Linguaggi e compilatori

Corso di Laurea in Informatica

Mauro Leoncini

A.A. 2023/2024

Creazione "semi automatica" di parser $LL(1)$

- In questa esercitazione svilupperemo un *parser predittivo* LL(1) in modo "semi-automatico"
- Con questo termine intendiamo una modalità che ricorda quella già usata per realizzare *analizzatori lessicali* e che a breve ritroveremo (in strumenti maturi e con funzionalità avanzate) anche per realizzare parser "seri"
- Essenzialmente un parser LL(1) include tutte le informazioni rilevanti per il suo funzionamento nella tabella di parsing
- L'idea che quì tradurremo in concreto usando semplici grammatiche consiste nello scrivere codice per un parser LL(1) generico, al quale si possa in qualche modo "aggiungere" informazione specifica sul linguaggio voluto per realizzare un parser completo
- Come per il lexer l'informazione specifica è fornita mediante le espressioni regolari, così nel caso del parsing essa è data tramite la grammatica e tradotta poi proprio nella parsing table

Creazione "semi automatica" di parser $LL(1)$

- Per trasformare la grammatica in tabella di parsing, usiamo due strumenti sviluppati "in casa" ma di cui, esattamente come per `Lex`, possiamo benissimo ignorare tutti i dettagli, se non sapere come vadano usati
- Da quanto abbiamo già appreso a lezione, sappiamo che ci sono tre passi fondamentali che andranno (possibilmente) automatizzati:
  - a partire dalla grammatica, il calcolo degli insiemi *FIRST* e *FOLLOW* per tutti i simboli non terminali;
  - mediante l'informazione contenuta in tali insiemi, e naturalmente ancora le produzioni della grammatica, la verifica che quest'ultima sia effettivamente di tipo LL(1);
  - in caso affermativo, la costruzione vera e propria della tabella di parsing.
- Oltre a ciò deve essere definito il modo mediante il quale la tabella di parsing diviene accessibile al "parser generico" in modo che questo possa essere specializzato concretamente sul particolare linguaggio.

Creazione "semi automatica" di parser $LL(1)$

- Per gli scopi esposti nella slide precedente, abbiamo realizzato due programmi `Python`, laddove la scelta di questo linguaggio è stata effettuata per "mera comodità di sviluppo"
- Il primo programma si chiama `first_and_follow.py` e naturalmente svolge il compito sotteso dal nome.
- `first_and_follow.py` può essere usato da solo, per scopi didattici, e per questo può produrre una stampa su standard output degli insiemi *FIRST* e *FOLLOW* della grammatica data in input
- Usato nel contesto della generazione di un parser, `first_and_follow.py` può serializzare gli insiemi in un file `pickle`, che viene letto dal secondo programma di nome `makeparser.py`
- Quest'ultimo produce il suo output in un file con estensione `hpp` che il parser generico dovrà inportare e che contiene le informazioni per specializzarlo sul particolare linguaggio

Semplice grammatica per le espressioni

- Vediamo come si presenta l'output "didattico" di `first_and_follow.py` nel caso della grammatica:

```
> cat expr3.ll1
F : 'number'
F : '(' E ')'
Y : '*' F Y
Y : EPS
T : F Y
B : '+' T B
B : EPS
E : T B
```

... e i relativi insiemi *FIRST* e *FOLLOW*

```
> python3 first_and_follow.py -p expr3.ll1
NT          First                   Follow
-------------------------------------------------
F   {(, tok_number}   {), *, +, tok_eof}
E   {(, tok_number}         {), tok_eof}
Y         {EPS, *}       {), +, tok_eof}
T   {(, tok_number}       {), +, tok_eof}
B         {EPS, +}         {), tok_eof}
```

Creazione "semi automatica" di parser $LL(1)$

- Per questa grammatica è possibile costruire la tabella di parsing $LL(1)$
- Il programma `makeparser.py` legge il file `.pickle` che contiene i dati della grammatica (terminali, non terminali, assioma, produzioni) e gli insiemi *FIRST* e *FOLLOW* e produce i file `tokens.h` e `pparser.hpp`
- `tokens.h` contiene la definizione dei token e dovrà essere importato da lexer e parser
- `pparser.hpp` contiene dati per il parser, che dipendono dal languaggio e che dunque non possono essere incluse nel codice generico
  - Definizione numerica dei non-terminali
  - Definizione delle produzioni
  - Rappresentazione "human readable" delle produzioni (da usare nella descrizione del processo di parsing)
  - Una funzione che costruisce la tabella di parsing

La prima grammatica (da rivedere) di Kaleidoscope

```
> cat kaleidoscope1.ll1
numexpr: 'number'
parexpr: '(' expr ')'
idexpr: 'id'
idexpr: 'id' '(' optexpr ')'
optexpr: exprlist
optexpr: EPS
exprlist: expr
exprlist: expr ',' exprlist
primary: idexpr
primary: numexpr
primary: parexpr
binoprhs: EPS
binop: '+'
binoprhs: binop primary binoprhs
binop: '*'
expr: primary binoprhs
proto: 'id' '(' idseq ')'
idseq: 'id' idseq
idseq: EPS
def: 'def' proto expr
external: 'extern' proto
top: def
top: external
top: expr
top : EPS
program : top ';' program
program : EPS
```

... i relativi insiemi *FIRST* ...

Non Terminals	First
numexpr	{tok_number}
expr	{(,tok_id,tok_number}
parexpr	{(}
idexpr	{tok_id}
optexpr	{(,EPS,tok_id,tok_number}
exprlist	{(,tok_id,tok_number}
primary	{(,tok_id,tok_number}
binop	{*,+}
binoprhs	{EPS,*,+}
idseq	{EPS,tok_id}
proto	{tok_id}
def	{tok_def}
external	{tok_extern}
top	{EPS,(,tok_extern,tok_def,tok_id,tok_number}
program	{EPS,(,;,tok_extern,tok_def,tok_id,tok_number}

... e gli insiemi *FOLLOW*

Non Terminals	Follow
numexpr	{), *, +, ,, ;}
expr	{), ;, ,}
parexpr	{), *, +, ,, ;}
idexpr	{), *, +, ,, ;}
optexpr	{)}
exprlist	{)}
primary	{), *, +, ,, ;}
binop	{(, tok_id, tok_number}
binoprhs	{), ;, ,}
idseq	{)}
proto	{(, ;, tok_id, tok_number}
def	{;}
external	{;}
top	{;}
program	{tok_eof}

La grammatica non è $LL(1)$

- La nostra prima grammatica (pur parziale) non è $LL(1)$.
- La ragione è semplice e riguarda il non terminale `idexpr`
- Le due produzioni per `idexpr` iniziano infatti con lo stesso terminale `id` e dunque la violazione di una delle due condizioni necessarie (e sufficienti) perchè si possa costruire un parser predittivo con un solo carattere di lookahed è di immediata evidenza
- Possiamo naturalmente costruire il lexer, calcolare gli insiemi *FIRST* e *FOLLOW* ma l'esecuzione di `makeparser.py` rileva il conflitto (al riguardo è utile esercitarsi con i programmi presenti nella cartella condivisa `setslide8`)

Esercizio

- Siamo i progettisti del linguagio, quindi possiamo prendere qualsiasi decisione che ci sembra più opportuna
- Come andiamo dunque a modificare la grammatica in modo da soddisfare i vincoli per il parsing predittivo $LL(1)$?
- Si può prendere spunto da linguaggi reali che si conoscono