Corso di Laurea Magistrale in Ingegneria Informatica
A.A. 2011-2012

Linguaggi Formali e Compilatori

YACC, BISON

Giacomo PISCITELLI

Ricapitolando . . .

Un parser è un programma che determina se il suo input è sintatticamente valido e determina la sua struttura. I parser possono essere scritti a mano o generati automaticamente da *parser generator*, mediante descrizioni di strutture sintatticamente valide di una grammatica context-free.

I parser generator possono essere usati per realizzare una vasta gamma di analizzatori di linguaggi, da quelli di semplici calcolatrici da tavola a complessi linguaggi di programmazione.

Il parser, che è realizzato come automa a pila, raggruppa i token (simboli terminali) in unità sintattiche (simboli non terminali). Quando viene riconosciuta una struttura sintattica, una versione estesa del parser di solito costruisce un albero sintattico (AST), che è una rappresentazione concisa della struttura del programma e guida la successiva analisi semantica.

Più in dettaglio il parser produce una rappresentazione del programma in forma di parse tree. La fase successiva del processo di compilazione esegue un'analisi context-sensitive (spesso chiamata analisi semantica statica) delle variabili e delle altre entità del parse tree. Tale fase è spesso combinata con l'analisi sintattica e l'informazione utilizzata per il *context-sensitive checking* è quella contenuta nella *symbol table*.

L'output di tale fase è un *annotated parse tree*. Per descrivere la semantica statica di un programma vengono usate le cosiddette grammatiche ad attributi.

Il Contesto

I file Lex/Flex e Yacc/Bison possono essere estesi per consentire la gestione dell'informazione context-sensitive: per esempio, si voglia che le variabili siano dichiarate prima di essere referenziate.

Il parser deve dunque essere in grado di consentire il confronto tra riferimento e dichiarazione di una variabile: un modo per fare ciò consiste nel costruire una lista delle variabili durante l'analisi della parte dichiarativa del programma e poi controllare la presenza, nella lista suddetta, di una variabile presente nella parte esecutiva del programma.

Una tale lista è detta **symbol table**; le symbol table (possono anche essere una per ogni tipo di token) possono essere realizzate usando liste, alberi, hash-table.

Indipendentemente da come si realizza, per costruire elementi della symbol table, necessari per costruire una grammatica ad attributi (in particolare, per ciascuna variabile, almeno gli attributi nome-valore-tipo-visibilità), bisognerà, quando lo scanner individua un token identificatore, assegnare a tale token il valore della variabile globale `yylval`.
In ogni azione semantica, si può esaminare il valore semantico `yylval` e l'indirizzo `yylloc` (se sono definiti) del token.

Generatori di analizzatori sintattici: YACC o Bison

Yacc (**Bison**) sono tool per generare parser, disponibili su Unix (o anche Windows).

Bison è una versione open source e più veloce di Yacc.

Quando si scrive un programma in Yacc/Bison, si descrivono le produzioni della grammatica del linguaggio da riconoscere e le azioni da intraprendere per ogni produzione.

Sebbene Yacc/Bison possa trattare numerose importanti sottoclassi delle grammatiche di tipo 2, Bison in particolare è ottimizzato per trattare le grammatiche con un metodo di analisi sintattica bottom-up, noto come **LARL(1)**[1], cioè con derivazione rightmost rovesciata (il non terminale più a destra viene sostituito per primo) e con un solo simbolo di lookahead.

Il parser usa l'analizzatore lessicale per prelevare dall'input i token e riorganizzarli in base alle produzioni della grammatica utilizzata.

Quando una produzione viene riconosciuta, viene eseguito il codice ad essa associata.

[1] There are various important subclasses of context-free grammar. Although it can handle almost all context-free grammars, Bison is optimized for what are called LALR(1) grammars. In brief, in these grammars, it must be possible to tell how to parse any portion of an input string with just a single token of lookahead. Strictly speaking, that is a description of an LR(1) grammar, and LALR(1) involves additional restrictions that are hard to explain simply; but it is rare in actual practice to find an LR(1) grammar that fails to be LALR(1).

YACC/BISON: struttura del programma

Ogni programma Yacc/Bison consta di tre sezioni: un prologo (o parte dedicata alle dichiarazioni, le regole (o parte dedicata alla descrizione delle strutture sintattiche della grammatica context-free considerata) e un epilogo (o parte dedicata alle routine ausiliarie) ed ha il seguente aspetto:

Prologo o Dichiarazioni

%%

Regole o Strutture sintattiche

%%

Epilogo o Sezione routine ausiliarie

La sezione delle regole è l'unica obbligatoria.

La forma più intellegibile per presentare le regole è quella di Backus-Naur o BNF.

Ogni grammatica espressa mediante BNF è una grammatica context-free. Pertanto l'input a Yacc/Bison è essenzialmente una BNF machine-readable.

I caratteri di spaziatura (blank, tab e newline) vengono ignorati.

I commenti sono racchiusi, come in C, tra i simboli /* e */

YACC/BISON: sezione Prologo

Nella **sezione Prologo** si definiscono alcune informazioni globali da dover usare per interpretare la grammatica.

Il Prologo può comprendere:

- una "black box" di definizioni ausiliarie, costituita da dichiarazioni C racchiuse tra i delimitatori `%{` e `%}`, i.e.

  `%{` **`#include`** e altre macro e direttive C, costanti, variabili `%}`

- una "white box", costituita da nomi di simboli terminali e non terminali, regole di precedenza/associatività tra simboli, ...

Tramite l'istruzione:

$$\%token_1\ token_2\ \ldots token_n$$

si definiscono quali sono i token inseriti nelle regole che sono il risultato dell'analisi lessicale.

Tramite l'istruzione:

```
% start assioma
```

si definisce qual è il non terminale della grammatica da considerare come assioma (per default il primo non terminale incontrato).

YACC/BISON: sezione Regole

La **sezione Regole** è composta da una o più produzioni espresse nella forma:

```
A : BODY ;
```

dove **A** rappresenta un simbolo non terminale e **BODY** rappresenta la modalità costruttiva del simbolo non terminale **A**, attraverso la sequenza di uno più simboli sia terminali che non terminali che concorrono a costruirlo.

I simboli : e ; sono separatori.

Nel caso la grammatica presenti più produzioni (o più sequenze costruttive) per lo stesso simbolo terminale, queste possono essere scritte senza ripetere il non terminale usando il simbolo |

YACC/BISON: Azioni

Ad ogni regola può essere associata un'**azione** che verrà eseguita ogni volta che la regola venga riconosciuta.

Le azioni sono istruzioni C e sono raggruppate in un blocco.

```
A : B C D {printf ("ciao")}
```

Le azioni possono apparire ovunque nel body di una regola.

Le azioni possono scambiare dei valori con il parser tramite delle pseudo-variabili introdotte dal simbolo (**$$**, **$1**, **$2**, …)

```
A : B {$$ = 1} C {$1 = 2; $2 = 12}
```

La pseudo-variabile **$$** è associata al lato sinistro della produzione mentre le pseudovariabili **$n** sono associate al non terminale di posizione **n** nella parte destra della produzione.

Per quanto attiene l'Epilogo, esso può contenere tutto il codice utile, incluso quello delle funzioni dichiarate nel Prologo.

YACC/BISON: Parser

Il parser generato da Yacc/Bison è un automa a stati finiti di tipo push-down in grado di avere un token di lookahead.

L'automa ha solo 4 azioni: **shift**, **reduce**, **accept** ed **error**.

In base allo *stato corrente* (simbolo sul top dello stack) il parser decide se necessita di un token di lookahead (ottenibile usando yylex).

Usando lo stato corrente ed il token di lookahead, il parser decide quale azione intraprendere e la espleta.

YACC/BISON: azioni dell'automa

Azioni di **shift**: usano sempre un token di lookahead, e consistono nel confrontare tale token con il token corrente ed in caso di match fare *pop-up* dello stato dallo stack, inserire il nuovo stato e saltare il token di lookahead.

Azioni di **reduce**: evitano il crescere incontrollato dello stack, e sono usate quando il parser esamina il lato destro di una produzione e lo sostituisce con il lato sinistro della stessa. Può servire un token di lookahead.

Azione di **accept**: l'input appartiene al linguaggio descritto dalla grammatica.

Azione di **error**: l'input si è rilevato non appartenente al linguaggio descritto dalla grammatica.

YACC/BISON: ambiguità e conflitti

Le produzioni di una grammatica possono essere ambigue, come ad esempio:

```
E : E + E
```

Infatti se in input si ha la stringa `E + E + E`, è possibile interpretarla sia come

`E + (E + E)` che come `(E + E) + E`.

Yacc/Bison è in grado di accorgersi di tale ambiguità.

Il parser può:

- applicare un'azione di reduce alla parte di stringa `E + E` ottenendo `E` e quindi riapplicare tale azione;

oppure può:

- applicare un'azione di shift `E + E` e quando ha letto tutta la stringa applicare le due azioni di reduce a partire dalla seconda coppia `E + E`.

Questo tipo di situazione è detta conflitto di tipo shift-reduce.

In modo analogo è possibile avere anche conflitti di tipo reduce-reduce.

Nel caso Yacc/Bison rilevi un conflitto, produce lo stesso un parser effettuando delle scelte su quale azione intraprendere per prima.

YACC/BISON: ambiguità e conflitti

Le regole adottate per risolvere tali conflitti sono:

- in un conflitto shift-reduce si da la precedenza all'azione di shift;
- in un conflitto reduce-reduce si da la precedenza alla regola che viene incontrata per prima.

E' sempre bene evitare i conflitti alla base. Questo è possibile riscrivendo la grammatica.

Un altro modo per risolvere i conflitti, o per lo meno per pilotarne la risoluzione, è quello di definire l'associatività dei simboli ambigui, tramite le istruzioni **%rigth** e **%left** da inserire nella sezione dichiarazioni.

Esempio

```
%rigth "="
%left "+" "-"
%left "*" "/"
```

stessa precedenza, associati a sinistra

precedenza crescente nella lista

In questo caso si stabilisce l'associatività (se a destra o a sinistra) di =, +, - , *, / e inoltre si definisce anche che "*****" e "**/**", anche se associano dallo stesso lato, hanno una priorità minore.

YACC/BISON: opzioni

per rendere visibile all'esterno la definizione dei token

-d (header) : genera `file.tab.h` = dichiarazioni delle informazioni esportabili

```
#define YYSTYPE int

Simboli per Lex

extern YYSTYPE yylval;
```

-V (verbose) : genera file.output = descrizione della tabella di parsing LALR(1)

Pragmaticamente: **Prima**: si lancia Yacc sulla G nuda (senza azioni semantiche, procedure ausiliarie, ecc.) per accertarsi che il parser generato sia conforme alle aspettative.
Poi: completamento.

YACC/BISON: identificatori e definizioni

- Identificatori interni di Yacc:

Identificatore	*Descrizione*
`file.tab.c`	Nome file output
`file.tab.h`	Header file generato da Yacc (mediante **-d**), contenente le `#define` dei token
`yyparse()`	Funzione di analisi sintattica
`yylval`	Valore del token corrente (nello stack)
`YYSTYPE`	Simbolo per il preprocessore C che definisce il tipo dei valori calcolati dalle azioni semantiche
`yydebug`	Variabile intera che abilita la traccia dell'esecuzione del parser generato da Yacc

- Definizioni Yacc

Keyword	*Definizione*
`%token`	Simboli di preprocessing per i token
`%start`	Assioma
`%union`	Union YYSTYPE per permettere di computare valori di tipo diverso nelle azioni semantiche
`%type`	Tipo variante associato ad un certo simbolo grammaticale
`%left`	Associatività sinistra dei token
`%right`	Associatività destra dei token
`%nonassoc`	Non associatività