Progetto Compilatori e Interpreti
2020-2021

Cotugno Giosuè 0000983620, Gaspari Michele 0001007249,
Giambi Nico 0001007301, Pruscini Davide 0001007343

5 settembre 2021

Introduzione

Il progetto del corso di Compilatori e Interpreti dell'anno accademico 2020-2021 prevede di:

- Realizzare un sistema di analisi semantica per il linguaggio SimpLanPlus
- Definire un linguaggio bytecode per eseguire programmi in SimpLanPlus
- Costruire un compilatore e implementare l'interprete

Nella seguente relazione si discuteranno quindi le scelte implementative fatte per la parte di analisi semantica e per la definizione del bytecode, subito dopo aver riportato qualche nota importante per quanto riguarda la struttura del progetto e della sua organizzazione. Il codice è stato prodotto con Java 16, utilizzando l'IDE Intellij con il plugin di ANTLR (versione 4.9.2), mentre in allegato ai sorgenti del progetto sono presenti anche delle istruzioni per eseguire il progetto sull'IDE Eclipse.

Struttura e Organizzazione

L'entry point del sistema è il file *Test.java*, in cui vengono richiamati in sequenza il lexer e il parser che controllano il file di input contenente il codice da compilare ed eseguire. Sia lexer che parser sono generati automaticamente da ANTLR a partire dalla grammatica specificata in *SimpLanPlus.g4*, dunque il problema di verificare la correttezza lessicale e sintattica del codice scritto nel file in input viene risolto automaticamente da ANTLR. In particolare, in questa fase precedente all'analisi semantica e alla definizione del linguaggio bytecode, è interessante sottolineare che il sistema notificherà un eventuale errore lessicale/sintattico gestendo un'eccezione e arrestando la compilazione. Fino a quando non saranno corretti tutti gli errori lessicali e sintattici non sarà quindi possibile passare all'analisi semantica. Questo tipo di comportamento è possibile grazie ad ANTLR che consente di specificare un listener di eventi che può essere aggiunto sia al lexer che al parser. Tale scelta è più conforme al normale comportamento di un compilatore e soprattutto non consente di compilare ed eseguire programmi lessicalmente o sintatticamente errati. Una volta completata l'analisi lessicale/sintattica è possibile passare all'analisi semantica e quindi si eseguono delle visite dell'albero di sintassi astratta. Vengono eseguite tre visite per l'analisi semantica, una per la visualizzazione dell'*AST* e una extra per la generazione di codice:

1. *checkSemantics()*
2. *typeCheck()*
3. *checkEffects()*
4. *toPrint()*
5. *codeGeneration()*

Queste 5 visite sono sequenziali, ognuna di esse può iniziare se la precedente non ha riportato errori e vengono descritte da un interfaccia generata da ANTLR a partire da *SimpLanPlus.g4*. Più nello specifico l'interfaccia *SimpLanPlusVisitor.java* definisce diverse visite per ogni nodo visitato e vengono dunque definite le visite di nodi *statement*, *block*, *declaration*, etc. A partire da quest'interfaccia si procede a scrivere il codice del sorgente *SimpLanPlusVisitorImpl.java* che implementa l'interfaccia appena menzionata. In questo file dunque è possibile modificare il normale comportamento di una visita dell'albero di sintassi e di manipolare le strutture dati generate da ANTLR in modo da poter svolgere l'analisi semantica. Di conseguenza viene naturale suddividere i file del progetto tenendo conto della visita dell'albero, in particolare:

- i sorgenti associati a nodi dichiarazione (che nel linguaggio corrispondono a dichiarazioni di variabili e funzioni) sono raggruppati assieme nel package *ast.node.declaration*
- i sorgenti associati a nodi espressione sono raggruppati assieme nel package *ast.node.exp*
- i sorgenti associati a nodi istruzione sono raggruppati assieme nel package *ast.node.statement*
- i sorgenti associati a nodi che corrispondono a tipi nel linguaggio sono raggruppati assieme nel package *ast.node.type*
- i sorgenti associati a nodi argomento e left-hand-side di assegnamenti sono raggruppati assieme nel package *ast.node.other*

Infine il *mainPackage* contiene l'entry point del sistema, *parser* contiene tutti i file prodotti da ANTLR in relazione alle grammatiche per il linguaggio e per il bytecode, *Interpreter* contiene l'interprete per il linguaggio bytecode. Nel package *util* ci sono i sorgenti che definiscono varie funzioni ausiliarie e di supporto per la definizione delle strutture dati, funzioni e procedure utilizzate durante le fasi di analisi semantica, analisi degli effetti, code generation, e infine alune strutture dati come *Label*, *Status*, *Environment*, *CGenEnv*.

Nell'esposizione di questa relazione si seguirà lo schema dell'esecuzione delle varie fasi della compilazione, andando ad esporre le scelte implementative per ogni tipo di nodo e andandone a discutere analisi semantica (*scope*, tipi ed effetti) e *code generation*.

(1)

SimpLanPlusVisitorImpl.java

Prima di proseguire passando in rassegna i vari nodi, si considera la classe *SimpLanPlusVisitorImpl.java*. In questa classe devono essere implementati i metodi specificati nell'interfaccia *SimpLanPlusVisitor.java*; solo i metodi strettamente necessari sono stati implementati (non tutti i simboli non terminali della grammatica richiedono un metodo dedicato di visita dell'AST).
Nell'entry point del sistema la classe viene istanziata per poter cominciare la visita dell'*ast*. In particolare *ParseTreeVisitor.java* ha un metodo di visita generico, *visit*, che prende in input la radice (un nodo di tipo block) e visita l'AST fino ad arrivare alle foglie. Una volta partita la visita, vengono dunque generati i vari nodi specificati dalla grammatica e quindi questo punto è già interessante per discutere qualche problema e soluzione riscontrati. Si è deciso di non modificare mai la grammatica data, per cui in varie occasioni questa decisione si è scontrata con la necessità di creare "nodi ad hoc". Un esempio di questo fatto è nel metodo di visita di una dichiarazione di funzione. Sia un blocco normale che funzione richiedono di aggiungere un nuovo *scope* all'ambiente, e siccome la funzione richiede di aprire un nuovo blocco, ci si pone il problema di evitare di aggiungere nell'ambiente uno *scope* vuoto superfluo. Una possibile soluzione sarebbe quella di pensare a due regole di inferenza distinte per il *BlockNode*, una per l'apertura di un blocco normale e una per l'apertura di un blocco di funzione, ma siccome non si vuole modificare la grammatica, si è giunti alla conclusione di settare un *flag* proprio durante la generazione di un *DecFunNode*. Grazie a questo *flag* risulta possibile capire se il blocco che si sta visitando sia un blocco di una funzione e di conseguenza evitare di aumentare il nesting level dell'ambiente corrente in quanto sarà già il *DecFunNode* a preoccuparsene.
Una simile strategia è utilizzata in diversi punti del sistema, come nella visita di una left-hand-side di un assegnamento. Da grammatica, un nodo *Lhs* può essere utilizzato sia a sinistra che a destra di un assegnamento, per cui bisogna capire da quale lato dell'assegnamento si trovi l'*Lhs*. Nel caso in cui il nodo *Lhs* venga visitato come right-hand-side, si passa per un nodo *exp* di tipo *#derExp*. Settando il *flag* durante la visita di un *DerExpNode* è quindi possibile sfruttare la classe *LhsNode* evitando di dover duplicare il codice. Il ragionamento fatto per left-hand-side e right-hand-side è esattamente lo stesso adottato per distinguere una normale chiamata di funzione da una chiamata di funzione fatta visitando un nodo *exp* (nel secondo caso la funzione non può avere tipo di ritorno pari a *void*).
Si passano ora a considerare i vari nodi generati durante la visita dell'albero di sintassi astratta, andando a vedere quali siano le problematiche e le soluzioni adottate, prendendo in considerazione per ognuno di essi l'analisi semantica e la generazione di bytecode.

Node

Si tratta dell'interfaccia che ogni nodo dell'albero implementa. Essa è un'estensione del *Node* del linguaggio *SimpLan* visto a lezione. In particolare:

- Il metodo *toPrint* serve per stampare l'AST una volta verificata la correttezza semantica del codice

- *checkSemantics*, *typeCheck* e *checkEffects* realizzano l'analisi semantica del codice scritto in *SimpLanPlus*
- *codeGeneration* ha lo scopo di generare la serie di istruzioni in bytecode per l'interprete di *SimpLanPlus*
- *getPointLevel* serve per la gestione dei puntatori
- *getStatus/setStatus* viene utilizzata per l'analisi degli effetti

Si noti che *checkSemantics*, *typeCheck*, *checkEffects* e *codeGeneration*, richiedono di effettuare una visita dell'*ast*. Inoltre *checkSemantics* e *checkEffects* creano ricorsivamente una *symbol table* (struttura dati *Environment* nel package *util*, i.e. una lista di *hash map* contenenti un nesting level) durante la loro visita dell'*AST*, in modo equivalente ma per scopi differenti :

- *checkSemantics* controlla che non ci siano degli errori di *scope* durante la visita dell'*AST*, e lascia delle *entry* nei nodi che ne avranno bisogno per le visite seguenti (*Lhs*, *Deletion*, *Call*);
- *checkEffects* ha bisogno di tutta la *symbol table* per poter andare a modificare gli *Status* delle variabili anche in *scope* precedenti.

Infine *codeGeneration* necessita di una versione ridotta di ambiente. Nello specifico, si fa riferimento a *CGenEnv* nel package *util*, ovvero una struttura dati con il minimo indispensabile per generare correttamente il bytecode.

Package ast.node.declaration

In questo package sono presenti le classi che riguardano la generazione di nodi dichiarazione. Nel linguaggio *SimpLanPlus* sono possibili due tipi di dichiarazione: le dichiarazioni di funzioni e di variabili.

DecFunNode

La dichiarazione di funzione è sicuramente un nodo ostico da trattare. Ci si è subito posti il problema di capire quale fosse la regola di portata che si voleva implementare. Conviene quindi discutere subito questo punto nel sistema realizzato:

- *BlockNode*: per i blocchi inline si è deciso di aggiungere uno *scope* all'ambiente corrente e di poter accedere alle variabili globali
- *DecFunNode*: per le funzioni si è deciso di creare un nuovo *scope* in un ambiente ex novo[1] così da non poter accedere agli identificatori di variabili globali; tale decisione è dovuta al fatto che per le variabili globali che si vogliono modificare all'interno della funzione ci sono i puntatori che possono essere passati come parametri; al nuovo ambiente creato per la funzione viene aggiunto anche un ambiente contenente solo le funzioni precedentemente dichiarate così da poter fare ricorsione e richiamare altre funzioni esterne

A questo punto si considera quindi l'analisi semantica di un *DecFunNode*. Nella *checkSemantics* si implementa la regola di portata appena discussa:

- Per prima cosa si controlla di non aver già definito la funzione nello *scope* corrente
- Si procede con la creazione di un nuovo ambiente da zero per evitare di accedere a variabili globali
- Prima di aggiungere un nuovo ambiente coi parametri della funzione, si aggiunge una nuova *hash map* contente gli identificatori e i tipi delle funzioni già dichiarate e visibili
- Si controlla la semantica del body della funzione utilizzando il nuovo ambiente appena creato
- Si rimuove l'ambiente della funzione e si decrementa il nesting level

La *typeCheck* non ha particolari difficoltà, in quanto basta controllare che il tipo della funzione e quello del suo corpo siano concordi. Si noti che i corpi delle funzioni devono contenere almeno un *return statement*.

Per quanto riguarda invece l'analisi degli effetti (*checkEffects*), si ricrea l'ambiente allo stesso modo della *checkSemantics* (dando per scontato che a questo punto della compilazione non ci siano errori di *scope*) e si considera l'algoritmo del punto fisso per poter capire quali siano gli effetti di una funzione ricorsiva.
L'idea alla base dell'implementazione è quella di continuare a modificare in *symbol table*, ad ogni step di algoritmo

[1] Siccome l'ambiente ricreato è nuovo (i.e. con nesting level pari a -1) e da grammatica è possibile definire funzioni dentro altre funzioni, al nuovo ambiente vengono subito aggiunte delle *hash map* vuote in modo da costruire il nuovo ambiente con il giusto nesting level

di punto fisso, gli effetti che la funzione apporta ai suoi parametri formali (quindi si ricicla sempre sulla stesso albero piuttosto che creare tanti alberi di prova). Per fare ciò si utilizzano due liste d'appoggio, *sigma0* e *sigma1*. Inizialmente si settano tutti i parametri della funzione allo stato *READWRITE* [2]e si pone *sigma1* pari a tale lista dei parametri formali:

- Si copiano in *sigma0* gli argomenti (coi relativi status, ciò è importante per fare il confronto una volta tipato il corpo della funzione) salvati in *sigma1*
- Si modificano gli status dei parametri formali della funzione in *symbol table*, richiamando la *checkEffects* sul body della funzione stessa
- Si ricostruisce *sigma1* con i nuovi status ottenuti alla fine dell'analisi degli effetti del *body*
- Si confrontano *sigma0* e *sigma1* e se questi sono diversi, si ripete un ulteriore passo dell'algoritmo, aggiornando *sigma0* e rigenerando *sigma1*, se invece i due sono uguali, abbiamo raggiunto il punto fisso

Alla fine dell'algoritmo gli effetti che la funzione apporta ai suoi parametri formali sono quelli descritti in *sigma1*.

L'ultima questione da discutere per un *DecFunNode* è la generazione di codice. In fase di *checkSemantics* gli offset dei parametri formali e delle variabili locali della funzione sono stati impostati pensando di avere il seguente layout per il record di attivazione:

-	variabili locali
	indirizzo di ritorno
	access link
+	parametri

Si noti che non si salva il valore del vecchio frame pointer prima dei parametri perché esso verrà memorizzato dentro la locazione di memoria dell'access link. In tal modo i parametri stanno sempre a offset positivi (a partire da $+1$) dal frame pointer (che punterà alla cella dell'access link), mentre le variabili locali staranno ad offset negativi (a partire da -2) rispetto al frame pointer.
Oltre a discutere di come organizzare il record di attivazione, è importante anche chiarire la questione delle etichette che si può notare nella *codeGeneration* di un *DecFunNode*. Ci si pone il problema di gestire lo stack correttamente una volta incontrato uno *statement return* nel body della funzione. Il problema è piuttosto spinoso per via del fatto che non si vogliono eseguire eventuali istruzioni scritte dopo un *return*. Pensare di gestire la questione del *return* è scomoda per via del fatto che manca la lista dei parametri della funzione e quindi non si può sapere quante pop eseguire sullo stack. Non solo in un nodo *return* non si conosce il numero di argomenti della funzione, ma non si conosce nemmeno il numero di variabili locali presenti nel corpo della funzione, invece ottenibile in un *DecFunNode*. Di conseguenza si è scelto di inserire una label per il *return* subito dopo la generazione di codice del corpo della funzione, così da poter gestire lo stack tutto in *DecFunNode*. In totale, per una dichiarazione di funzione sono state utilizzate tre etichette:

- Un'etichetta univoca per ogni funzione all'interno di tutto il programma (diversa ovviamente anche tra funzioni con lo stesso nome, ma dichiarate in *scope* diversi)
- Un'etichetta per lo *statement* return, da mettere subito dopo la generazione del codice del body
- Un'etichetta da porre in fondo a tutto il codice generato da un *DecFunNode* per evitare di eseguire codice quando si trova la dichiarazione (il codice va eseguito solo dopo una chiamata di funzione)

(2)

DecVarNode

Sicuramente un *DecVarNode* è più semplice da gestire rispetto ad una dichiarazione di funzione. Una dichiarazione di variabile può avere anche un'inizializzazione, *exp*, per cui il primo passo nella discussione di un *DecVarNode* è capire se ci sia o meno l'inizializzazione, indipendentemente dal fatto che ci si trovi in *checkSemantics*, *typeCheck*, *checkEffects* o *codeGeneration*. Il motivo per cui si sceglie di controllare prima di tutto l'*exp* della dichiarazione stessa è che l'*exp* potrebbe contenere l'id della variabile che si sta dichiarando, non ancora dichiarata, e in quel caso c'è un errore da segnalare. Detto ciò:

- Nella *checkSemantics* l'unico controllo da fare è che la variabile non sia già presente nell'ambiente, mentre la *typeCheck* deve controllare che il tipo dichiarato e il tipo di *exp* siano gli stessi (particolare attenzione va fatta coi puntatori per i quali è necessario non solo verificare il tipo ma anche che abbiano lo stesso *point level* [3])

[2]Per rendere possibile l'utilizzo dei parametri formali senza che questi vengano inizializzati, altrimenti non si potrebbe neanche scrivere una funzione che presi due interi in input ritorna la somma dei due

[3]Un puntatore singolo ha *point level* $= 1$, un puntatore doppio ha *point level* $= 2$, etc.

- Nella *checkEffects* si assegna lo stato *READWRITE* (di default una variabile viene sempre messa a *DECLARED*) solamente se *exp* è presente
- Nella *codeGeneration* se *exp* è presente allora si aggiunge in stack la *codeGeneration* dell'inizializzazione, altrimenti si lascia un posto vuoto al giusto offset nel record di attivazione del blocco o funzione in cui la variabile è dichiarata.

Package ast.node.exp

In *SimpLanPlus.g4* sono specificate le seguenti possibili espressioni: *#baseExp*, *#negExp*, *#notExp*, *#derExp*, *#newExp*, *#binExp*, *#callExp*, *#boolExp*, *#valExp*. In generale la varie espressioni non presentano particolari problemi, per cui i commenti del codice sono già sufficienti, senza contare che si è già discusso [1] di come siano state affrontate *#callExp* e *#derExp* (vedi sezione SimpLanPlusVisitorImpl.java).
I punti più complessi riguardano la *typeCheck* e la *codeGeneration* delle *#binExp*. In particolare in questo tipo di espressioni, l'analisi semantica diviene più difficile non solo perché ci sono diversi operatori da distinguere, ma anche per via dei puntatori. La prima cosa che si nota in *typeCheck* è che dopo aver ottenuto i tipi dei due operandi si va subito a controllare se questi siano puntatori o meno. Tale struttura del codice è dovuta al fatto che si è presa la decisione di poter applicare ai puntatori (distinguibili tramite il campo *pointLevel* presente in *IntTypeNode* e *BoolTypeNode*) solo alcuni operatori binari, i.e. il ! = e il ==. Tuttavia, non basta controllare che i due operandi siano due puntatori, ma è necessario anche controllare che essi abbiano lo stesso *pointLevel* e assicurato ciò, è possibile applicare l'operatore. Per quanto riguarda invece la *codeGeneration* di una *#binExp* va sottolineato il fatto che, al fine di non perdere risultati intermedi di qualche calcolo della *codeGeneration*, va fatta una push sullo stack tra la *codeGeneration* dell'operando di sinistra e quello di destra.
L'ultima espressione che richiede particolare attenzione è la *codeGeneration* della *#newExp*, perché essa si occupa di inizializzare un puntatore. Al fine di spiegare come venga gestita l'inizializzazione di un puntatore, si considerino i seguenti esempi:

```
^int p = new int;
^^^int q = new ^^int;
```

Il primo è un normale puntatore ad un intero, dunque con una dereferenziazione si raggiunge subito il valore intero nell'heap, mentre il secondo è un triplo puntatore e al fine di inizializzarlo correttamente è necessario che la right-hand-side sia una new con 3-1 = 2 accenti circonflessi (se non si fosse scritto così, ci sarebbe stato un errore di tipo). Col triplo puntatore è necessario allocare due celle di memoria nell'heap per i livelli di dereferenziazione successivi, e una terza contenente l'intero. Il problema che ci si è posti qui, è quello di decidere cosa poter fare con i puntatori ottenuti dalle dereferenziazioni di q (mentre la tripla dereferenziazione è un intero, la doppia e la singola sono puntatori). L'utilizzo dei puntatori del linguaggio è dunque da intendere come segue:

- Al momento della dichiarazione di q si riporta in *symbol table* il *pointLevel* del puntatore, ovvero 3
- Ogni volta che il puntatore è utilizzato come right-hand-side si controlla con che *pointLevel* viene dereferenziato

Detto ciò, la *codeGeneration* di una *#newExp* prevede di incrementare il valore dell'heap pointer, per poter quindi allocare un intero o un booleano (entrambi hanno la stessa occupazione di memoria di un byte). Allocato l'intero/il booleano con valore di default pari a 0, si allocano più indirizzi contigui nell'heap in caso il puntatore abbia *point level* maggiore di 1, per esempio, la memoria associata al puntatore q nell'heap dell'esempio precedente è la seguente:

0x1	0
0x2	0x1
0x3	0x2

In stack, q avrà associato l'indirizzo 0x3, grazie al fatto che il valore dell'heap pointer viene caricato sul registro $\$a0$ così da poter essere assegnato a q.

Package ast.node.other

In questo package sono presenti i sorgenti che riguardano la visita di un nodo che è un parametro di una funzione o di nodi generati a partire dal non terminale *Lhs* della grammatica. Mentre *Argnode.java* non ha particolari note, più complessa è la gestione di un nodo *Lhs* (tenendo conto che in questa classe si gestisce anche

il comportamento previsto per un *DerExpNode*).
Sebbene la *checkSemantics* di un nodo *Lhs* si limiti alla ricerca in *symbol table* dell'identificatore (se la ricerca fallisce si segnala un errore di utilizzo di un identificatore non dichiarato), la *typeCheck* e la *checkEffects* sono più articolate per via dei controlli fatti per un corretto utilizzo de puntatori[4].

Il problema che si vuole risolvere coi puntatori è quello di implementare la semantica esposta nel paragrafo precedente, per cui bisogna capire quante volte un puntatore venga dereferenziato. Innanzitutto va sottolineato che durante la visita dell'albero si contano quanti "^" ci siano a destra dell'identificatore (si trovano a sinistra soltanto in dichiarazione e nel caso di un *new*), per cui vengono fatti dei controlli tra il *pointLevel* di un identificatore salvato in *symbol table* e il numero di dereferenziazioni effettuate in un nodo *Lhs* . Ovviamente si possono verificare una gran quantità di errori, non solo per quanto riguarda i puntatori ma anche per le variabili normali:

- *typeCheck*: considerando che la *typeCheck* è la seconda passata dell'*AST*, di sicuro l'*LhsNode* che si sta analizzando sarà per forza presente in *symbol table* (si noti la struttura di *Test.java*, se si verificano errori in *checkSemantics* non si procede con la *typeCheck*) per cui per una variabile normale basta ritornare il tipo indicato in *symbol table*, invece per un puntatore bisogna anche controllare che il numero di "^" sia corretto
- *checkEffetcs*: il problema più grande si verifica nel caso in cui si abbia una un puntatore in right-hand-side, in tal caso è proprio questo il punto in cui bisogna controllare che non si stia accedendo ad un puntatore con status diverso da *READWRITE*, mentre per una left-hand-side bisogna assicurarsi di non dereferenziare un puntatore in status *DELETED*

L'*Lhs* è un punto interessante anche per la *codeGeneration*, in quanto è proprio questa la classe in cui è necessario capire se si stia facendo riferimento ad un identificatore dichiarato localmente o globalmente. Vista la regola di portata e discussa la semantica dei puntatori, sia nel caso del blocco che della funzione, un riferimento a variabili locali (e parametri formali nel caso di funzione) viene calcolato grazie all'offset scritto nell'ambiente, per cui a partire dalla cella contenente l'access link è possibile recuperare il valore della variabile. Se la variabile viene dereferenziata, prima di recuperare il valore bisogna eseguire delle salti nell'*heap* a partire dalla cella di memoria nello stack in cui essa è memorizzata (per comodità, se siamo in una left-hand-side, si carica in $*al* l'indirizzo di memoria della variabile, nel caso di una right-hand-side, invece si salva il valore contenuto a quell'indirizzo). Nel caso in cui si debba recuperare una variabile globale (dunque ci si trova in un blocco inline, dato che dalle funzioni non si può accedere alle variabili globali), bisogna andare a leggere anche il *nesting level* corrente dal *CGenEnv* in modo da risalire la catena statica il numero giusto di volte.

Package ast.node.*statement*

Si prendono ora in considerazione l'analisi semantica e la produzione di *bytecode* per gli *statement*. La prima scelta implementativa da sottolineare per quanto riguarda gli *statement* è sulla *typeCheck*. Durante la passata dell'*AST* da parte della *typeCheck*, alcuni nodi non ritornano dei valori interi o booleani per cui ci si è posti il problema di creare un tipo oltre a int/bool. L'utilizzo di un tipo *void* non va bene in quanto già utilizzato per le funzioni e quindi si è giunti alla definizione di un nodo particolare, *NullTypeNode*, da utilizzare come tipo di ritorno per alcuni *statement* come la *print* (ma non solo *statement*, il *NullTypeNode* torna utile anche per altri nodi come le dichiarazioni o in tutti quesi casi in cui non si può definire un tipo di ritorno tra int/bool/*void*). In altre parole tutti quei costrutti per cui la propagazione di tipo non è interessante torna utile il *NullTypeNode*.

AsgNode

La logica utilizzata per scrivere analisi semantica e generazione di codice di questo nodo è simile a quella già vista per il *DecVarNode*, l'unica differenza apprezzabile è che in un assegnamento si potrebbe avere in *left-hand-side* un puntatore cancellato. Abbiamo scelto inoltre di permettere al programmatore di reinizializzare puntatori cancellati.

BlockNode

Durante la discussione di *DecFunNode* erano già state anticipate alcune problematiche riguardanti i blocchi *inline*, come il fatto di poter far riferimento a variabili globali e che entrando in un blocco si crea un nuovo *scope*.
Per quanto riguarda la *checkSemantics* nel *BlockNode*, l'unica accortezza da prendere è quella di controllare che

[4]Una nota va fatta per le funzioni in quanto esse non possono mai essere usate come left-hand-side (SimpLanPlus non è un linguaggio di ordine superiore).

il blocco non sia associato ad una dichiarazione di funzione (altrimenti si rischia di aumentare il nesting level in modo errato).
Per la *typeCheck*, invece, si vuole sottolineare il problema dovuto al fatto che potrebbero anche esserci più *return statement* in una funzione/blocco per via dei costrutti if-then-else, o per via del fatto che sintatticamente parlando è possibile scrivere più *return* in sequenza (si noti però che in *BlockNode* il *return* non è obbligatorio).

Detto ciò, prima viene eseguita la *typeCheck* per le dichiarazioni del blocco, poi la *typeCheck* per gli *statement* raccogliendo in una lista i tipi risultanti dei *RetNode* e *IteNode* (solo se diversi da *NullType* e completi, ovvero ogni if-then ha il suo ramo else). Il prossimo passo è controllare che tutti i tipi raccolti nella lista siano concordi e in caso negativo segnalare un errore. Si noti che un blocco *inline* può non contenere *return statement*, e quindi essere di tipo *NullType*, mentre un blocco di funzione avrà sempre tipo in {int, bool, void}.

Così come la *checkEffects*, anche la *codeGeneration* di un *BlockNode* richiede di distinguere tra blocco inline e blocco di funzione. Per quanto riguarda la *codeGeneration*, il record di attivazione di un blocco inline è uguale a quello di una funzione (ma senza parametri e *return* address) e le etichette rivestono un ruolo importante per via del fatto che se un costrutto if-then-else contiene uno *return statement*, allora bisogna fare attenzione da quale blocco dovrà riprendere l'esecuzione. Si consideri per esempio la funzione seguente in SimpLanPlus:

```
{
    int fib(int a){
        if(a <= 1){
                return a;
        }
        else{
                return fib(a-1) + fib(a-2);
        }
    }
        print fib(10);
}
```

chiaramente il *return* di uno dei due branch tra then ed else deve ritornare l'esecuzione al blocco della funzione fib e non ai blocchi dei rami then ed else. Grazie alle label e ad un istruzione di salto nel *RetNode* è possibile risolvere questo problema. Altro problema if-then-else

```
{
    int f(){

        if (false) return 1;
        else if (false) return 2;
             else if (false) return 3;
                  else if (false) return 4;
                       else return 5;
    }
    print f();
}
```

In questo caso i rami if-then-else sono completi, quindi l'esecuzione ritornerà correttamente 5, ma nel caso in cui l'ultimo else non fosse presente, la funzione ha bisogno di un altro *return statement* per passare il controllo di tipo.

CallNode

In questo nodo viene gestito anche il caso in cui la chiamata di funzione sia un'espressione e non uno *statement*, dunque i nodi di riferimento sono *CallExpNode* e *CallNode*. La parte interessante di una chiamata di funzione è l'analisi degli effetti, mentre:

- La *checkSemantics* si limita a controllare che la funzione sia stata dichiarata prima di poter essere utilizzata
- La *typeCheck* invece controlla che i parametri attuali corrispondano in numero e tipo ai parametri formali della dichiarazione (oltre a controllare che per un *CallExpNode* il tipo di ritorno sia int o bool)
- La *codeGeneration* carica sulla pila i parametri della funzione (ma senza salvare prima il frame pointer [2]) e compie il salto alla label della funzione.

Nella *checkEffects* ci sono diversi punti critici da affrontare.
Il primo problema è la gestione dei puntatori: considerando le scelte implementative fatte per la regola di

portata, vogliamo che gli indirizzi nell'heap possano subire modifiche anche dall'interno del corpo di una funzione. Considerando che nel nostro linguaggio abbiamo soltanto passaggi per valore, durante una chiamata di funzione che prende in input un puntatore, verrà creata una copia dell'indirizzo contenuto nel puntatore passato nel AR della funzione. Quindi a livello di bytecode, le operazioni (e.g. *Deletion*) fatte sul parametro formale, non possono essere riportate sul puntatore passato alla funzione. Per questo si analizzano gli effetti della funzione per aggiornare lo stato dei puntatori dati in input, e intervenire nel caso ci siano errori invisibili all'interprete. Da grammatica, le espressioni[5] che possono, in più passi di derivazione, essere dei puntatori sono la *#baseExp* (dalla quale si estrapola l'eventuale Id del puntatore contenuto) e la *#derExp*. Il primo controllo da fare quello di non aver passato nessun identificatore con status *DELETED/ERROR*, dato che si potrebbe avere come parametro di una funzione una chiamata di funzione che va a cancellare un puntatore.

```
{
    int f(^int q){
        delete a;
        return 1;
    }

    int g(^int r, int a){
        return 0;
    }

    ^int p = new int;
    g(p, f(p));
}
```

In questo caso, sia **p** che **f(p)** sarebbero parametri validi da passare a **g**, ma se passati in contemporanea porterebbero ad avere **p** *DELETED*, quindi serve un ulteriore controllo dopo aver valutato tutti i parametri attuali.

Il prossimo passo è quello di applicare ai puntatori appena controllati gli effetti associati alla funzione, facendo attenzione ad associare il parametro attuale con il giusto parametro formale, per cui dalla *symbol table* si ricava la lista di status associati ad ogni parametro formale e si applica l'operatore *sequenza* con gli status dei parametri attuali (*Seq(actualParam, formalParam)*). In questo punto, si raccolgono gli stati finali dei parametri di tipo puntatore in un dizionario per poter gestire il problema degli aliasing. Siccome c'è la necessità di confrontare tra loro gli status finali degli alias, l'approccio utilizzato per risolvere il problema è quello di costruire un dizionario che associ ad un parametro attuale una lista di alias. Tale lista avrà lunghezza minima pari ad 1, nel caso in cui non ci sia aliasing. Grazie a questo dizionario è dunque possibile eseguire l'operatore *par* tra gli stati finali di ogni alias associato ad un parametro attuale e poter segnalare eventuali errori.

DeletionNode

Lo *statement delete* è pensato per eliminare il collegamento tra il puntatore nello stack e l'oggetto puntato nell'heap. Di conseguenza la parte interessante di questo node si vede nella *checkEffects*. In *checkSemantics* e *typeCheck* ci si assicura solamente di richiamare la *delete* su un identificatore presente in *symbol table* e che tale identificatore sia effettivamente un puntatore.
Nella *checkEffects* si recupera dall'ambiente il puntatore e poi si esegue l'operatore *sequenza*[6]con lo status *DELETED* e si va a rimpiazzare l'entrata della *symbol table* con il nuovo status per il puntatore. Il punto importante da notare è che così facendo qualsiasi tentativo di accedere a tale puntatore sarà rilevato come errore. Tuttavia, va notato che un puntatore *DELETED* può ritornare allo stato *READWRITE* se inizializzato nuovamente.
Nella *codeGeneration* invece dopo aver recuperato l'identificatore del puntatore grazie all'access link, si marca la sua cella nello stack con il valore -10000, che indica che il puntatore non ha più riferimenti all'heap.

IteNode

Con gli *IteNode* si fa riferimento agli statment if-then-else e if-then. I problemi riscontrati con questo nodo sono dovuti solamente alla *checkEffects*, nel caso in cui lo *statement* abbia anche un ramo else. Dall'analisi degli effetti di un if-then-else, è necessario creare due ambienti, in ognuno analizzare gli effetti delle rispettive istruzioni, e avere infine nell'ambiente in uscita gli status massimi per ognuno degli identificatori in symbol table. Per fare ciò, a partire dall'ambiente di partenza dell' *IteNode*, bisogna creare due copie: una sarà modificata dalla *checkEffects*

[5]Da sintassi, i parametri attuali in chiamata di funzione sono delle espressioni.
[6]Il codice di questo operatore, così come con lo status *max* e con lo status *par*, si trovano nel package *util*, nel sorgente *SimpLanPlusLib.java*

del ramo then, mentre l'altra sarà modificata dalla *checkEffects* del ramo else. Siccome Java passa un riferimento per le strutture dati e non le copia, è risultato necessario creare il metodo *cloneEnvironment* presente nel package *util*, sorgente *SimpLanPlusLib.java*. Grazie a questo metodo è possibile ottenere un ambiente identico a quello di partenza ma senza riferimenti tra loro, così una volta stabilito il massimo tra l'ambiente ottenuto col ramo then e l'ambiente ottenuto col ramo else è possibile modificare l'ambiente iniziale implementando correttamente l'analisi degli effetti per l'*IteNode*.

PrintNode

L'unica nota per questo nodo è che può stampare anche i puntatori e quindi gli indirizzi di memoria a cui puntano.

ReturnNode

L'analisi semantica e la produzione di bytecode di questo nodo sono già stati in parte discussi nel *BlockNode* e le due funzioni interessanti per questa classe sono *typeCheck* e *codeGeneration*.
Nella *typeCheck* va tenuto in considerazione che il *return* potrebbe essere di una funzione di tipo *void* e in più viene segnato errore il caso in cui si tenti di ritornare un puntatore.
Nella *codeGeneration*, invece, è scritto del bytecode che va a completare quello scritto nel *DecFunNode* o nel *BlockNode*, a seconda del tipo di blocco in cui il *return* si trova. Nella *codeGeneration* del *RetNode* non si fa altro che caricare il registro $\$rv$ con il valore di ritorno e si salta alla label di fine blocco.

Package ast.node.type

In questo package sono racchiuse le definizioni dei tipi che i nodi dell'albero di sintassi astratta possono avere. L'unica nota interessante è che per definire i puntatori si setta la variabile *pointLevel* degli *IntTypeNode*/*BoolTypeNode*, mentre per impostare lo status ai fini dell'analisi degli effetti si va modificare il campo *Status* del nodo (si noti che le operazioni utili per manipolare *pointLevel* e *Status* di nodi sono quelle descritte nell'interfaccia *Node*).

Package Interpreter

L'interprete per il bytecode prodotto con la *codeGeneration* è un'estensione di quello di *SimpLan*. La differenza principale con esso è l'aggiunta dei registri $\$a0$, $\$t0$, $\$rv$ e $\$al$, per cui sono stati modificati anche il lessico e la sintassi specificati nel file *SVM.g4*. In particolare sono state modificate tutte le operazioni che lavoravano direttamente con il top dello stack, quindi al posto di caricare/scaricare dal top dello stack caricano/scaricano dal registro $\$a0$.

Folder test

Nella cartella test sono presenti più di 40 file che esaminano molte situazioni interessanti a livello di compilazione, e servono perlopiù a verificare l'esito della compilazione di un certo codice. Ogni test è formato da:

- Descrizione del problema trattato nel codice sottostante
- Pass/Fail, in base alla correttezza del codice
- Output: che contiene sia gli errori riscontrati in fase di compilazione che gli output corretti del codice (generabili da *print*)
- Codice sorgente da eseguire, con eventuali commenti