Relazione del Progetto di Linguaggi e Traduttori

Gabriele Quagliarella, Mat. 870773
gabriele.quagliarella@studenti.unimi.it

Emmanuel Esposito, Mat. 893364
emmanuel.esposito@studenti.unimi.it

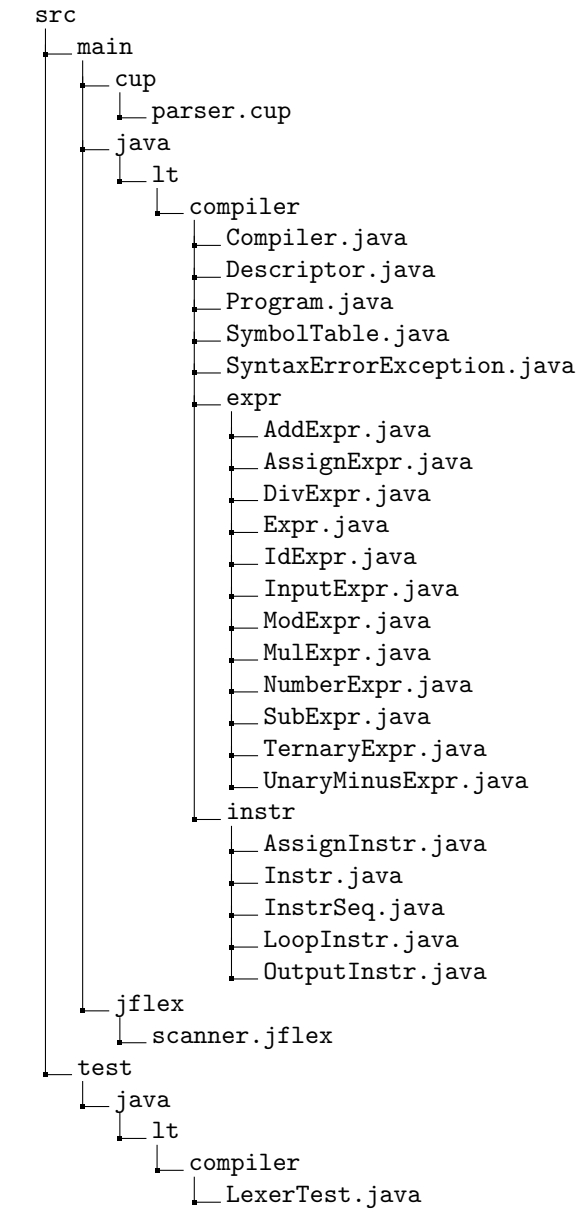
Appello di Settembre 2018

Indice

1 Informazioni preliminari

1.1 Struttura del progetto

Si riporta di seguito la struttura ad albero con cui sono organizzati i file sorgenti del compilatore, i quali si trovano sotto la directory `src`:



Nei paragrafi successivi verrà descritto il funzionamento di ciascuna classe in base al suo utilizzo durante le varie fasi di compilazione.

1.2 Build del progetto

Per poter gestire in maniera più veloce e semplice la fase di building del progetto, quest'ultimo è stato gestito utilizzando il tool Gradle. Per questo motivo, i passi per poter effettuare il building richiedono l'esecuzione del comando `./gradlew build` su sistemi Linux/macOS mentre il comando `gradlew.bat build` su sistemi Windows. Tale script genera l'archivio `ltCompiler.jar` all'interno della directory `build` del progetto.

1.3 Compilazione ed esecuzione

Per rendere più semplice ed immediata la compilazione di sorgenti e la successiva esecuzione, si consiglia rispettivamente l'utilizzo degli script `compile.sh` ed `exec.sh`. Il primo script richiede come argomenti (in ordine) il nome del file sorgente da compilare, e il nome del file oggetto da generare (output della compilazione). Il secondo, invece, richiede come argomento il file oggetto e lo esegue utilizzando `lib/ltMacchina.jar`. Sono inoltre presenti altri due script per testare il progetto. Uno di essi è `testLexer.sh`, il quale esegue il metodo `main` della classe `LexerTest` (vedi paragrafo 2.2). L'altro si chiama `compileExamples.sh`, e permette la compilazione dei sorgenti d'esempio presenti nella directory `examples/` (vedi paragrafo 5).

2 Analisi Lessicale

2.1 `src/main/jflex/scanner.jflex`

Per la realizzazione dell'analizzatore lessicale è stato utilizzato JFlex. Le direttive per la scansione di file sorgenti e la conseguente produzione di token (compatibili con il parser generato da CUP), si trovano all'interno del file di specifica `scanner.jflex`. Una volta compilato con JFlex, viene prodotto il file `src/main/java/lt/compiler/Scanner.java`, contenente la classe dello scanner. Oltre alla definizione delle espressioni regolari necessarie per il riconoscimento delle parole riservate del linguaggio e delle altre strutture lessicali (identificatori, letterali numerici, stringhe, ...), è stato inserito il codice necessario per integrare la `ComplexSymbolFactory` di CUP, generando token ad esso compatibili. Sono inoltre stati gestiti, come suggerito nella traccia del progetto, i commenti, le righe vuote e le interruzioni di riga '&' mediante la loro eliminazione (non viene restituito alcun token al loro riconoscimento).

2.2 `src/test/java/lt/compiler/LexerTest.java`

La classe `LexerTest.java` è stata scritta per poter testare l'analizzatore lessicale. Il suo funzionamento consiste nell'elencazione di tutti i token prodotti dalla classe `Scanner`, leggendo codice sorgente da standard input.

3 Analisi Sintattica

3.1 `src/main/cup/parser.cup`

L'analizzatore sintattico è stato realizzato usufruendo di CUP, e permette di generare l'Abstract Syntax Tree (insieme alla symbol table) a partire dai token prodotti dal lexer (come accennato al paragrafo 2.1). Il file di specifica contenente le regole grammaticali è nominato `parser.cup`. Dalla compilazione di questo file di specifica, vengono generate due classi dentro la directory `src/main/java/lt/compiler`. I file che contengono tali classi sono:

- `Parser.java`: il parser che riconosce la grammatica (come definita nelle specifiche del progetto).
- `ParserSym.java`: contiene le costanti enumerative corrispondenti ai simboli terminali della grammatica.

Le regole grammaticali presenti in `parser.cup` aderiscono fedelmente alle specifiche indicate nel documento di specifica. Oltre a ciò, nella sezione *parser code* sono stati inseriti alcuni metodi (Java) al fine di controllare i possibili errori sintattici identificabili in fase di parsing, per poi sollevare eccezioni appropriate. Ciò è stato ottenuto tramite l'overriding dei metodi standard usati dal parser, e definendo un'eccezione ad-hoc (`SyntaxErrorException`). Nei successivi paragrafi è presente una descrizione più dettagliata delle classi utilizzate nella rappresentazione e costruzione dell'AST.

3.2 `src/main/java/lt/compiler/Program.java`

Un'istanza della classe `Program` viene restituita per rappresentare il simbolo iniziale della grammatica, e contiene la radice dell'AST, ovvero un riferimento alla sequenza principale di istruzioni del programma, oltre che alla symbol table. Mediante l'utilizzo del metodo `getInstructions()` è possibile ottenere il riferimento alla radice della struttura come puntatore a un oggetto di tipo `InstrSeq`.

3.3 `src/main/java/lt/compiler/instr/InstrSeq.java`

Questa classe modella una sequenza di istruzioni sotto forma di linked list. L'unica operazione significativa in essa contenuta è `generateCode(Codice c)`, che si occupa di generare il codice dell'istruzione di cui tiene il riferimento e richiamare il medesimo metodo sul riferimento al resto della sequenza. Essa richiede un solo parametro, ossia un'istanza della classe `Codice` appartenente al package `lt.macchina` (presente nella libreria `ltMacchina.jar`), la quale viene usata per generare istruzioni eseguibili dalla macchina a stack di riferimento.

3.4 `src/main/java/lt/compiler/instr/Instr.java`

L'interfaccia `Instr` definisce il metodo `void generateCode(Codice c)` che sarà implementato dalle classi che modellano i vari tipi di istruzione disponibili nel linguaggio.

3.4.1 Sottoclassi di `Instr`

Nella directory `src/main/java/lt/compiler/instr`, oltre ad `Instr.java` sono presenti i file delle tre classi che modellano le istruzioni del linguaggio (implementando il metodo `generateCode(Codice c)` per tradurle). Essi sono:

- `AssignInstr.java`: permette di gestire istruzioni di assegnamento.
- `OutputInstr.java`: utilizzata per realizzare le istruzioni di output.
- `LoopInstr.java`: classe che modella l'istruzione di ciclo.

3.5 `src/main/java/lt/compiler/expr/Expr.java`

Così come `Instr.java`, anche `Expr.java` è un'interfaccia che dichiara il metodo `void generateCode(Codice c)`. Essa viene implementata dalle classi che modellano le varie espressioni ammissibili dal linguaggio.

3.5.1 Sottoclassi di `Expr`

Dentro la directory `src/main/java/lt/compiler/expr/` si trovano le sottoclassi di `Expr`, le quali permettono di rappresentare le espressioni ammesse dal linguaggio. In particolare, troviamo:

- `AddExpr.java`: modella l'operazione aritmetica di *addizione*.
- `SubExpr.java`: modella l'operazione aritmetica di *sottrazione*.
- `MulExpr.java`: modella l'operazione aritmetica di *moltiplicazione*.
- `DivExpr.java`: modella l'operazione aritmetica di *divisione*.
- `ModExpr.java`: modella l'operazione aritmetica di *modulo* (%).
- `IdExpr.java`: rappresenta variabili del linguaggio dentro espressioni.
- `AssignExpr.java`: modella l'istruzione di assegnamento come espressione.
- `NumberExpr.java`: permette di gestire i letterali numerici.
- `TernaryExpr.java`: classe usata per modellare l'operatore ternario.
- `UnaryMinusExpr.java`: usata per gestire la negazione (meno unario).
- `InputExpr.java`: modella le istruzioni di input (con eventuale prompt).

3.6 `src/main/java/lt/compiler/SymbolTable.java`

All'interno del file `SymbolTable.java` si trova l'implementazione della symbol table. I descrittori delle variabili (rappresentati da istanze di `Descriptor`) sono memorizzati nella tabella all'interno di un oggetto di tipo `Vector`. Tale scelta è dovuta al fatto che non si ha la necessità di reperire i descrittori in modo efficiente, dato che l'unico loro utilizzo è fatto nella traduzione delle istruzioni. Infatti, le istruzioni o le espressioni che necessitano il valore delle variabili possiedono già un riferimento al oggetto `Descriptor` associato all'identificatore richiesto (il quale è stato assegnato direttamente dal parser in fase di analisi sintattica). Altrimenti, la scelta più appropriata sarebbe stata una mappa associativa (e.g. `HashMap`). Sono stati messi a disposizione da `SymbolTable` dei metodi per operazioni di ricerca ed inserimento (effettivamente usati solo dal parser), i quali sono:

- `Descriptor find(String s)`: prende in input un identificatore (stringa) e restituisce il descrittore ad esso associato (oppure `null` se non presente nella symbol table).
- `void insert(Descriptor d)`: metodo usato per inserire un descrittore all'interno della tabella.
- `Descriptor findInsert(String s)`: cerca nella tabella il descrittore associato al identificatore passato come argomento, e se non presente lo istanzia ed inserisce in tabella.

3.6.1 `src/main/java/lt/compiler/Descriptor.java`

La classe `Descriptor` rappresenta la singola entry della symbol table. Ad ogni variabile è associato un unico descrittore che contiene i seguenti attributi:

- `String identifier`: nome dell'identificatore.
- `int address`: indirizzo di memoria (stack della macchina) assegnato alla variabile.
- `int length`: lunghezza che occupa la variabile (attualmente fissato ad 1), utile per sviluppi futuri nel caso di variabili che occupano più di un intero.

I metodi all'interno di questa questa classe sono esclusivamente getter e setter degli attributi, e il metodo `equals`.

3.7 `src/main/java/lt/compiler/SyntaxErrorException.java`

La classe `SyntaxErrorException` estende `RuntimeException` e viene utilizzata dal parser per segnalare errori sintattici. Gli attributi della classe sono:

- `Symbol token`: rappresenta il *token* su cui il parser si è bloccato per via di un errore sintattico.
- `int line`: linea nel file sorgente in cui si trova l'errore sintattico.
- `int column`: colonna nel file sorgente in cui si trova il token causante l'errore sintattico.
- `List<String> expectedTokens`: lista di stringhe che rappresentano i possibili token che il parser si aspetta sulla base della grammatica definita. Tale lista viene mostrata all'utente come suggerimento alla correzione.

Oltre al costruttore e ai metodi getter per ottenere gli attributi dell'oggetto, l'unico metodo interessante è `createMessage` che, sulla base degli attributi, genera una stringa di errore (principalmente usato dalla classe `Compiler`).

4 Code generation

A partire da un file sorgente scritto nel linguaggio riconosciuto, il compilatore genera un file oggetto eseguibile tramite la libreria `ltMacchina.jar` (o tramite lo script `exec.sh` per maggiore semplicità). La generazione del codice avviene in `Compiler.java` mediante la chiamata al metodo `generateCode` e la conseguente esplorazione in profondità dell'AST.

4.1 src/main/java/lt/compiler/Compiler.java

La classe `Compiler` contiene il metodo `main`, che prende come primo argomento il nome di un file sorgente, e svolge tutti i passi della compilazione fino ad ottenere il file eseguibile, il cui nome è specificato dall'utente come secondo argomento. Se non si verificano errori, il parser restituisce un oggetto di tipo `Program` che contiene sia la symbol table che l'AST. In caso contrario viene sollevata l'eccezione `SyntaxErrorException` e stampato il messaggio d'errore. In seguito, dopo aver inizializzato con valori random le prime 2 locazioni di memoria riservate per i valori temporanei (usate nel calcolo dell'operazione di modulo, %), per ogni descrittore contenuto nella symbol table viene riservato un indirizzo di memoria, e generato il codice per inizializzare tale locazione ad un valore pseudorandom. Le variabili sono allocate in modo da occupare contiguamente la porzione iniziale dello stack, spostando al contempo lo stack pointer. Infine, avviene la chiamata `instructions.generateCode(code)` che permette di scorrere in profondità l'AST e generare il codice oggetto per l'intero programma.

5 Esempi

Alcuni esempi di codice sorgente e compilato si trovano all'interno della directory `examples/`. Per ogni test è stata creata una directory contenente il file sorgente e il codice compilato in formato leggibile. Alcuni di essi (`test8` e `test9`) non compilano in quanto contengono errori di sintassi. In particolare, l'errore nel file `test9.mylang` è identificato alla riga 2, poiché la virgola dopo un assegnamento non è ammessa per poterne fare un altro sulla stessa riga:

```
1  //Swapping small numbers without extra variables
2  a = 3, b = 5
3
4  a = -(b = (a = b + a) &
5           - b)          &
6      + a
7
8  output "a = " a
9  newLine
10 output "b = " b
11 newLine
12
```

Se si tenta di compilarlo si ottiene il seguente errore:

```
[-] Syntax error at line 2, column 6: unexpected "," token
     a = 3, b = 5
          ^
[-] Expected token classes are [ADD, SUB, MUL, DIV, MOD,
    QUESTION, ENDL]
```

Per comodità, è possibile compilare tutti gli esempi (rispettando la loro organizzazione in directory) eseguendo una sola volta lo script `compileExamples.sh`. Esso genera codice oggetto per gli esempi, sia in formato eseguibile (con estensione `.o`) che in formato leggibile (con estensione `.out`).