Laboratorio di Linguaggi e Computabilità

DaveRhapsody

30 Ottobre 2019

Indice

Capitolo 1

LeX & Yacc

Sono due software complementari atti a sviluppare dei compilatori. Più precisamente sono un **Parser** ed un **Lexer**

1.1 parser

Un parser semplicemente analizza quella che è la grammatica formale (quello che vi dice se è giusto il codice che avete scritto).

- Produce un albero sintattico
- Ha bisogno di considerare il testo tramite dei Token (vedremo in seguito da dove arrivano)
- Ricorre all'analizzatore lessicale (Lexer, quello che analizza la sequenza di carattere)

Essendo due programmi distinti, sono sviluppabili entrambi. Per dire il parser lo possiamo sviluppare con C o C+. In questo caso sfrutteremo dei tool fatti apposta per generare sia il parser che il Lexer.

Come funziona? Il Lexer analizza carattere per carattere, e non fa altro che sparare al parser dei token che verranno analizzati. Più precisamente il lexer produce **grammatiche regolari** mentre il parser produce **grammatiche context-free**

Altra cosa formale: Noi si lavorerà con i generatori, per fare in modo di avere Parser e Lexer, perchè di fatto quello che accade è che si genera un automa a pila.

1. Gli si dà un file al generatore di parser
    - Quello che faremo sarà ragionare su questi file, la parte più rognosa è questa alla fine.
    - Composto da tre sezioni, ma le vedremo in seguito.
    - Spoiler:
        (a) Dichiarazioni
        (b) Regole di traduzione
        (c) Procedure ausiliarie
2. Il generatore di parser mi genera il codice sorgente del parser

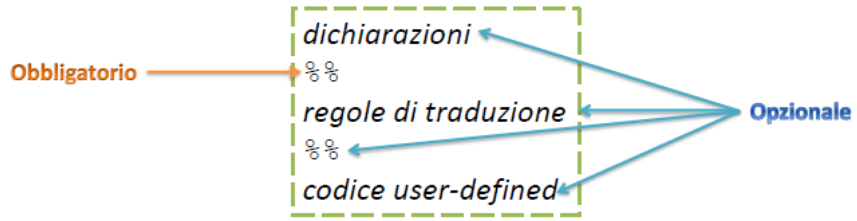
1.2 Generatore di Lexer

Come detto sopra il lexer è una specie di scanner, ed il file di input che va al lexer sarà un pattern. Ogni pattern avrà a sè associata una action, ossia un frammento di programma che ti dà un token.

Cos'è un token? Molto semplicemenente è una stringa che viene riconosciuta da un pattern. Invece Lex va a generare un programma che implementa un automa a stati finiti

1.3 La struttura di un file

Stessa struttura dei file .y usati con le versioni di Yacc per generare codice C/C++



- Minimo contenuto valido

%%

Cosa fa? Copia tutto il linguaggio preso in input dal *parser* sull'output...

1.4 Definizione della grammatica

Definiamo il concetto di simbolo Si divide in tre macrocategorie:

1.4.1 Simboli non terminali

Per convenzione vanno scritti in **minuscolo**, sono dei normali nomi ma semplicemente si scrivono in minuscolo

Esempio: Non terminali: %type nome Es.: %type string_ass

1.4.2 Simboli terminali (token)

Per convenzione questi invece vanno in MAIUSCOLO

Esempio: Terminali: %token NOME Es.: %token NUM

1.4.3 Regole sintattiche

Sono le famose 'Produzioni' già affrontate a lezione. Fondamentalmente questi simboli si definiscono nella sezione dichiarativa.

Per default: tutti i simboli sono di tipo int, ma è possibile utilizzare altri tipi di dato

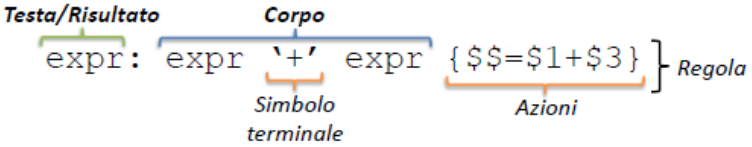
1. La classe ParserVal fornisce dei metodi che permettono l'uso di tipi di dato differenti da int
2. Associando il tipo ai simboli terminali e non terminali attraverso l'uso di parentesi angolari nei costrutti che ne permettono la dichiarazione

1.5 Definizione delle regole

Forma:

```
testa: corpo azioni
```

- **Testa** *: simbolo ***non terminale*** a cui si riduce l'espressione nella *parte destra* della regola
- **Corpo:** è composto da simboli ***terminali, non terminali***
- **Azioni:** insieme di istruzioni Java che vengono eseguite se il corpo della produzione è verificato



E' possibile elencare "parti destre" alternative che portano alla riscrittura dello stesso simbolo *Non terminale* nella parte sinistra

```
expr: expr "+" expr {$$=$1+$3} |
expr "*" expr {$$=$1*$3};
```

Se la parte destra è vuota allora la produzione viene soddisfatta anche dalla stringa vuota

```
expr: /* stringa vuota */ |
expr_1;
```

La produzione è ricorsiva se il simbolo non terminale della parte sinistra compare anche nella parte destra

```
expr_seq: expr |
expr_seq "," expr;
```

Osservazione: Dalle slides si consiglia di evitare un uso eccessivo della ricorsione, da Prolog tutto è ricorsivo, insomma qualcuno che si metta d'accorod in questo corso è difficile a trovarsi.

1.6 Produzioni ed azioni

1.6.1 L'azione

E' una parte di codice in Java che viene eseguita quando la produzione viene applicata (Riduzione). Concettualmente il valore dell'n-esimo elemento della produzione corrisponde a $n; $$, e rappresenta la parte sinistra.

SE non si specificasse nulla si avrebbe: $$=S1, inoltre il tipo associato a $n è quello dell'elemento corrispondente

Mi spiego peggio: Si può forzare qualsiasi altro tipo, come si faceva con il casting, per intenderci, e si fa usando $*Magico tipo per i dati*n

1.7 Associatività e Precedenza

E' possibile definire l'associatività dei simboli (sia a sinistra che a destra) MA allo stesso modo si può anche definire la precedenza IN BASE al'ordine delle dichiarazioni.

Disclaimer importante: (Da qui in poi conviene guardare le slides riportate nella cartella 'Materiale utile' perchè mi son reso conto che (ennesimamente) non sto facendo altro che riformulare le frasi scritte lì. ERGO, forse converrà di più guardarsi solo gli esercizi)