accedervi sia in lettura che in scrittura;
- d: significa la variabile è cancellata, non è più possibile accedervi, né in lettura né in scrittura;
- $\top$: significa che si è verificato un errore.

B è l'insieme dei tipi effetto, cioè $B = \{\bot, rw, d, \top\}$, ed è un dominio di ordine parziale con la seguente relazione di ordinamento $\bot \leq rw \leq d \leq \top$. Su **B** sono definite tre operazioni monotone: *max*, *seq*, *par*.

Effetti

La classe `Effect` è stata creata per rappresentare gli effetti associati agli identificatori. Per implementare l'ordinamento tra i tipi effetto, nella classe è presente un attributo statico di tipo intero per ogni tipo di effetto, ad esempio, $\bot = 0$, $rw = 1$ e così via. La classe ha l'attributo `value` che rappresenta l'effetto corrente. I metodi della classe implementano le tre operazioni sugli effetti (*max*, *seq*, *par*).

Rappresentazione degli effetti associati agli identificatori. Ci sono tre casi:

- gli effetti di una **variabile** sono rappresentati da un array contenente un solo oggetto di tipo `Effect`;
- gli effetti di un **puntatore** sono rappresentati da un array contenente due o più oggetti di tipo `Effect`. Ad esempio, un puntatore semplice avrà un array con due effetti, mentre un puntatore di puntatore (catena di puntatori) avrà un array con tre effetti. L'ultimo effetto nell'array è sempre quello relativo al valore puntato dal puntatore (o dalla catena di puntatori), mentre gli altri effetti nell'array corrispondono agli effetti dei puntatori che compongono la catena di puntatori;
- gli effetti di una **funzione** sono rappresentati da un array contenente un array di oggetti di tipo `Effect` per ogni parametro in input. Gli effetti contenuti in questo array sono quelli dei parametri formali al termine della valutazione del corpo della funzione.

STEntry

Gli effetti di ogni identificatore si trovano nell'oggetto di tipo `STEntry` associato, infatti nella classe `STEntry` sono presenti i seguenti due attributi:

- `varEffects`: è un array di oggetti di tipo `Effect`, quindi rappresenta gli effetti di una variabile o di un puntatore. In questi casi, l'attributo `parEffects` sarà un array vuoto.
- `parEffects`: è un array di array di oggetti di tipo `Effect`, quindi rappresenta gli effetti di una funzione. In questo caso, l'attributo `varEffects` sarà un array vuoto.

Inoltre, nella classe `STEntry` sono presenti dei metodi per ottenere e modificare gli effetti degli identificatori:

- `getVarEffectList`: restituisce un array contenente gli effetti di una variabile o un puntatore;
- `setVarEffectList`: prende in input un array contenente gli effetti di una variabile o un puntatore e setta l'attributo `varEffects` della `STEntry`;
- `getVarEffect`: prende in input un intero n e restituisce l'effetto che si trova in poszione n nell'array `varEffects`;
- `setVarEffect`: prende in input un intero n e un effetto e. Setta l'effetto in posizione n nell'array `varEffects` al valore e;
- `getSizeVarEffects`: restituisce la lunghezza dell'array `varEffects`;

- `getParEffectList`: restituisce un array di array contenente gli effetti di una funzione;
- `setParEffectList`: prende in input un array di array di oggetti di tipo `Effect` e setta l'attributo `parEffects` della `STEntry`.

Environment
Nella classe `Environment` sono presenti dei metodi utilizzati nell'analisi degli effetti (i primi quattro metodi non sono descritti in quanto sono stati implementati esattamente nel modo in cui sono stati visti a lezione):

- `maxEnv`: prende in input due ambienti e restituisce un nuovo ambiente. Implementa l'operazione `max` definita su ambienti;
- `seqEnv`: prende in input due ambienti e restituisce un nuovo ambiente. Implementa l'operazione `seq` definita su ambienti;
- `parEnv`: prende in input due ambienti e restituisce un nuovo ambiente. Implementa l'operazione `par` definita su ambienti;
- `updateEnv`: prende in input due ambienti e restituisce un nuovo ambiente. Implementa l'operazione `update` definita su ambienti;
- `checkExpressionEffects`: prende in input un ambiente e un array di `LhsNode`, restituisce un array di errori. Questo metodo viene usato per valutare gli effetti sulle variabili e puntatori presenti in un'espressione. Per ogni variabile o puntatore viene settato l'effetto al valore risultante della `seq` tra l'effetto corrente della variabile (o puntatore) e l'effetto *rw*. Se la `seq` restituisce l'effetto $\top$, il metodo restituisce un errore.

checkEffects
L'analisi degli effetti effettua la terza visita all'AST e viene eseguita invocando il metodo `ArrayList <SemanticError> checkEffects(Environment env)` sulla radice dell'albero di sintassi astratta (AST), in questo modo viene creato un ambiente e si controllano gli effetti di tutti gli identificatori presenti nel programma.
Il metodo `checkEffects` viene invocato su tutti i nodi figli dell'AST. In input prende sempre un ambiente e restituisce un array di errori trovati durante l'analisi. L'implementazione di questo metodo varia in base al tipo di nodo dell'AST su cui il metodo e' definito, coerentemente con le regole di tipo con effetti viste a lezione.

Variabili non inizializzate
Nella dichiarazione di una variabile, il suo effetto viene settato a $\bot$. Se nella sua dichiarazione viene anche specificata un'espressione, allora l'effetto della variabile viene settato a rw. Per inizializzare una variabile dopo la sua dichiarazione bisogna assegnare un valore alla variabile.

L'accesso alle variabili non inizializzate è possibile solo in scrittura, quindi è permesso l'assegnamento di un'espressione ad una variabile non inizializzata.
Si accede in lettura ad una variabile quando essa compare in un'espressione. In questo caso, se la variabile non è stata inizializzata verrà restituito un errore. Questo controllo è stato implementato nella `checkEffects` della classe `DerExpNode`, ovvero la classe che rappresenta la presenza di un identificatore in un'espressione.

Corretto uso dei puntatori
Gli effetti di un puntatore sono rappresentati da un array di oggetti `Effect` di lunghezza maggiore di 1. Al momento della dichiarazione di un puntatore, tutti gli effetti vengono settati a $\bot$. Nella dichiarazione

di un puntatore è possibile specificare un'espressione di tipo `NewExp`, la quale serve ad allocare una cella nello heap. In questo caso, il primo effetto contenuto nell'array viene settato all'effetto rw.
Per inizializzare un puntatore dopo la sua dichiarazione bisogna assegnare al puntatore un'espressione di tipo `NewExp`, in questo modo l'effetto del puntatore passa da $\bot$ a rw. Un altro modo per inizializzare un puntatore è quello di assegnare un puntatore già inizializzato ad un puntatore non inizializzato, ad esempio `x=y` dove `x` e `y` sono puntatori. In questo caso gli effetti del puntatore `y` vengono copiati negli effetti del puntatore `x`.

- **Puntatori non inizializzati**
  Ad un puntatore non inizializzato si può accedere solo in scrittura. Si accede in lettura quando esso compare in un'espressione. Diversamente delle variabili, in questo caso sono stati distinti due casi di espressioni che possono contenere puntatori:

  – espressioni che compaiono negli assegnamenti e nelle *print*. In questo caso, si deve controllare che l'effetto del puntatore non sia $\bot$;
  – espressioni che compaiono come parametri nelle invocazioni di funzione. In questo caso, non e' necessario che l'effetto del puntatore sia $\bot$. Nelle invocazioni di funzione, non viene fatto il controllo perché è permesso il passaggio di puntatori non inizializzati come argomenti;

  Per distinguere questi due casi viene utilizzato l'attributo booleano `inAssign` nella classe `DerExpNode`. Nella creazione di un oggetto di tipo `DerExpNode`, l'attributo viene settato a *false*, mentre viene settato a *true* nel metodo `checkEffects` delle classi `AssignmentNode` e `PrintNode`. Nella `checkEffects` della classe `DerExpNode`, se il valore `inAssign` è *true* si è nel primo caso, quindi si controlla che il puntatore sia inizializzato, invece, se il valore dell'attributo è *false* il controllo non viene eseguito.

  Infine, anche nelle espressioni dei `return` è necessario controllare che non ci siano puntatori non inizializzati. Questo controllo si trova nella `checkEffects` della classe `LhsNode`.

- **Cancellazione di puntatori**
  E' permesso cancellare solo puntatori che siano interamente inizializzati, ovvero puntatori che abbiano tutti gli effetti nell'array a rw. Se questo non si verifica, allora viene restituito un errore. Questo controllo viene fatto nella `checkEffects` della classe `DeletionNode`.
  Cancellare un puntatore significa settare tutti gli effetti a d. Questo implica che, dopo una cancellazione, il puntatore (o la catena di puntatori) non è più accessibile né in lettura né in scrittura.

  – Se si prova ad accedere in lettura ad un puntatore cancellato, verrà restituito un errore dal metodo `checkExpressionEffects` della classe `Environment`, il quale esegue la `seq` tra l'effetto del puntatore letto e l'effetto rw, se il risultato è $\top$ vuol dire che il puntatore è stato cancellato;
  – Se si prova ad accedere in scrittura ad un puntatore cancellato, verrà restituito un errore dal metodo `checkEffects` della classe `AssignmentNode`, dove viene verificato l'effetto del puntatore a cui si vuole assegnare un'espressione;
  – Se si prova a cancellare un puntatore già cancellato, verrà restituito un errore dal metodo `checkEffects` della classe `DeletionNode`, dove eseguendo l'operazione `seq` tra gli effetti del puntatore e l'effetto d, se il risultato è l'effetto $\top$ allora significa che il puntatore era già stato cancellato.

Dichiarazione e invocazione di funzione

Gli effetti di una funzione sono gli effetti dei parametri formali dopo la valutazione del corpo della funzione. Per calcolare questi effetti è necessario utilizzare il metodo del punto fisso perché il linguaggio *SimpLanPlus* permette la ricorsione.

- **Dichiarazione di funzione**
  Il metodo del punto fisso viene eseguito utilizzando un ambiente composto da due scope. Il primo scope contiene le funzioni dichiarate precedentemente e la dichiarazione della funzione stessa, invece, il secondo scope contiene i parametri formali. Al termine dell'esecuzione del metodo del punto fisso, l'insieme degli effetti dei parametri formali ottenuti viene settato come effetto della funzione nella sua entry nell'ambiente.

  Inoltre, nella entry della funzione vengono memorizzati anche l'oggetto `DecFunNode` e una copia dell'ambiente contenente solo le funzioni dichiarate precedentemente e la funzione stessa. In seguite verrà descritto il loro utilizzo.
  Il metodo `checkEffects` della classe `DecFunNode` restituisce una lista di errori che potrebbe essere vuota oppure contenere gli errori trovati durante il calcolo del metodo del punto fisso, in particolare durante l'analisi degli effetti del corpo della funzione.

- **Invocazione di funzione**
  Il metodo `checkEffects` della classe `CallNode` può essere descritto suddividendolo nelle seguenti fasi:

  1. analisi degli effetti dei parametri attuali.
     - bisogna controllare che le variabili passate come parametro siano inizializzate. Questo viene fatto invocando il metodo `checkEffects` su tutte le espressioni che rappresentano i parametri attuali della funzione.
     - bisogna controllare gli effetti dei puntatori passati come parametro. Dato che ad una funzione possono essere passati anche puntatori non inizializzati, bisogna controllare che prima del loro utilizzo nel corpo della funzione, essi vengano effettivamente inizializzati. Per far questo, dalla entry della funzione si recupera l'oggetto `DecFunNode` corrispondente alla dichiarazione della funzione che si vuole invocare. Si crea un array contenente gli effetti dei parametri attuali e lo si passa come input all'invocazione del metodo `checkEffectsActualArgs` della classe `DecFunNode`. Questo metodo esegue di nuovo il metodo del punto fisso, ma questa volta utilizzando gli effetti dei parametri attuali. Questa riesecuzione del metodo del punto fisso viene eseguita solo una volta, altrimenti in caso di ricorsione si andrebbe in loop;
  2. dalla entry della funzione si recupera il tipo della funzione. Si calcolano gli indici dei parametri che sono puntatori e gli indici dei parametri che non sono puntatori;
  3. dalla entry della funzione si recuperano gli effetti dei parametri formali;
  4. si controlla che i parametri formali, che non sono puntatori, non abbiano un effetto $\top$, altrimenti si restituisce un errore;
  5. si aggiorna l'effetto delle variabili che compaiono nei parametri formali che non sono puntatori. Il nuovo effetto è il risultato dell'operazione `seq` tra l'effetto corrente della variabile e l'effetto rw;
  6. per ogni parametro che è un puntatore, si calcola l'operazione `seq` tra l'effetto del parametro attuale e quello del parametro formale. Al termine si esegue l'operazione `par` per controllare errori dovuti all'aliasing.
  7. si esegue l'operazione `update` su due ambienti, uno contenente i parametri che non sono puntatori e uno contenente i parametri che sono puntatori;
  8. nell'ambiente ottenuto al passo precedente, con il metodo `checkErrors` della classe `Environment` si controlla che le variabili e i puntatori non abbiano effetti uguali a $\top$, altrimenti si restituisce un errore.

  Se l'analisi degli effetti termina senza errori, allora si procederà alla generazione del codice intermedio.

4 Interprete

Conclusa la fase di analisi semantica, l'AST viene visitato un'ultima volta al fine di tradurre il programma ricevuto in input nel codice intermedio che verrà letto dall'interprete.

4.1 Codice intermedio

Il codice intermedio viene generato tramite il metodo `codeGeneration`, in accordo con le scelte implementative della memoria e delle istruzioni *bytecode*. Questo codice viene memorizzato in un file con estensione *.asm*, che verrà quindi controllato dal lexer e dal parser generati da ANTLR per la grammatica del bytecode.

Più in dettaglio, le istruzioni permesse dalla grammatica `SVM` sono le seguenti:

```
instruction:
    (   PUSH    reg=REGISTER
      | POP

      | ADD     res=REGISTER    term1=REGISTER      term2=REGISTER
      | SUB     res=REGISTER    term1=REGISTER      term2=REGISTER
      | MULT    res=REGISTER    term1=REGISTER      term2=REGISTER
      | DIV     res=REGISTER    term1=REGISTER      term2=REGISTER
      | ADDI    res=REGISTER    term1=REGISTER      term2=NUMBER
      | SUBI    res=REGISTER    term1=REGISTER      term2=NUMBER
      | MULTI   res=REGISTER    term1=REGISTER      term2=NUMBER
      | DIVI    res=REGISTER    term1=REGISTER      term2=NUMBER

      | LI      res=REGISTER    term=NUMBER
      | LB      res=REGISTER    term=BOOL
      | STOREW  value=REGISTER  offset=NUMBER   '(' address=REGISTER ')'
      | LOADW   value=REGISTER  offset=NUMBER   '(' address=REGISTER ')'
      | l=LABEL COL
      | BRANCH          l=LABEL
      | BRANCHEQ        term1=REGISTER      term2=REGISTER      l=LABEL
      | BRANCHLESSEQ    term1=REGISTER      term2=REGISTER      l=LABEL
      | JR          reg=REGISTER
      | JAL         l=LABEL
      | MOVE        to=REGISTER     from=REGISTER
      | PRINT       reg=REGISTER    term1=NUMBER
      | DELETION    reg=REGISTER

      | AND     res=REGISTER    term1=REGISTER      term2=REGISTER
      | OR      res=REGISTER    term1=REGISTER      term2=REGISTER
      | NOT     res=REGISTER    term1=REGISTER

      | HALT
      ) ;
```

Le istruzioni `PUSH` e `POP` servono per la gestione della memoria stack. Seguono le operazioni binarie tra due registri (`ADD`, `SUB`, `MULT`, `DIV`), e le corrispettive operazioni tra registro e valore immediato (`ADDI`, `SUBI`, `MULTI`, `DIVI`).

Le operazioni che permettono di caricare dati nei registri sono: `LI` per valori immediati interi, `LB` per valori immediati booleani e `LOADW` per recuperare dati dalla memoria. L'istruzione `STOREW`, invece, permette di salvare in memoria il contenuto di un registro, mentre la `MOVE` sposta il contenuto di un registro in un altro registro specificato.

Sono inoltre presenti le istruzioni per eseguire salti condizionati e non condizionati (`BRANCHEQ`, `BRANCHLESSEQ`, `BRANCH`, `JR`), e quelle per eseguire le operazioni del linguaggio *print* e *delete* (rispettivamente `PRINT` e `DELETION`).

Infine, `LABEL` non è una istruzione ma rappresenta l'etichetta che identifica specifiche sezioni di codice, come l'inizio e la fine di una dichiarazione di funzione, mentre l'istruzione `HALT` indica la fine del programma.

Tutte le istruzioni seguono uno schema comune, che viene implementato nella classe `Instruction` e che può essere rappresentato dai seguenti cinque campi:

Tra questi, "**istruzione**" è l'unico campo obbligatorio, e rappresenta l'operazione che l'interprete dovrà svolgere.

Ogni istruzione necessita una particolare combinazione dei rimanenti campi, ognuno caratterizzato da uno scopo:

- **Argomento 1**: può memorizzare una etichetta oppure un registro;
- **Offset**: contiene un numero che verrà sommato al contenuto di un registro, per l'utilizzo nelle operazioni di caricamento e salvataggio tra registri e memoria;
- **Argomento 2**: nelle operazioni binarie, nelle operazioni di *branch* e nel `move` rappresenta un registro, mentre nelle operazioni di caricamento `LI` e `LB` contiene un valore intero;
- **Argomento 3**: contiene un registro o un numero se utilizzato nelle operazioni binarie, mentre nelle operazioni di *branch* contiene una etichetta.

4.2 Memoria e registri

L'interprete esegue le istruzioni basandosi su un modello di memoria semplificato costituito da **stack**, **heap** e un insieme fissato di **registri**.

In particolare, i registri disponibili sono i seguenti:

- **$a0**: registro accumulatore, che viene utilizzato per memorizzare i risultati di ogni nodo del codice;
- **$t1**: registro utilizzato per memorizzare i valori temporanei, quando necessario;
- **$sp**: Stack Pointer (SP), che punta alla cima dello stack;
- **$fp**: Frame Pointer (FP), che punta alla cella *Access Link* del frame corrente nello stack;
- **$ip**: Instruction Pointer (IP), che indica la prossima istruzione da eseguire;
- **$al**: Access Link (AL), utilizzato per l'implementazione della catena statica;
- **$ra**: Return Address (RA), che memorizza l'indirizzo di ritorno;
- **$hp**: Heap Pointer (HP), che punta alla prima cella libera nella memoria heap;
- **$ret**: registro ausiliare, utilizzato per la gestione dei *return* delle funzioni.

Una classe `Registers`, contenente un campo per ogni registro disponibile, viene utilizzata per facilitare la gestione dei registri (memorizzazione e lettura del contenuto, incremento e decremento dello Stack Pointer, ed aggiornamento dell'Instruction Pointer).

La memoria è invece costituita da due principali aree: la **Code Area**, e la **Data Area**, che rispettivamente memorizzano le istruzioni del codice da eseguire e i dati del programma.

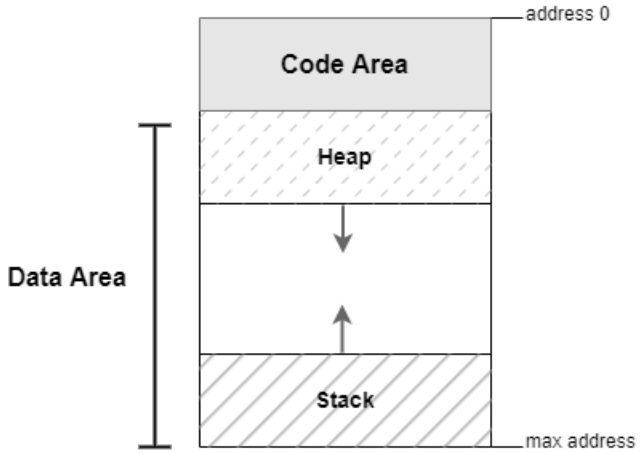
Ogni cella della totalità della memoria (Code Area + Data Area) viene rappresentata da una istanza della classe `Cell`, allo scopo di facilitarne la gestione. Questa classe contiene i campi:

- `isInCodeArea`: booleano per distinguere le celle della Code Area da quelle della Data Area;

- `code`:per la memorizzazione dell'istruzione per le celle nella Code Area;
- `data`: per la memorizzazione dei dati per le celle nella Data Area;
- `isFree`: booleano per distinguere le celle occupate da quelle libere.

I metodi della classe permettono di leggere e aggiornare i campi della cella, eseguendo i necessari controlli per il corretto uso della memoria. Ad esempio: viene impedito l'accesso a dati in lettura se la cella a cui si accede è libera o se si trova nella Code Area.

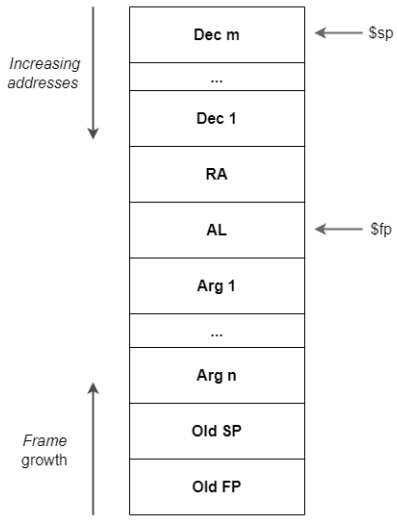
Più in dettaglio la *Data Area* è a sua volta suddivisa in due zone: **stack** e **heap**, che iniziano e crescono verso indirizzi opposti della memoria, come rappresentato dallo schema seguente:



Lo **heap** inizia dagli indirizzi bassi della memoria e cresce verso indirizzi alti, memorizzando le allocazioni dei puntatori.
Lo **stack**, invece, inizia dagli indirizzi alti della memoria e cresce verso gli indirizzi bassi e contiene i dati allocati staticamente.

Nel caso in cui heap e stack dovessero crescere fino a riempire l'intera Data Area, successive allocazioni non saranno più possibili e verrà ritornato un errore di *out of memory*. In modo simile, siccome la dimensione della Code Area può essere specificata dall'utente, all'inizio dell'esecuzione, l'interprete procederà a controllare che la dimensione della Code Area sia abbastanza per memorizzare la totalità delle istruzioni del programma. In caso contrario verrà restituito l'errore `SmallCodeAreaCException("Code area too small!")`.

I dati memorizzati nello Stack sono organizzati in **Activation Record** (o *frame*), che vengono creati all'entrata di uno scope e deallocati alla sua uscita. La struttura dei frame dello stack è la seguente:



In riferimento allo schema qui riportato, *Arg 1 ... n* identificano gli eventuali argomenti delle funzioni, mentre *Dec 1 ... m* rappresentano le dichiarazioni locali allo scope, quando presenti. *RA* contiene il valore di ritorno del blocco, mentre *AL* permette la realizzazione della catena statica. Infine *Old SP* e *Old FP* permettono di ripristinare lo stato del frame precedente al momento dell'uscita del frame corrente.

4.3 Casi particolari di generazione di codice intermedio

In seguito alla spiegazione del modello della memoria utilizzato, in questa sezione si riportano alcuni dettagli e alcuni casi particolari nella generazione di codice intermedio.

- Quando viene eseguita una *delete* di un puntatore, tutta l'eventuale catena di puntatori intermedi memorizzata sullo heap viene cancellata, oltre alla cella contenente il valore puntato. Queste celle verranno quindi aggiornate a celle libere;
- Quando viene effettuata una *new*, viene allocata una cella nello heap che conterrà il valore -1 fino al momento dell'assegnamento.
- Come indicato precedentemente, una *label* univoca viene generata per ogni funzione dichiarata. Una seconda *label* viene generata per indicare l'istruzione del codice intermedio a cui saltare in caso di return. Questa etichetta è quindi aggiunta tra il codice del corpo della funzione e l'inizio delle istruzioni per la pulizia e il ripristino della memoria al frame precedente.
- Ogni volta che viene eseguito un *return* all'interno di una funzione, si carica il valore 1 nel registro $ret. Questo registro viene utilizzato specificatamente per controllare se sono stati eseguiti *return*, e all'uscita delle funzioni viene reimpostato al valore di default 0.

  Il valore del registro $ret viene controllato nell'esecuzione dei blocchi. Siccome ogni blocco del linguaggio SimpLanPlus è inteso come uno scope, un *frame* viene allocato e deallocato sullo stack rispettivamente al suo ingresso e alla sua uscita. In caso di *return* all'interno di una funzione, prima di saltare alla fine della stessa è necessario eseguire la corretta rimozione dei frame di ogni eventuale blocco innestato.
- Ogni comando del linguaggio SimpLanPlus deve mantenere invariata la struttura dello stack, e quindi la generazione di codice intermedio è stata implementata in accordo con questa invariante.
- Il risultato di ogni comando del linguaggio viene caricato nel registro $a0.

4.4 Esecuzione dell'interprete

L'interprete è implementato nella classe `ExecuteVM`, che viene istanziata inizializzando i valori dei registri e della memoria.

In particolare, la dimensione della memoria sarà la somma della dimensione della Code Area e della Data Area. I registri, invece, vengono inizializzati nel seguente modo:

- $a0, $t1, $ra, $ret ed $ip sono messi a 0;
- $hp indica la prima cella libera dello heap, ovvero al primo indirizzo libero dopo la Code Area;
- $sp, $fp ed $al sono fissati ad indirizzi successivi a partire dall'ultima cella di memoria (ovvero la prima cella dello stack).

L'esecuzione del codice intermedio avviene tramite il metodo `cpu` della classe `ExecuteVM`. Questo metodo è costituito da un ciclo che controlla la prossima istruzione da eseguire e termina quando incontra l'istruzione di fine programma `HALT`, o nel caso in cui venga sollevata una eccezione legata allo scorretto uso della memoria. Ad ogni nuova istruzione, uno *switch-case*, in base all'istruzione letta, permette di compiere le necessarie operazioni per l'esecuzione del codice.

5 Esecuzione

Per l'implementazione del progetto è stata utilizzata la versione 15.0.1 di Java e la versione 4.9.3 del tool ANTLR.

Per eseguire il progetto bisogna aprire il terminale, posizionarsi nella directory `SimpLanPlus` e utilizzare il seguente comando:

```
java -jar SimpLanPlus.jar <inputFileName> [-ast] [-codesize=n1] [-memsize=n2] [-debug]
```

dove:

- `inputFileName`: indica il path del file contenente il programma in input;
- `ast`: per la visualizzazione dell'albero di sintassi astratta (AST). Default: *false*;
- `codesize`: per specificare la dimensione della *Code Area*. Default: *1000*;
- `memsize`: per specificare la dimensione della *Data Area*. Default: *1000*;
- `debug`: per la creazione del file *debug.txt* contenente lo stato della memoria e dei registri dopo ogni istruzione eseguita dall'interprete. Default: *false*.

Solo l'argomento `inputFileName` è obbligatorio.