Compiler's notes

Giulia Brugnatti

May 2021

1 Compiler

Un compilatore è un programma che preso in input tutto il testo di un programma scritto in un determinato linguaggio è in grado di tradurlo in un altro[1], controllando che questi rispetti determinate regole.

1.1 Fasi

Per poter tradurre il programma nel codice sorgente in quello del linguaggio target, il *compiler* deve attraversare diverse fasi che possono essere riassunte in due macro categorie: quelle di Front-end e quelle di *back-end*.

Della fase di Front-end fanno parte le fasi di analisi che mirano a rilevare i possibili errori del programma. In particolare queste consistono in:

- **analisi lessicale**: processo operato dal *Lexer* in cui il programma preso in input come stream di caratteri viene raggruppato in token;
- **analisi sintattica**: insieme di operazioni effettuate dal *Parser*[2]; quest'ultimo prende la sequenza di *tokens* generata dal *Lexer* e verifica che una certa sequenza possa essere generata come una derivazione della *CFG* del programma sorgente[3]. Alla fine di questa operazione il *Parser* produce il *Parse Tree*
- **analisi semantica**: task che si assicura che dichiarazione di un programma sia semanticamente corretta. Questa fase include il *Type-Checking*.

Della fase di back-end fanno parte le ottimizzazioni e la generazione del codice.

[1] Il compilatore in particolare traduce in codice oggetto.
[2] il *Parser* sta a una *CFG* come un *DFA* sta a una *Regular expression*
[3] Il nostro *Parser* sia *top-down*

2 Stack Machine

Il compilatore sviluppato, è un compilatore per *Stack Machine*, ossia esso utilizza lo *stack* per il calcolo delle operazioni, consentendo così la generazione del codice per una qualsiasi espressione.

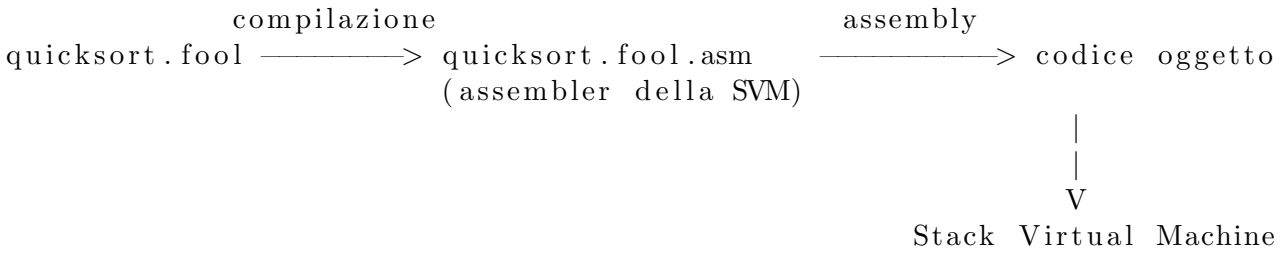
Per esempio; per calcolare `exp1 op exp2` si fa quanto segue:

- calcolo `exp1` (fa calcoli usando lo *stack* e produce un risultato come unico valore aggiunto in cima allo stack);
- calcolo `exp2` (fa calcoli usando lo *stack* e produce un risultato come unico valore aggiunto in cima allo stack);
- `op` (fa `pop` di operandi dallo stack e `push` del risultato sullo *stack*);

In questo contesto; ogni sotto espressione è un'invariante che usa lo stack per fare i calcoli al termine dei quali lo Stack viene lasciato invariato se non aggiungendogli il valore che ha calcolato.

REALIZZAZIONE DELLA STACK VIRTUAL MACHINE (SVM) E DEL RELATIVO ASSEMBLATORE

Esempio: compilazione ed esecuzione file sorgente quicksort.fool



---

3 ANTLR

ANother Tool for Language Recognition; è uno strumento scritto in *Java*, che permette di tradurre una grammatica in un parser/lexer[4] per un linguaggio target (*Java, C#, Python* o *JavaScript*)[5].

Nello specifico è un generatore di parser in grado di: costruire e navigare parse trees, leggere, processare, eseguire e tradurre testi strutturati e *files* binari.

Il *Parser* di *ANTLR* è di tipo `LL(*)`; ossia è un *parser* predittivo[6] adattivo utilizza un numero arbitrario di *lookhaed*[7], ma è *bounded*[8].

Accetta grammatiche ricorsive a sinistra e le gestisce in modo trasparente: se c'è un ambiguità per una frase e *ANTLR* ha più scelte, sceglie la prima che può essere applicata; ossia il primo parse tree che può generare (in caso di fallimento fa *backtrack*). L'ordine con cui le regole sono scritte nel file .g4 conta; di default è associativo a sinistra (derivazione canonica sinistra[9]) Usa una grammatica *EBNF*;

I file contenenti la grammatica sono specificati nel formato ".g4", sono composti:

- dal nome della grammatica,
- dalle sezioni dedicate:
  - opzioni,
  - import,
  - *token*,
  - canali,
  - @azioni
- da un insieme di regole.

Per distinguere tra: grammatiche per il *Parser* e grammatiche per il *Lexer* si possono usare due notazioni differenti nella specifica del nome della grammatica: `parser grammar Name` e `lexer grammar name`; in assenza di queste la grammatica definita è una grammatica combinata che può contenere regole per entrambi.

ANTLR permette di specificare il linguaggio tramite una grammatica *context-free* e usando notazioni *EBNF*.

EBNF : estensione delle *CFG* che permette di utilizzare più operatore in particolare permette di descrivere i token per mezzo di *espressioni regolari*.

[4] Genera di default codice *Java*; alcune delle sue classi sono: *Charstream, Lexer,Token, Parser, ParseTree*.

[5] Il target è specificabile tramite l'opzione `-Dlanguage =` da linea di comando, o installando *ANTLR* direttamente nell'ambiente di sviluppo (*Eclipse, IntelliJ, Visual Studio, Maven*) tramite un *plugin*.

[6] Lavora top-down.

[7] Usa quanti simboli di *lookhead* gli servono a seconda dei casi analizzando cioè la grammatica a runtime, cercando di fare ottimizzazioni

[8] Bounded: può variare ma non può essere infinito.

[9] Derivazione canonica sinistra: viene sostituita sempre la variabile più a sinistra con il corpo di una delle sue regole.

4 Progetto

Lo scopo del programma del progetto è quello di riuscire a tradurre un programma scritto in un linguaggio ad alto livello, *FOOL*, nel corrispondente linguaggio macchina.

Tale risultato tramite una serie di passi operati da due entità principali: il compilatore e la SVM.

In generale, tale processo può essere riassunto come segue. Il compilatore opera in due fasi: la prima, quella di front-end e la seconda di *back-end.*

Nella prima, il programma scritto in linguaggio *FOOL* viene convertito in uno *stream* di caratteri (la classe *CharSream* automaticamente generata da *ANTLR* si occupa di tale passaggio), dopodiché lo stream viene preso in carico dal *Lexer* (anch'esso automaticamente generato dal tool) e trasformato in uno *stream* di *token* che viene infine passato al *Parser* (ultimo elemento creato dal generatore) che produce il *Syntax Tree.*

A questo punto, grazie ad una visita a tale struttura dal *ASTGenerationVisitor* viene prodotta una rappresentazione astratta dell'albero l'*AST.*

Quest'ultimo elemento viene poi dato in input a successivi visitor per controllare la correttezza del programma. Nello specifico prima viene passato al *SymbolTableVisitor* perchè *matchi* le dichiarazioni con i relativi usi e di seguito al *TypeCheckVisitor* perché controlli che le regole del linguaggio siano rispettate.

Se si è riusciti a *matchare* tutte le dichiarazioni coi relativi usi e non si sono ottenuti errori durante la fase di *type-checking* allora l'albero è completo e quindi la fase di front-end termina e si passa a quella di backend.

Durante quest'ultima avviene la vera e propria produzione di codice ad opera del *CodeGenerationVisitor.* Quest'ulitmo produce una rappresentazione intermedia del codice (`fool.asm`) che verrà poi utilizzata dalla *SVM.* Nello specifico, la *SVM* opera lo stesso processo effettuato dal compilatore, ma invece che utilizzare come sorgente il file scritto in linguaggio *FOOL* usa il `fool.asm`. Tale entità infatti a partire da tale file produce un nuovo *stream* di caratteri che viene passato al suo *Lexer*, il quale produce lo *stream* di *token* che viene utilizzato dal *Parser* per produrre il codice *assembly* che verrà eseguito da *ExecuteVM*;

5 Key Words

Fasi

- front-end:
  - analisi lessicale;
  - parsing;
  - analisi semantica;
- back-end:
  - ottimizzazione;
  - code generation;

Lexer : presi in input una sequenza di caratteri, questi li partiziona in un insieme di lessemi che vengono quindi mappati a una sequenza di token.

Possono essere imperativi o, dichiarativi, in *ANTLR* si implementa un parser dichiarativo dove ogni *token* viene descritto come un'espressione regolare e il simbolo di *lookhead* utilizzato per *matchare* le produzioni è *bounded* ma variabile a seconda dei casi.

In generale un *Lexer*: traduce le *regex* in un *FA*, unisce tutti gli *FA* in un unico automa, tradotto poi in un *DFA*, produce codice che implementi una speciale simulazione del *DFA* in modo da implementare la regola del *maximal match*.

Regular Expression : modo compatto per descrivere un linguaggio accettato da un automa (esiste una traduzione 1 a 1 tra un *DFA* e una *regex*).

Possono essere utilizzate come input per un generatore di *Lexer*; essi descrivono i lessemi che devono essere mappati in *token*, così come altri caratteri come *white-space* e commenti [10]

Parser : responsabile di due compiti: l'analisi sintattica e la realizzazione di *ST* e *AST*. L'analisi sintattica; cerca di rilevare se il programma è affetto da errori sintattici creando un *ST*, quest'ultimo viene poi sfruttato per la realizzazione dell'*AST* che in *ANTLR* viene realizzato top-down).

In particolare, *ANTLR* utilizza un parser top-down predittivo che utilizza un nuomero arbitrario di *lookahead* per predirre quale produzione utilizzare: è `LL(*)`: dove la prima L sta per *left-to right*, la seconda per "*leftmost derivation*"(*ANTLR* è di default associativo a sinistra) e l' "*" sta appunto al numero arbitrario di *lookahead*.

[10] Caratteri bianchi e commenti non devono essere mappati a *token* ma devono poter essere riconosciuti e ignorati.

ST : rappresentazione dei *token* del programma che contiene tutti i suoi elementi, anche quelli non utili ai fini delle analisi lessicali ecc., ma non sono altro che zucchero sintattico utilizzato dai programmatori per dire al *Parser* in che ordine deve essere costruito l'albero.

In questo caso i simboli terminali sono nelle foglie.

NB Spesso non viene realizzato fisicamente ma gestito internamente dal programma.

AST : rappresentazione dei *token* utili del programma, viene realizzato appiattendo l'ST e liberandosi di tutti i simboli inutili: parentesi di vario genere, punti e virgola ecc. In questo caso i simboli terminali sono i nomi dei padri.

Analisi sintattica : prende in input un *ST* e restituisce in output un *AST*. Nella costruzione del *ST* viene controllata la sintassi del programma, ossia se il numero di parentesi è bilanciato, se ci sono i punti e virgola alla fine delle istruzioni ecc.

La struttura sintattica di un programma è descritta attraverso una grammatica; nel caso specifico di ANTLR la grammatica in questione è una *EBNF*.

Grammatica : la sintassi di un linguaggio per essere espressa bisogna farne una descrizione; i *Parser parsano* la descrizione formale del linguaggio (grammatica[11]).

NB I costrutti di un linguaggio di programmazione hanno struttura ricorsiva.

CFG :Per descrivere il linguaggio del compilatore, o per meglio dire per esprimere le regole di sintassi, si necessitano i *linguaggi liberi*, ossia un'estensione dei linguaggi regolari che utilizza anche lo *stack* [12], in particolare viene utilizzata un'estensione delle *CFG* denominata *EBNF*.

Espressioni : numeri variabili, risultati di valutazioni di altre espressioni (`exp - exp` è sua volta una `exp`), in generale calcolano un valore che viene restituito.

Statement : pezzo di codice che non restituisce un valore, ma fa qualcosa.

Attivazione : invocazione di una procedura, include tutti gli *step* in cui questa è eseguita e chiamata.

Lifetimes of Variable : porzione di esecuzione in cui questa è definita, è un concetto dinamico (run-time), mentre lo scope è un concetto statico.

[11] Le grammatiche hanno un punto di vista generativo (generano stringhe), mentre gli automi hanno un punto di vista riconoscitivo

[12] Regex: linguaggi regolari= CFG: linguaggi liberi

Access Link : settato al frame che racchiude sintatticamente il blocco, per una funzione tale blocco è quello che contiene la dichiarazione.

Come settare l'AR?

In generale risale la catena degli AL per un numero di passi pari alla differenza tra il *nesting level* corrente e quello dove la funzione è stata dichiarata.

Activation Record : L'*activation record* (*frame*) contiene le informazioni che sono necessarie per gestire la procedura di attivazione (comprese quelle per ripristinare l'esecuzione di una procedura se durante la sua esecuzione viene sospesa.)

Generalmente l'AR di una funzione contiene: valore di ritorno, variabili locali, parametri (allocati dal chiamante), puntatore al precedente AR (CL).

Il *layout* degli AR va deciso a priori (*compile-time*), per poter generare codice che accede alle corrette locazioni dell'AR.

NB Il *Main* a livello logico non ha un *return address*, quindi potrebbe avere un *layout* speciale, ma nel caso di questo progetto non è così perché si è scelto di assegnargli. Perché il *Main* non ha un *return address*? Perché esso non rappresenta altro che il punto in cui l'esecuzione riprende una volta che è finita la chiamata alla procedura, quando il *Main* termina non c'è più nulla da fare generalmente.

NB2 Le variabili globali non sono contenute negli AR tutte i riferimenti ad esse puntano allo stesso elemento (sono allocate a indirizzi fissi).

NB3 Riferimenti a variabili non dichiarate all'AR in cui ci si trova vanno cercate negli AR precedenti secondo la regola del ***most closely nested***, vanno cercati via via nell'AR dello scope che racchiude immediatamente quello valutato, risalendo se necessario fino all'ambiente globale.

Control Link : riferimento al AR del chiamante di una funzione.

Frame Pointer : punta all'AR del corpo della funzione in esecuzione in questo momento e più in particolare punta alla sua posizione di riferimento.

Heap :contiene i valori che devono sopravvivere alla procedura che li crea;

Analisi lessicale : controlla se ci sono *token* "illegali", *token* che non fanno parte dei lessemi della grammatica;

Analisi semantica : ultima fase di front end trova tutti gli altri errori; dichiarazioni multiple, variabili non dichiarate, *type mismatch*, argomenti passati in numero sbagliato.

Lavora in due fasi: nella prima arricchisce l'AST; attraversando l'albero in maniera top down utilizza una *symbol table* per controllare che tutte gli *statement* utilizzati siano stati dichiarati in precedenza e che non vi siano dichiarazioni multiple per lo stesso *statement*.

Nella seconda l'albero viene attraversato *bottom-up* per controllare che gli *statement* rispettino le regole del *type-checking*; tutti gli *statement* vengono processa ti nuovamente

e viene utilizzate le informazioni presenti nella *symbol table* per poter determinare il tipo di ogni espressione e trovare *type error*.

Symbol Table : struttura dati che mappa dichiarazioni a relativi usi, per poter controllare che non vi siano *statement* dichiarati più volte o usati senza essere stati dichiarati.

Ogni *entry* della *symbol table* ha una serie di attributi quali tipo di nome (classe, variabile, campo, ecc.), tipo, *nesting level*, indirizzo (dove può essere trovata a *runtime.*)

La *symbol table* è utilizzata per fare i precedenti controlli e arricchire l'AST, finito di processare tutti gli *statement* diventa inutile.

Tipo : la nozione di tipo varia da linguaggio a linguaggio; in generale si intende un insieme di valori e operazioni legali su quei valori.

Type system : il sistema di *tipaggio* di un linguaggio specifica quali sono le operazioni valide per ogni tipo del linguaggio.

Lo scopo è quello di assicurare che vengano utilizzate operazioni corrette coi tipi corretti.

Type Checking : processo che si occupa di controllare che il programma obbedisca alle regole del *type system.*

Virtual Table : *Symbol table* relativa allo *scope* interno di una classe, *virtual* perchè potrebbe contenere metodi e campi che non sono fisicamente nella classe, ma che appartengono alla classe da cui estende.

Class Table : tabella che mappa nomi di classe in rispettive *symbol table*, serve perché finita la visita della classe la *Virtual Table* relativa andrebbe distrutta ma in realtà se ci sono delle sottoclassi ce n'è ancora bisogno, quindi la salva qui.

Dispatch Table : esiste una tabella per ogni classe, essa contiene un riferimento all'indirizzo dei metodi della classe, perché i metodi possono essere chiamati anche da fuori la classe stessa con la notazione punto.

NB Ogni metodi ha lo stesso offset nella super-classe e nelle classi che lo estendono.

Object pointer : riferimento all'oggetto; è ciò che è ritornato dalla "new" punta alla posizione dir riferimento dell'oggetto; nel caso di questo compilatore punta al *dispatch pointer.*

L'*object pointer* è ciò che nel *layout* del compilatore viene puntato dall'AL per i metodi di una classe.

Dispatch pointer : puntatore alla *dispatch table* dell'oggetto; questa contiene gli indirizzi dei metodi dell'oggetto.

Liskov Principle : data una classe generica `C<T>`, essa è definita covariante rispetto a `T` se la relazione di sottotipo tra le classi `D <: B` implica la stessa relazione di sottotipo tra le classi `C<D> <: C<B>`.

Covarianza : è la capacità di accettare tipi più derivati rispetto alla classe prevista originariamente. Quindi, ad una variabile o argomento di una classe che implementi un'interfaccia generica di questo tipo, è possibile passare oggetti derivati dalla classe base inizialmente prevista.

So, covariance means that the compatibility of two types implies the compatibility of the types dependent on them.

Controvarianza : è la capacità di accettare tipi meno derivati rispetto alla classe prevista originariamente. Quindi, ad una variabile o argomento di una classe che implementi un'interfaccia generica di questo tipo, è possibile passare oggetti meno derivati (o classe base) dalla classe inizialmente prevista.

NB In generale nei tipi funzionali del progetto bisogna avere covarianza sul tipo di ritorno e *controvarianza* sui paramentri.