Esercitazione 4

Componenti del gruppo

Colledan Andrea - andrea.colledan@studio.unibo.it Berti Matteo - matteo.berti11@studio.unibo.it

Obiettivi

- Definire un linguaggio bytecode per eseguire programmi nell'estensione di SimpleStaticAnalysis definita nell'Assegnamento 3 e codificare le istruzioni del linguaggio ad alto livello.
- Implementare l'interprete per il bytecode.
- Compilare ed eseguire i programmi del linguaggio ad alto livello.

Installazione ed utilizzo

Il progetto è all'interno della cartella **codegeneration**, che può essere importata direttamente da Eclipse tramite: "File > Open Projects from File System... > Directory..." click singolo sulla *cartella* **codegeneration** poi "OK", ed infine "Finish".

Il file da eseguire per l'esecuzione del compilatore è "codegeneration/src/compiler/Compiler.java" all'interno del quale è possibile specificare i file .txt da compilare. Di default vengono eseguiti i file contenenti codice e funzioni richieste dagli assignment 3 e 4. Per testare codice a piacimento è consigliabile modificare il file "codegeneration/tests/multipurpose.txt", che viene sempre testato dopo i file di default.

Per testare l'**esercizio 1 dell'assignment 3** (in cui gli statement delete sono disabilitati) è sufficiente settare a `false` il campo `allowDelete` in `Compiler.java`. Tale campo è di default `true` (si veda la sezione **Analisi semantica** per maggiori dettagli).

POSSIBILI ERRORI:

- "ANTLR Tool version X.X used for code generation does not match the current runtime version Y.Y" Se il plugin ANTLR4 è già presente è possibile assicurarsi di avere in esecuzione la corretta versione di Antlr tramite: "Click destro sul progetto **codegeneration** > Build Path > Configure Build Path... > [a sinistra] ANTLR 4 > [espandere] Tool > Enable project specific settings". Nella sezione "Distributions" click su "Add" per importare la versione 4.6 disponibile all'interno del progetto: "lib/antlr-4.6-complete.jar", click sulla spunta della versione 4.6 ed infine "Apply and Close".
- [icona punto esclamativo nel progetto] "Errors exist in required projects(s): codegeneration" Se la Java System Library in qualche modo non combacia con quella di default nel proprio workspace Eclipse è possibile procedere

come segue: "Click destro sul progetto **codegeneration** > Build Path > Configure Build Path... > [a sinistra] Java Build Path > [Tab] Libraries > Edit..." click su "Workspace default JRE" per quella di default, oppure su "Alternate JRE:" per importarne una a proprio piacimento. Infine "Apply and Close".

Scelte progettuali

Generali

Il processo di compilazione ed esecuzione dei programmi è stato prevede tre fasi.

La fase di **analisi statica** (classe `Compiler`, metodo `analyze(String fileName, boolean allowDeletion)` prevede l'apertura del codice, il suo parsing e la generazione del relativo albero di sintassi astratta. In seguito avviene la verifica della correttezza del programma, in tre parti:

- Nella fase di **analisi semantica** viene verificata la correttezza basilare del programma. In una prima passata dell'albero di sintassi astratta viene verificato che:
    - Tutti gli utilizzi e le eliminazioni di funzioni e variabili siano preceduti dalle rispettive dichiarazioni, secondo le regole di scoping statico.
    - I parametri formali e attuali delle funzioni coincidano in numero al momento della loro invocazione.
    - Eventuali parametri passati per variabile corrispondano, al momento dell'invocazione, con singoli identificatori di variabili esistenti, e non con generiche espressioni.
    - Non vi siano dichiarazioni multiple di variabili e funzioni nello stesso scope (solo se la deletion è disattivata, vedi in seguito)
- Nella fase di **type checking** viene verificato che i tipi delle variabili del programma siano compatibili con le operazioni svolte su di loro. Viene inoltre verificato che, al momento dell'invocazione, i parametri attuali di una funzione abbiano lo stesso tipo di quelli formali.
- Nella fase di **analisi comportamentale** viene verificata la consistenza del programma per quanto riguarda le operazioni di dichiarazione, eliminazione, ridichiarazione ed utilizzo di variabili e funzioni. Questa analisi viene svolta attraverso l'astrazione di **tipo comportamentale** del programma (vedi in seguito).

Al termine dell'analisi statica, viene restuito l'albero di sintassi astratta, se esso è privo di errori, altrimenti `null`.

La fase di **compilazione** (classe `Compiler`, metodo `compile(Block ast, String compiledFileName)`) prevede la generazione del codice intermedio corrispondente all'albero di sintassi astratta. Tale codice è salvato in forma testuale in un percorso apposito (`generated_code/[nomefile].wtm`).

La fase di **esecuzione** (classe `Compiler`, metodo `executeWTM(String code)`), infine, prevede la verifica del codice intermedio e la sua esecuzione sulla World Turtle Machine (WTM, vedi in seguito).

Ambiente

Classe `Environment`

L'ambiente prevede due **symbol table distinte** per variabili e funzioni (`variableSymbolTable` e `functionSymbolTable`). Entrambe sono implementate attraverso liste di hash table.

Oltre a queste è presente anche un **buffer di parametri** (`parameterBuffer`), ovvero una lista volta a contenere le entry relative ai parametri di una funzione mentre vengono dichiarati, affinché possano essere aggiunti in blocco come variabili locali nello scope che viene aperto successivamente dal corpo della funzione (il buffer realizza una sorta di "dichiarazione anticipata" dei parametri in uno scope che ancora non esiste).

Infine, sono presenti campi utili alla generazione del codice, che tengono conto del **numero di etichette** utilizzate globalmente (`labels`) e del **numero di variabili locali** dichiarate in ciascuno scope del programma (la lista di interi `localVars`).

Le strutture dati che realizzano l'ambiente sono **massimamente generiche** e gestiscono oggetti di tipo `BaseSTEntry`, ovvero symbol table entry minimali che contengono solo id e livello di annidamento della variabile/funzione dichiarata. Ciascuna fase di analisi e compilazione usa poi specifiche **sottoclassi** di `BaseSTEntry` (e.g. `BehavioralSTEntry` per l'analisi comportamentale) e **metodi ad hoc** per leggere e manipolare l'ambiente secondo le proprie esigenze specifiche. I metodi di `Environment` sono pertanto suddivisi in sezioni diverse a seconda della fase della compilazione che ne fa uso.

Analisi semantica

Sottoclassi di `Node`, metodo `checkSemantics(Environment env, boolean allowDeletion)`

L'analisi semantica si limita a verificare le proprietà sopracitate. Come anticipato, se l'analisi semantica viene avviata con `allowDeletion = true`, il controllo delle dichiarazioni multiple viene sospeso e rimandato alla successiva fase di analisi comportamentale. Se invece `allowDeletion = false` (è questo il caso dell'Esercizio 1 dell'Assignment 3), i delete statement vengono considerati nulli e ogni caso di dichiarazione multipla viene considerato semanticamente incorretto.

L'analisi semantica non riporta gli errori durante la visita dell'albero, bensì li accumula in una lista di `SemanticError` che viene restituita al termine della visita dell'albero di sintassi astratta.

Type checking

Sottoclassi di `Node`, metodo `checkType(Environment env)`

Il type checking è stato implementato come visto a lezione. La visita di un nodo dell'albero di sintassi astratta attraverso `checkType` restituisce un valore dell'enumeratore `Type` che può essere `INT`, `BOOL` o `VOID`. Se un sottoalbero dell'albero di sintassi astratta non tipa correttamente, `checkType` ritorna su di

esso `null`.

Analisi comportamentale

Sottoclassi di `Node`, metodo `checkBehavioralType(Environment env)`

L'analisi comportamentale è la parte più sostanziosa dell'analisi statica svolta dal compilatore. In essa viene verificato che l'istruzione `delete` venga utilizzata correttamente, ovvero che:

- Non vi siano variabili dichiarate più volte (senza un'eliminazione tra le due dichiarazioni).
- Non vi siano variabili eliminate più volte (senza una ridichiarazione tra le due eliminazioni).
- Non vi siano utilizzi di variabili eliminate (e non ridichiarate).

Tipi comportamentali

Questa verifica viene portata a termine attraverso l'astrazione di **tipo comportamentale** (classe `BehavioralType`) del programma, che funge da "sommario" delle operazioni svolte da un programma sulle sue variabili.

Un tipo comportamentale è descritto da due campi:

- Una lista di **azioni** (lista `actions` di `ActionEntry`), ciascuna caratterizzata da un tipo (`ActionType`) tra dichiarazione (`DEC`), ridichiarazione (`RED`), lettura/scrittura (`RW`) ed eliminazione (`DEL`) e dall'oggetto dell'azione (nella pratica, un riferimento alla symbol table entry dell'oggetto che viene dichiarato, ridichiarato, letto/scritto o eliminato).
- Un flag che indica se il tipo è **provvisorio** o definitivo (`tentative`). Un tipo è provvisorio quando deriva dall'analisi di operazioni il cui tipo non è ancora definito. Nella pratica, l'unico caso in cui si verifica questo scenario è quello in cui una funzione ricorsiva invoca se stessa. L'invocazione ha in tal caso tipo vuoto, ma provvisorio.

Nello scrivere un tipo comportamentale, lo rappresentiamo come una lista di tipi di azione applicati ad oggetti (i.e. variabili o funzioni) del programma. Ad esempio, il seguente codice

```
{
    int x = 42;
    f(var int n) {
        print n;
        delete n;
    }
    f(x);
    int x = 24;
    print x;
}
```

stampa prima `42`, poi `24` e ha tipo comportamentale:

```
{DEC(x:1),DEC(f:2),RW(x:1),DEL(x:1),RED(x:3),RW(x:3)}
```

Ciascun oggetto è denotato dal suo identificatore, seguito dall'id unico della sua entry nella symbol table. Due oggetti sono uguali *se e solo se* hanno lo stesso id unico. Due oggetti distinti possono avere lo stesso id, ad esempio nel caso di ridichiarazione in scope più interni.

Consistenza

Un tipo comportamentale è **consistente** quando rappresenta una serie di operazioni non problematiche dal punto di vista di eliminazioni e ridichiarazioni. Più formalmente, un tipo comportamentale è consistente quando:

- *Nessuna* lettura/scrittura `RW(x:id)` al suo interno è preceduta da un'eliminazione `DEL(x:id)` dello stesso oggetto. Questo verifica che nessuna variabile venga usata dopo essere stata eliminata. Nel caso simile (e altrettanto corretto) di ridichiarazione tra eliminazione ed utilizzo abbiamo oggetti diversi (ma con lo stesso identificatore) per `RW` e `DEL`, per cui questo requisito di correttezza non comporta falsi negativi.
- *Ogni* ridichiarazione `RED(x:id)` al suo interno è preceduta dall'eliminazione dello stesso *identificatore* (non lo stesso *oggetto*) `DEL(x:id')`. Questo verifica che ogni identificatore sia ridichiarato solo previa eliminazione della variabile/funzione precedentemente associata ad esso. La ridichiarazione di un identificatore viene infatti aggiunta al tipo solo se al momento della dichiarazione è già presente, nella symbol table, un'associazione per esso.
- *Nessuna* eliminazione `DEL(x:id)` sia preceduta da un'altra eliminazione `DEL(x:id)` dello stesso oggetto. Questo verifica che nessuna variabile sia eliminata pià volte. Nel caso simile (e altrettanto corretto) di un identificatore eliminato una volta, ridichiarato, ed eliminato una seconda volta, abbiamo oggetti diversi nella prima e seconda eliminazione, per cui questo requisito di correttezza non comporta falsi negativi.

NOTA: è evidente come con questi requisiti l'eliminazione di una variabile nello scope corrente *non* sollevi lo shadowing di eventuali variabili con lo stesso nome in scope più esterni. Questa scelta progettuale è dovuta alla difficoltà di controllare l'accesso a variabili precedentemente shadowed in maniera statica.

Un programma passa l'analisi comportamentale se nel suo complesso ha un tipo comportamentale consistente.

Incremento dei tipi comportamentali

Tipi comportamentali di base vengono assegnati in modo ovvio alle istruzioni fondamentali di dichiarazione (`DEC` o `RED`), utilizzo (`RW`) ed eliminazione (`DEL`) di funzioni e variabili. Per tipare costrutti più complessi (e.g. espressioni con più variabili, blocchi di più statement etc.) si ricorre all'**incremento** di un tipo comportamentale con un altro tipo comportamentale (classe `BehavioralType`, metodo `incrementType(BehavioralType target)`). L'incremento prevede l'aggiunta, ad una ad una, delle azioni del secondo tipo a quelle del primo. Dopo ogni aggiunta, si **testa la consistenza** del tipo ottenuto (metodo `isLastConsistent()`). Il risultato dell'incremento è il risultato del test di consistenza, ovvero un oggetto di tipo `Consistency` contenente l'esito del test (che può essere `OK`, in caso di tipo consistente, oppure `DEC/RED`, `DEL/RW` o `DEL/DEL` in caso di inconsistenze) e gli eventuali identificatori causa di inconsistenza. Se a

un certo punto dell'incremento l'aggiunta di un'azione rende inconsistente il tipo, l'operazione di incremento termina con un risultato di consistenza negativo.

Pulizia dei tipi comportamentali

Entità ad alto livello quali blocchi e corpi di funzioni esibiscono comportamenti locali che *non* hanno alcun impatto su come il loro comportamento viene percepito all'esterno. Si prenda per esempio il seguente programma:

```
{
    int x = 30;
    {
        int y = 1;
        print x+y;
    }
    {
        print x;
    }
}
```

Nonostante i due sotto-blocchi abbiano tipi comportamentali diversi (il primo `{DEC(y:2),RW(x:1),RW(y:2)}` e il secondo `{RW(x:1)}`), agli occhi del blocco più esterno la dichiarazione di `y` è del tutto indifferente, in quanto il suo scope è limitato al primo sotto-blocco. Al fine di eliminare operazioni strettamente locali dal tipo dei blocchi, alla fine dell'analisi comportamentali degli stessi (e solo se il risultato è consistente) viene effettuata una **pulizia del tipo** (classe `BehavioralType`, metodo `cleanType()`). La pulizia prevede la rimozione dal tipo comportamentale del blocco di tutte le operazioni che hanno come oggetto variabili locali, di modo che queste non siano visibili durante l'analisi dei blocchi più esterni.

Tipo del costrutto If-then-else

In caso di if-then-else, imponiamo che il ramo then e il ramo else siano **compatibili**. Due blocchi si dicono if-then-else compatibili (classe `BehavioralType`, metodo `isITECompatible(BehavioralType other)`) quando sono entrambi consistenti ed eseguono le stesse operazioni sugli stessi oggetti, a meno di ordine e numero delle operazioni. Nel caso di rami compatibili, l'if-then-else tipa col tipo comportamentale della condizione, incrementato col tipo comportamentale del ramo then.

Due rami non compatibili possono essere accettati qualora uno dei due sia **provvisorio**. Questo corrisponde a uno scenario in cui nel ramo then o nel ramo else sono presenti una o più chiamate ricorsive a una funzione che circonda l'if-then-else. In tal caso lo statement if-then-else tipa col tipo provvisorio ottenuto incrementando il tipo comportamentale della condizione col tipo comportamentale del ramo non provvisorio, se c'è ne uno, altrimenti del ramo else. La provvisorietà viene risolta in una seconda passata del corpo della funzione (vedi in seguito).

Tipi parametrici

Il tipo comportamentale delle funzioni è un'estensione del tipo comportamentale visto finora nota come **tipo comportamentale parametrico** (classe `ParametricBehavioralType`, sottoclasse di `BehavioralType`). Oltre alla consueta lista di azioni, esso prevede una lista degli identificatori dei parametri della funzione. I tipi parametrici devono ubbidire agli stessi requisiti di consistenza dei tipi comportamentali regolari.

Ciò che distingue i tipi parametrici da questi ultimi è la possibilità di essere **istanziati** con un vettore di riferimenti per ottenere un tipo comportamentale regolare (classe `ParametricBehavioralType`, metodo `instantiateType(List<BaseSTEntry> references)`). Se la funzione prevede il passaggio per variabile di certi argomenti, l'istanziazione consiste nel sostituire tutte le occorrenze di ogni tale parametro formale preceduto da var col rispettivo parametro attuale contenuto in `references`. L'istanziazione è eseguita nel momento in cui una funzione è invocata, e può compromettere la consistenza del tipo della funzione.

Funzioni ricorsive

Nell'analisi di una dichiarazione di una funzione (classe `FunctionDeclaration`, metodo `checkBehavioralType(Environment env)`), se questa invoca se stessa all'interno del suo stesso corpo è impossibile tipare correttamente l'invocazione, in quanto il tipo comportamentale della funzione stessa non è ancora noto. In tal caso, l'invocazione tipa con tipo comportamentale vuoto e **provvisorio**. La provvisorietà viene propagata, durante l'incrementazione, all'intero corpo della funzione, che viene inserita nella symbol table col tipo comportamentale (provvisorio) risultante. In seguito il corpo della funzione viene analizzato nuovamente , questa volta avendo a disposizione la funzione che sta venendo dichiarata, e il tipo comportamentale restituito è definitivo. Non è difficile convincersi che il tipo risultante è *effettivamente* il tipo della funzione

NOTA: la mutua ricorsione *non* è supportata.

Compilazione

Sottoclassi di `Node`, metodo `generateWTMCode(Environment env)`

Il **bytecode** utilizzato per la generazione del codice intemedio è del tutto analogo a quello visto a lezione. Le uniche differenze sono una sintassi diversa per l'istruzione `top` (per salvare la cima dello stack nel registro `$r1` scriviamo `top $r1` anziché `$r1 <- top`) e l'aggiunta di tre istruzioni primitive:

- `csig`, che permette di effettuare il cambio di segno di un registro. La sintassi di `csig` è `csig $r1 $r2` e la semantica è quella della scrittura di `-$r2` in `$r1`.
- `print`, che permette di stampare il contenuto di un registro. La sintassi di `print` è `print $r1` e la semantica è quella di stampa a schermo dell'intero contenuto in `$r1`.
- `halt`, senza argomenti, che interrompe la computazione.

NOTA: Le invarianti riguardanti lo stato dello stack e il contenuto di `$a0` prima e dopo la valutazione di espressioni viste a lezione valgono sempre anche per il nostro codice intermedio.

Tipo bool e operazioni booleane

I valori di tipo bool sono codificati nel codice intermedio à la C, ovvero traducendo `false` in `0` e `true` in `1`. Le operazioni logiche di **congiunzione e disgiunzione** sono realizzate con etichette e branching sfruttando le proprietà di **short-circuiting** delle reispettive operazioni. L'espressione `e1 && e2` viene ad esempio tradotta in:

```
...
e1.generateWTMCode(env);
li $t1 0
beq $t1 $a0 SHORTCIRCUIT    //se e1 viene valutata false,
cortocircuita
e2.generateWTMCode(env);
SHORTCIRCUIT:
...
```

Operatori relazionali

Gli unici operatori relazionali presenti nel bytecode sono `bleq $r1 $r2 LABEL`, che fa saltare l'esecuzione a `LABEL` se e solo se `$r1 <= $r2`, e `beq $r1 $r2 LABEL`, che fa saltare l'esecuzione a `LABEL` se e solo se `$r1 == $r2`. La traduzione dell'espressione `e1 == e2` è la seguente:

```
...
e1.generateWTMCode(env);
push $a0
e2.generateWTMCode(env);
top $t1
pop
beq $t1 $a0 TRUE
li $a0 0    //caso affermativo
b END
TRUE:
li $a0 1    //caso negativo
END:
...
```

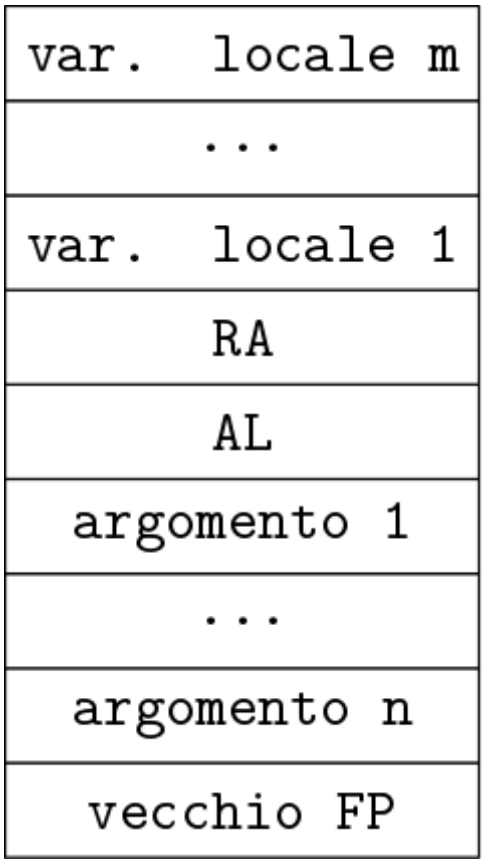
I restanti operatori (`!=, >=, < e >`) sono stati ricondotti ciascuno ad uno di queste due primitive, attraverso le seguenti equivalenze:

$$
\begin{aligned}
x \neq y &\iff \neg(x = y) \\
x \geq y &\iff y \leq x \\
x < y &\iff \neg(y \leq x) \\
x > y &\iff \neg(x \leq y)
\end{aligned}
$$

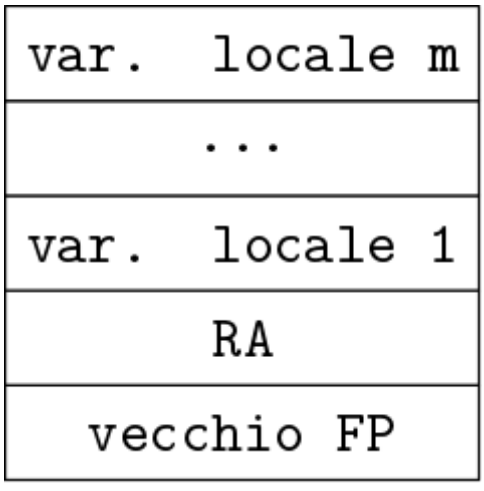
La negazione è ottenibile facilmente scambiando il contenuto dei rami affermativo e negativo.

Activation Record

Il layout degli activation record assunto durante la generazione del codice intermedio è simile a quello visto a lezione, con la differenza che, essendo presenti in questo linguaggio **variabili locali**, queste vengono allocate nella parte dinamica dello stack, quella solitamente riservata ai risultati intermedi delle espressioni. L'activation record di una **funzione** con `n` parametri e `m` variabili locali ha pertanto la seguente forma:



Si noti come le variabili locali vengono allocate (necessariamente) dal basso verso l'alto. Ne consegue che gli argomenti hanno offset crescenti rispetto a `AL` (il primo con offset `+1`), mentre le variabili hanno offset decrescenti (la prima con offset `-2`). L'environment tiene traccia di quali variabili sono parametri e quali sono locali nel campo `parameter` di ciascun oggetto `CompilationSTEntry` salvato nella symbol table durante la fase di compilazione. L'activation record dei blocchi è invece più semplice, grazie all'assenza di argomenti e al fatto che `AL` e vecchio `FP` coincidono. Un **blocco** con `m` variabili locali ha il seguente activation record:



NOTA: nonostante `RA` sia inutile al funzionamento dell'activation record del blocco, esso viene comunque messo sullo stack per preservare l'offset iniziale delle variabili locali.

La chiamata di funzione e il corpo di una funzione danno origine a codice del tutto analogo a quello visto a lezione, con l'unica eccezione che durante la chiusura di una funzione (o di un blocco) vengono rimosse dallo stack anche le variabili locali (con `m` pop, dove `m` è fornito dall'environmente, che conta le variabili locali dichiarate in ciascuno scope).

Passaggio per variabile

Il passaggio per variabile (o per riferimento) è realizzato in modo intuitivo salvando sullo stack, durante la chiamata di funzione, non tanto il valore, quanto **l'indirizzo assoluto** della prima allocazione della variabile passata. L'environment tiene traccia di quali variabili sono riferimenti e quali no nel campo `reference` di ciascun oggetto `CompilationSTEntry`.

L'accesso in lettura e in scrittura a variabili passate per riferimento prevede un ulteriore stadio di **dereferenziazione** rispetto al regolare accesso a variabili statiche locali o globali (si veda il metodo `generateWTMCode(Environment env)` delle classi `VariableValue` e `Assignment`).

NOTA: l'indirizzo di una variabile viene calcolato solo la prima volta che essa è passata per riferimento. Se il tale riferimento è passato a sua volta per riferimento ad un'altra funzione, ci è sufficiente copiare il relativo l'indirizzo nell'activation record della seconda funzione (si veda il metodo `generateWTMCode(Environment env)` della classe `FunctionCall`).

Delete

L'eliminazione di identificatori di variabili e funzioni è stata trattata come un **costrutto puramente logico** per il riutilizzo di identificatori, e non come una primitiva di allocazione di memoria. Pertanto, l'eliminazione è del tutto trasparente nella fase di compilazione, che tratta le ridichiarazioni come se fossero normalissime dichiarazioni di nuove variabili e funzioni.

Nota: se il programma è consistente secondo l'analisi comportamentale, questo approccio funziona (i.e. si accede sempre alle variabili/funzioni corrette durante l'esecuzione del codice).

Interpretazione

Rappresentazione interna del codice

La prima fase dell'interpretazione consiste nell'**analisi** del codice intermedio nella sua forma testuale e nella conseguente **generazione di una rappresentazione interna** del dodice adatta ad essere passata alla World Turtle Machine.

L'analisi lessicale è portata a termine dal lexer (classe `WTMLexer` generata da ANTLR4) e consiste semplicemente nel conteggio degli errori lessicali (i.e. token imprevisti) presenti nel programma.

NOTA: in un linguaggio dalla grammatica semplice quanto quella del bytecode l'analisi lessicale è sufficiente a rilevare ogni possibile erorre sintattico.

In seguito il parser (classe `WTMParser` generata da ANTLR4) genera l'albero di sintassi concreta. Una **visita di tale albero** tramite il processing visitor (classe `WTMProcessingVisitor`) permette di tradurre le singole istruzioni nella loro rappresentazione interna(vedi prossimo paragrafo). Al termine della visita, le **etichette** vengono rimosse dal codice e ogni loro riferimento viene sostituito dall'indirizzo assoluto della loro prima occorrenza (così come visto a lezione. Si veda il metodo `visitAssembly(AssemblyContext ctx)`) e un'istruzione di `HALT` viene aggiunta in fondo al codice.

La rappresentazione interna di un'istruzione della World Turtle Machine non è altro che un'oggetto `WTMInstruction` contenente **quattro campi interi**: `instr`, `arg0`, `arg1` e `arg2` . Il campo `instr` contiene sempre un identificatore numerico della funzione da eseguire (gli identificatori sono quelli generati automaticamente da ANTLR4 in `WTMLExer`), mentre i restanti campi contengono o interi, o identificatori numerici di registri (definiti nella classe `WTM`), a seconda di quali siano gli argomenti previsti dall'istruzione specificata da `instr`. Ad esempio, l'istruzione `addi $a0 $a0 1` dà origine alla rappresentazione interna `[6, 0, 0, 1]`, dove `6` identifica l'istruzione `addi` e `0` identifica il registro `$a0`.

La World Turtle Machine (WTM)

> *Il pesce falco non avrebbe potuto tenerla sopra le sue ali per molto tempo, così chiese agli altri animali e agli uccelli un aiuto: avrebbero dovuto creare un terreno solido su cui la donna potesse stare. Uno svasso si calò fino al fondo del mare e riportò del fango nel suo becco. Una tartaruga spalmò il fango sul suo guscio, e si rituffò ancora molte volte. Anche le anatre portarono i loro becchi pieni del fondo dell'oceano e lo spalmavano sul dorso della tartaruga. Il castoro aiutò a costruire il terreno, rendendo la copertura più grande. Gli uccelli e gli animali costruirono i continenti fino a formare la terra, mentre la donna era al sicuro seduta sulla schiena della tartaruga. Ma, secondo tale tradizione, la tartaruga continua a tenere la terra sulle sue spalle.*
>
> *– Mito della creazione irochese.*

La seconda (e ultima) fase dell'interpretazione consiste nell'esecuzione del codice intermedio sulla World Turtle Machine, una macchina virtuale che interpreta il bytecode precedentemente introdotto. Tale macchina è dotata di **sei registri**: `$a0`, `$t1`, `$fp`, `$al`, `$sp` ed `$ra`, ovvero gli stessi registri visti a lezione. La macchina è inoltre dotata di una **memoria** di lettura e scrittura (`memory`) della dimensione di 10.000 parole (ciascuna di 4 byte), contenente dati e manipolata interamente come uno **stack**, e di una seconda **memoria di sola lettura** (`code`), della dimensione anch'essa di 10.000 parole (o 2.500 istruzioni), contenente il codice da eseguire.

La WMT viene inizializzata col la rappresentazione intermedia generata al passo precedente in `code`, con `$sp` che punta all'ultima locazione di memoria ed `$fp` che punta alla penultima. Il campo intero `ip` tiene conto della prossima istruzione da eseguire, ed è inizializzato a 0. Ad ogni passo di esecuzione, ul'istruzione alla `ip`-

esima posizione di `code` viene analizzata ed eseguita. L'esecuzione termina qualora il codice si esaurisca, o si incontri un'istruzione di `HALT` (Stampa `"Halted."`), o la memoria a disposizione si esaurisca (ovvero quando `SP <= 0`, stampa `"Error: out of memory."`).