RELAZIONE FINALE COMPILATORI ED INTERPRETI

Alessandro Serra mat.891061
Andrea Longo mat.902517
LM Informatica per il management
A.A 2018-2019

Indice

1 Introduzione

Il progetto realizzato rappresenta lo sviluppo di un'estensione di un nuovo linguaggio denominato *SimpleStaticAnalysis*. Il linguaggio è stato sviluppato attraverso lo svolgimento di 4 consegne svolte durante il corso.

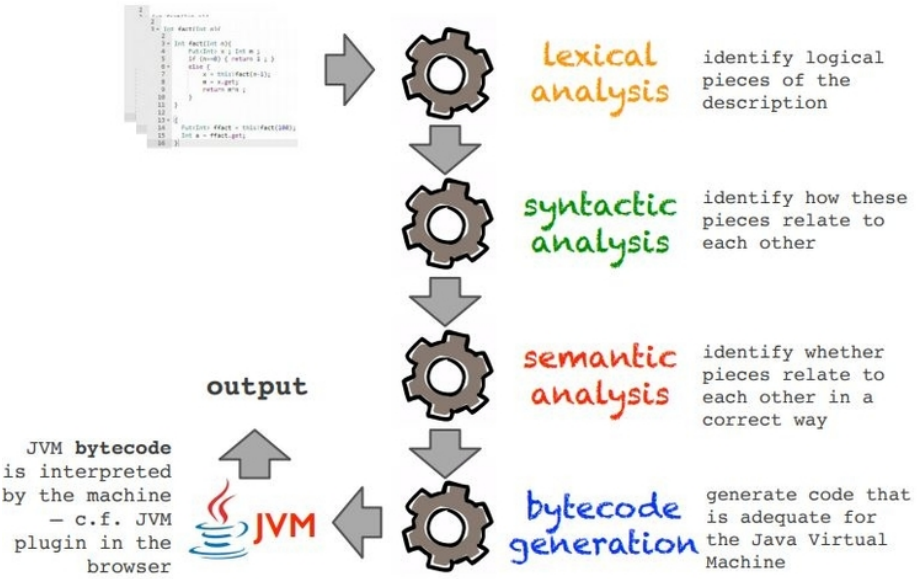

Figura 1: Fasi dello sviluppo di un compilatore.

Tali consegne rappresentano le fasi principali di analisi e compilazione di un nuovo linguaggio, come rappresentato in Figura 1, e sono:

1. **Analisi lessicale:** in questa fase vengono esaminati tutti i caratteri inseriti nel programma di input verificando, per ciascun token, se questo è ammesso dalla nostra grammatica oppure se si tratta di un token illegale. Nel caso di un token illegale, viene lanciato un errore ed il processo termina immediatamente la sua esecuzione.

2. **Analisi sintattica:** in questa fase si verifica che la costruzione dell'abstract syntax tree sia avvenuta con successo, ovvero che non ci siano delle malformazioni dovute alla mancanza di caratteri laddove la grammatica se li aspetta. Anche in questo caso, ogni errore è segnalato da un apposito messaggio.

3. **Analisi semantica:** in questa fase sono esplicitati i vari errori che si possono riscontrare durante la costruzione della symbol table oppure durante il type checking delle varie operazioni.

4. **Generazione del Bytecode:** nell'ultima fase viene creato il codice assembly che è successivamente utilizzato dal metodo *cpu* della classe *ExecuteVM* che esegue le istruzioni generate.

Lo "strumento" di parsing utilizzato è *ANTLR*, un generatore di parser che fa uso del sistema di parsing LL(*).

2 Analisi lessicale

L'obiettivo di questa fase è la rilevazione di eventuali token illegali presenti nel programma di input. Infatti, l'analisi lessicale è il processo che prende in ingresso una sequenza di caratteri e produce in uscita una sequenza di token.

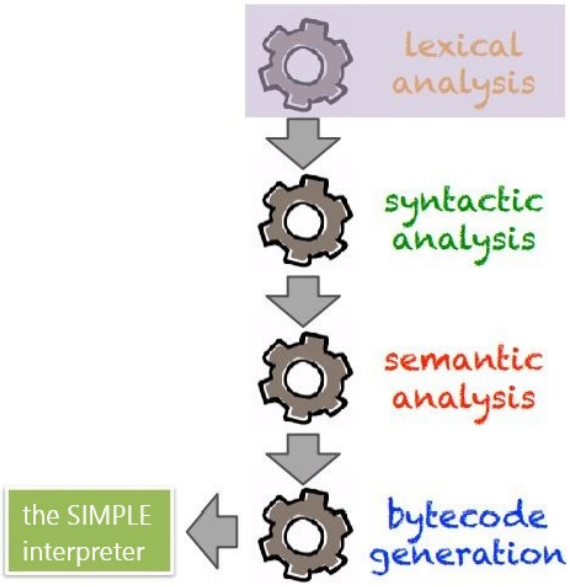

Figura 2: Analisi lessicale nel processo di sviluppo.

In questa fase, quindi, sono analizzati tutti i caratteri presenti nel programma di input che, una volta suddivisi in token, sono analizzati e riconosciuti dal lexer. Infine, i token riconosciuti, sono inviati al parser. Nel caso siano presenti token illegali, viene lanciata un'eccezione facendo terminare il processo e salvando eventuali errori lessicali sul file *lexsyn.err*. Il riconoscimento dei token avviene in relazione alla grammatica descritta nel file *Simple.g4*, a sua volta basata sul linguaggio delle *espressioni regolari*.

2.1 Lexer: regole di partenza

Di seguito sono riportate le regole lessicali presenti nella grammatica di base.

- Gestione identificatori:
  fragment CHAR : 'a'..'z' |'A'..'Z' ;
  ID : CHAR (CHAR | DIGIT)* ;
- Gestione numeri:
  fragment DIGIT : '0'..'9';
  INTEGER : DIGIT+;

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

Possiamo notare come, in presenza di *fragment* e *skip*, non viene generato alcun nodo nell'albero in quanto:

- Nel primo caso, il termine riconosciuto è utilizzato come parte di una regola più grande.
- Nel secondo caso, tutti i termini riconosciuti vengono skippati.

2.2 Lexer: regole aggiunte

Di seguito è riportata la regola lessicale aggiunta alla grammatica di base:

- Operatori relazionali:
  ROP : '==' | '>' | '<' | '<=' | '>=' | '!=';
- Operatore di gestione degli errori:
  ERR : . -> channel(HIDDEN);

3 Analisi sintattica

Attraverso l'analisi sintattica, è possibile assegnare al testo in input una struttura che prende in considerazione le singole unità del discorso e le relazioni tra loro.

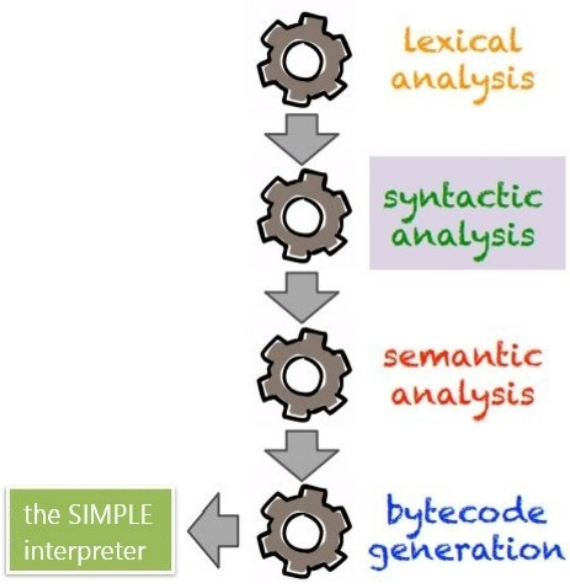

Figura 3: Analisi sintattica nel processo di sviluppo.

Le singole unità del discorso sono tutti i token riconosciuti dal lexer, in base ai quali il parser costruisce l'albero su cui viene effettuata l'analisi sintattica. L'albero AST essendo costruito in relazione alle regole definite dalla grammatica, permette subito di individuare eventuali token mancanti ma previsti dalla stessa. Quando viene riscontrata una malformazione dell'AST viene lanciata un'eccezione e l'esecuzione del processo viene interrotta, salvando gli errori sul file *lexsyn.err*. L'albero generato, invece, è salvato sul file *tree.ast*. Di seguito viene fornito un breve esempio che rappresenta la costruzione dell'*abstract syntax tree*.

```
{
    //Test chiamata di funzioni annidate
    f(int a){
        print a-1000;
    }
    g(int b){
        print b;
        f(b);
    }
    int c=1;
    g(c);
    delete c;
}
```

Figura 4: Codice di test.

```
InitBlock
  Block
    Fun:f
      Par: a
      Mode: val
      Type: IntType
      Entry:    STentry: nestlev 2
    STentry: deleted FALSE
    STentry: typeIntType
    STentry: offset 1
      Block
        Print
          Subt
            Id:a at nestlev 3
              STentry: nestlev 2
              STentry: deleted FALSE
              STentry: typeIntType
              STentry: offset 1
            Int:1000
    Fun:g
      Par: b
      Mode: val
      Type: IntType
      Entry:    STentry: nestlev 2
    STentry: deleted FALSE
    STentry: typeIntType
    STentry: offset 1
      Block
        Print
          Id:b at nestlev 3
            STentry: nestlev 2
            STentry: deleted FALSE
            STentry: typeIntType
            STentry: offset 1
        Call:f at nestlev 3
          STentry: nestlev 1
          STentry: deleted FALSE
          STentry: typeArrowType
IntType
          STentry: offset -1
          Id:b at nestlev 3
            STentry: nestlev 2
            STentry: deleted FALSE
            STentry: typeIntType
            STentry: offset 1
    Var:c
      IntType
      Int:1
    Call:g at nestlev 1
      STentry: nestlev 1
      STentry: deleted FALSE
      STentry: typeArrowType
IntType
      STentry: offset -2
      Id:c at nestlev 1
        STentry: nestlev 1
        STentry: deleted TRUE
        STentry: typeIntType
        STentry: offset -3
    Deletion: c
```

Figura 5: *AST* ottenuto.

4 Analisi semantica

L'analisi semantica è il processo che consente di stabilire il significato delle parole contenute in una frase, all'interno di una discussione orale oppure di un testo scritto.

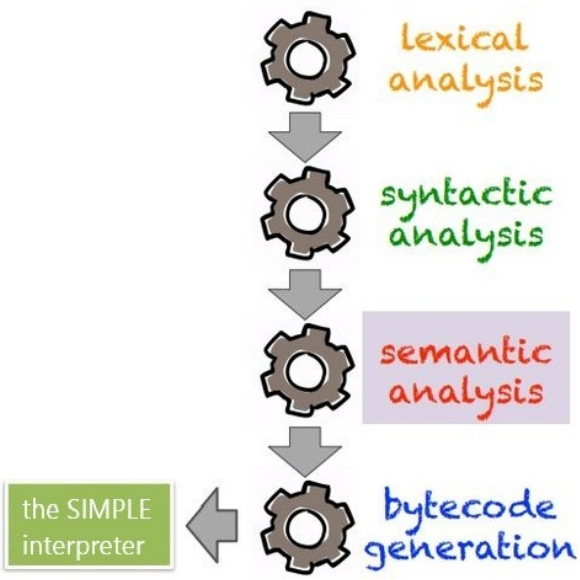

Figura 6: Analisi semantica nel processo di sviluppo.

In questa fase si cerca di individuare tutti gli errori che non sono stati trovati dal lexer e dal parser.
Di seguito sono riportati alcuni errori semantici tipici:

- Variabili/funzioni non dichiarate: una variabile/funzione non può essere utilizzata se prima non è stata dichiarata correttamente.
- Dichiarazioni multiple: una variabile dev'essere dichiarata nello stesso scope al massimo una volta.
- Parametri attuali non conformi ai parametri formali (inclusi quelli passati per var): i metodi dovrebbero essere chiamati con il giusto numero di parametri e la giusta tipologia.
- Type mismatch: il tipo della parte sinistra di una qualsiasi operazione deve corrispondere al tipo della parte destra.

Un analizzatore semantico agisce attraversando l'AST costruito e si compone di due fasi:

1. Scope checking: processa le dichiarazioni e le istruzioni aggiungendo nuove voci alla *symbol table* e generando nuovi *ID Node* legati ai simboli corrispondenti nella *symbol table*.
2. Type checking: utilizza le informazioni della *symbol table* per determinare il tipo di ogni espressione e trovare errori di tipo.

Tutti gli errori semantici sono stampati a video.

4.1 Scope checking

Attraverso lo scope è possibile recuperare l'insieme di celle del programma dove sono memorizzati gli elementi in dichiarazione. Ciò viene fatto attraverso la *symbol table*.

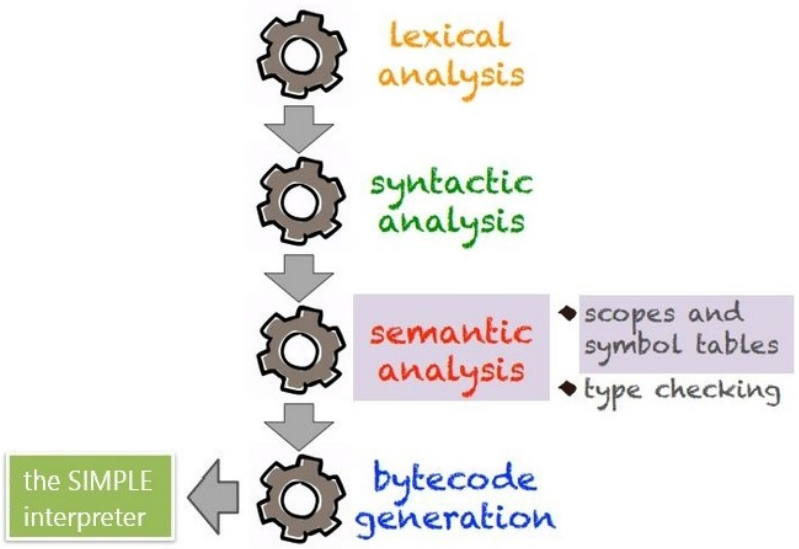

Figura 7: Scope checking nel processo di sviluppo.

Nel nostro caso, la *symbol table* è costruita secondo il metodo delle liste di hashtable. Per ogni scope del programma, che avrà un suo *nesting level*, sono disponibili le seguenti operazioni:

- Ad ogni ingresso in uno scope: incrementa il *nesting level* e aggiunge una nuova hashtable.

- Per processare una dichiarazione: guarda nella prima tabella se è presente l'identificatore. Se lo è allora si restituisce l'errore *"dichiarazione multipla"*, altrimenti si aggiunge l'identificatore alla lista.

- Per processare un utilizzo: cerca nelle liste la dichiarazione dell'identificatore. Se non lo si trova, allora si restituisce l'errore *"variabile non dichiarata"*.
- Ad ogni uscita: rimuove la prima tabella dall'elenco e diminuisce il numero del *nesting level*.

4.2 Type checking

Il type checking è l'operazione che consiste nel controllare che i valori assegnati ad una variabile siano di un tipo di dato ammissibile per il tipo della variabile e che le operazioni siano usate con i dati corretti. Nel type checking tutto ruota attorno al concetto di *vtable*, *ftable* e *environment* e, all'interno del codice, tutto viene controllato attraverso il metodo *typeCheck*.

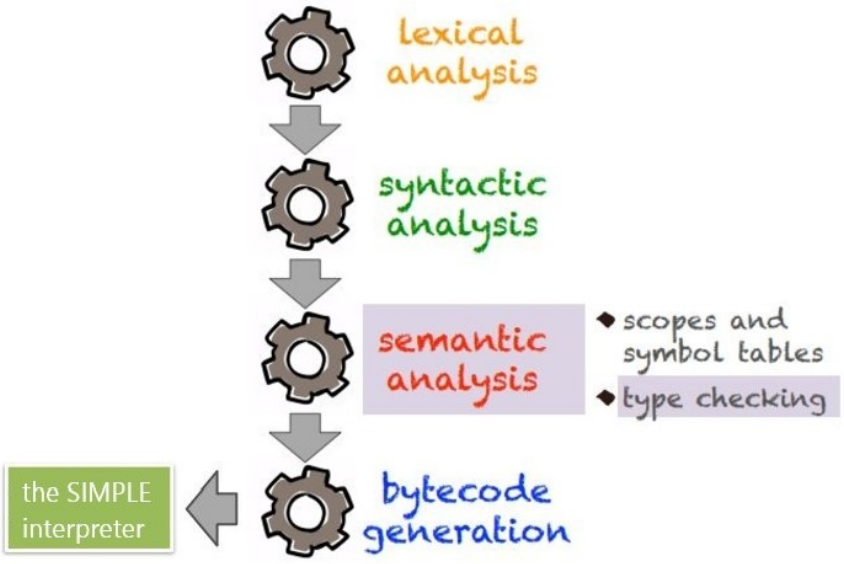

Figura 8: Type checking nel processo di sviluppo.

5 Generazione del Bytecode

L'ultima fase prevede la generazione di codice oggetto. L'obiettivo è generare le istruzioni assembly che permetteranno di memorizzare ogni variabile, metodo e funzione all'interno dell'indirizzo di memoria corretto.

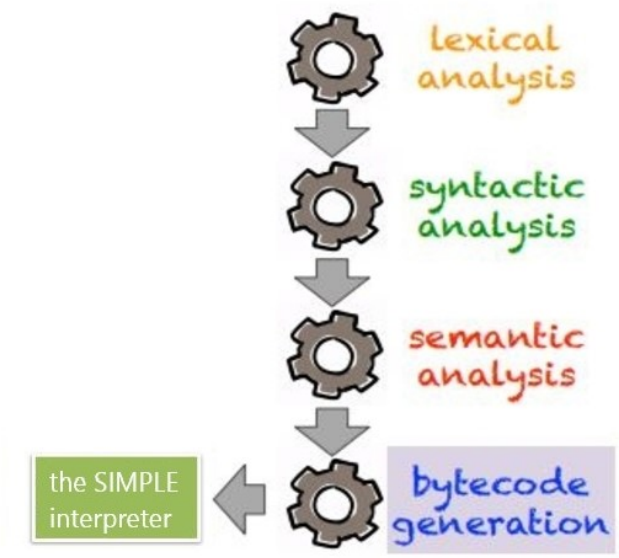

Figura 9: Generazione di codice nel processo di sviluppo.

Questa fase si compone di due sottofasi:

- Generazione del codice oggetto
- Esecuzione della Virtual Machine

5.1 Generazione e visita del codice oggetto

Per la generazione del codice sono state utilizzate le regole viste e discusse a lezione (del compilatore *FOOL*) con l'aggiunta di altre regole caratteristiche. Nello specifico, vengono descritte solo le regole assembly aggiunte:

- **BRANCHNOTEQ:** 'bneq' - va alla label se next != top.
- **BRANCHGT:** 'bgt' - va alla label se next > top.
- **BRANCHGREATEREQ:** 'bgeq' - va alla label se next >= top.
- **BRANCHLT:** 'blt' - va alla label se next < top.

Una volta generate, le regole vengono utilizzate in base alla tipologia di nodo in esame e, le regole attivate, vengono inserite nel file *prova.simple.asm*. Successivamente tale file viene letto e, per ogni istruzione presente al suo interno, viene riempito un array (denominato *code* e successivamente usato dalla VM) dove l'indice rappresenta l'i-esima istruzione visitata, ed il contenuto è il codice identificativo dell'istruzione nel parser.

5.2 Esecuzione della Virtual Machine

Per l'esecuzione del codice vengono utilizzati due elementi fondamentali:

- Stack machine - SVM (Simple Virtual Machine): semplice modello di valutazione che si basa su uno stack di valori utile a memorizzare i risultati intermedi. Tale modello può implementare anche i registri per rendere l'esecuzione più rapida attraverso un minor numero di accessi alla memoria stessa.
- Interprete: traduce il bytecode precedentemente generato eseguendo il codice ottenuto attraverso l'utilizzo di registri specifici (descritti nel capitolo *6:Interprete*).

6 Interprete

Come detto, l'interprete è il focus del nostro progetto.

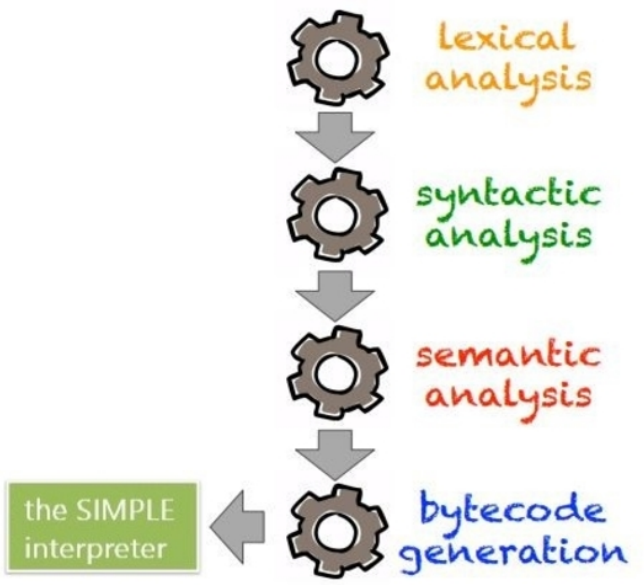

Figura 10: Interprete nel processo di sviluppo.

Il passo finale è quello di eseguire il codice utilizzando specifici registri e ciò avviene attraverso il metodo *cpu* della classe *ExecuteVM*.
I registri sono:

- **IP:** instruction pointer - punta all'istruzione successiva da eseguire.
- **SP:** stack pointer - punta in cima allo stack.
- **FP:** frame pointer - punta alla prima istruzione presente all'interno di un blocco.
- **HP:** heap pointer - punta alla coda dello heap.
- **RA:** return address - rappresenta l'indirizzo dell'istruzione successiva da eseguire quando termina l'esecuzione della funzione chiamata.
- **RV:** return value - memorizza il valore di ritorno di una funzione.

7 Struttura del progetto

La consegna si articola all'interno del progetto *Java* denominato *consegnaFinale_Longo_Serra*. Tale progetto è costituto da vari *package*, ovvero:

- **default:** contiene la classe per testare il compilatore e l'interprete.
- **ast:** contiene le classi che definiscono i comportamenti sui vari tipi di nodi.
- **lib:** contiente la classe dove sono presenti i metodi di supporto alle classi contenute nel package *ast*.
- **parser:** è il cuore pulsante del progetto, dove si trovano tutte le classi auto-generate da *ANTLR*.
- **util:** contiene le classi che definiscono gli strumenti di supporto all'analisi semantica.

In aggiunta a ciò, sono presenti anche:

- le librerie utilizzate: *Java* e *ANTLR*(versione 4.6).
- esempi: cartella contenente degli esempi di codice (molti dei quali visti a lezione).
- 3 file: due di questi (*lexsyn* e *ast*) sono stati descritti durante le fasi di analisi lessicale, sintattica e semantica mentre, il file *prova* dev'essere utilizzato per testare il codice (come descritto nel capitolo 8).

8 Testing

Si è scelto di utilizzare due tipologie di rilascio (e quindi successivo testing). È possibile avviare il progetto attraverso:

- **Eseguibile:** dopo aver modificato il file *prova.simple* inserendo il codice da testare, è possibile lanciare il file *test_project_Longo_Serra.bat* visualizzando le informazioni direttamente su command line.
- **Import su Eclipse:** dopo aver importato il progetto su Ecplise e aver modificato il file *prova.simple* inserendo il codice da testare, invece, è possibile lanciare il test attraverso la classe *Test.java* presente all'interno del package di default.