YACC

José João Almeida
Fernando Jorge Ferreira

92/10/08

Contents

1 YACC - gerador de parsers

YACC é um comando UNIX. YACC significado - yet another compiler-compiler - (o título é exagerado).

O yacc recebe um texto com a descrição de uma gramática com acções semânticas (i.e. acções escritas em **C** que indicam o que é que se deve fazer cada vez que se reconheceu uma produção [1]) e gera um reconhecedor sintatico do tipo SLR(1) correspondente. O yacc é, portanto, um gerador de programas.

O texto de entrada pode ainda incluir procedimentos/ funções auxiliares. O texto do reconhecedor gerado (codigo C) pode ser junto a outros modulos e ou transportado para outros ambientes.

1.1 Gramática do YACC

```
textoYacc       -> <definicoes> "%%" <producoes> <codigoCaux>
                ;
<definicoes>    -> <def>*
                ;
<producoes>     -> <producao>+
                ;
```

1.1.1 Zona das definições

Em <definicoes> são definidos os seguintes elementos referentes à gramática:

- Axioma da gramática

[1] além das habituais intruções **C** existe suporte para associação e cálculo de valores associados aos símbolos da gramática: gramática de atributos (só com atributos sintetizados)

- Conjunto dos terminais (**T**) e respectivo tipo de valor (caso exista)
- Tipo de valor associados a cada não terminal (**NT**) (caso exista)

elementos referentes a resolução automatica de conflitos e ambiguidades:

- Indicação de prioridades de operadores
- Indicação de associatividade de operadores (à esquerda/ direita/ não associatividade)

codigo **C** global

```
<def>          -> <defAxioma>
               |  <defTiposdeValor>
               |  "%{" <codigoCglobal>  "%}"
                   |  <defTerminais>
                   |  <defAssociatividade>
                   |  <defTipodeValorT_NT>

 <defAxioma> -->  "%start" ID
```

Quando defAxioma é omitido, o lado esquerdo da primeira produção é tomado como axioma.

```
<defTiposdeValor> -> "%union" <unionDef>
```

Definição de tipos de valores para **T** e **NT**. Esta definição sera' usada internamente para a declaração de uma stack valor. <unionDef> é uma declaração de uma **union C**; <unionDef> vai definir um conjunto de identificadores de tipos que poderão ser usados em <tipo>.

```
<defTerminais> -> "%token" <tipo> <T>*
```

Define os terminais da gramática podendo ainda associar-lhes um tipo de valor. Essa definição pode ser feita em varias linhas

```
<tipo>         ->
               |  "<" ID ">"
```

Tipo é opcional; quando é usado, **ID** é um identificador que deve figurar na <unionDef> da <defTipos_T_NT>

```
<defAssociatividade> -> "%left" <tipo> <T_NT>*
                     |  "%rigth" <tipo> <T_NT>*
                     |  "%noassoc" <tipo> <T_NT>*
```

Indica se um operador é associativo ou não. Esta informação é usada para tratamento de conflitos. Note-se portanto que podem ser tratadas gramáticas ambiguas. A cada símbolo **S** fica associada uma prioridade de acordo com a ordem em que ele aparece:

- Os **S** declarados na mesma linha tem igual prioridade
- Os **S** declarados na linha **i** tem prioridade maior que os da linha i–1

```
<defTipodeValorT_NT> -> "%type" <tipo> <NT>*
```

Permite associar um tipo de valor a um terminal ou não terminal.

1.1.2 Produções

```
<producao>      -> <NT> ":" <rhss>
```

produção da gramática segundo a estrutura sintatica habitual.
<NT> - não terminal.
rhs (=rigth hand side) - lado direito da produção

```
<rhss>          -> <rhs>
                |  <rhs> "|" <rhss>
```

Lado direito possivelmente multiplo e nesse caso separado por "|"

```
<rhs>           -> <simb>*
                |  <simb>* <prec>
```

Lado esquerdo é uma sequencia de símbolos podendo opcionalmente incluir **<prec>** para alterar prioridades de operadores.

```
<simb>          -> <T>
                |  <NT>
                |  <simbSemantico>
```

T - símbolo terminal
NT - símbolo não terminal
símboloSemântico (para colocar acções no parser)

```
<T>             -> ID_token
                |  "'" char "'"
```

T- símbolo terminal é um identificador declarado como token ou caracter entre pelicas.

```
<NT>            -> ID
```

O identificador **ID** deve figurar em pelo menos um lado esquerdo de uma produção

```
<simbSemantico> -> "{" <codigoCcom$> "}"

  <codigoCcom$>   -> [codigo C com possibilidade de usar $$, $1, ...]
```

`$$` permite a associação de um valor ao lado esquerdo da produção (`$$ = ..` – atributo sintetizado)
`$i` permite a utilização de um valor dum símbolo do rhs (**i** - ordem do símbolo) esse símbolo pode ser **T**, **NT** ou **símboloSemântico**

```
<prec>          -> "%prec" <T>
```

Permite associar a prioridade de um símbolo terminal **T** a uma produção.

1.1.3 Codigo C

```
<codigoCaux>    -> "%%" <funcoes_C>
                |                       /* vazio */

<funcoes_C>     -> [codigo C puro; nao esquecer a definicao da funcao
                    de  analise lexica yylex() ex: #include "lex.yy.c"]
```

1.2 Exemplo: calculadora tipo LISP

Calcular o valor de uma expressão **S** incluindo multiplicações e adições

```
(* 12 12) = 144
(* 1 (+ -1 2 3)) = 4
(*) = 1
```

1.2.1 texto yacc

```
-- cat calc.y -----------------------------------------------------------

        %{
        #include <stdio.h>
        typedef struct cel{
          int at;
          struct cel *next;} *seq;
        %}
        %start axioma

        %union {
          int ival;
          seq seqv;
        }
        %token <ival> INTEIRO
        %token '(' ')' '*' '+'            /* Linha nao obrigatoria */
        %type <ival> sexp
        %type <seqv> lista


        %%
        axioma : sexp { printf ("= %d\n",$1);}
               | sexp { printf ("= %d\n",$1);} axioma
               ;

        sexp : INTEIRO  {$$ = $1;}
             | '(' '+' lista ')'   {$$ = soma($3);}
             | '(' '*' lista ')'   {$$ = prod($3);}
             ;

        lista : sexp lista   {$$ = cons($1,$2);}
              |              {$$ = makelista();}
              ;
        %%

        int soma ( s)   /* calcula soma e liberta a lista */
        seq s;
```

```
{ int som = 0;
  seq aux;
  for ( ;s!=NULL; aux = s,s= s->next,free(aux) ) som += s->at;
  return (som);
}

int prod ( s)  /* calcula produto e liberta a lista */
seq s;
{ int pro = 1;
  seq aux;
  for ( ;s!=NULL; aux=s, s= s->next, free(aux) ) pro *= s->at;
  return (pro);
}

seq makelista()
{return ((seq)NULL);}

seq cons(ele,s)
seq s;
int ele;
{seq aux;
 aux = (seq) malloc(sizeof(struct cel));
 aux->at = ele;
 aux->next = s;
 return(aux);
}

#include "lex.yy.c"
```

Na ultima linha aparece a inclusão do analisador lexico (neste caso gerado pelo LEX e cujo texto se apresenta de seguida).

```
-- cat analex.l -----------------------------------------------------------

DIG [0-9]
INT ([+-]?{DIG}+)

%%
{INT}           {yylval.ival=atoi(yytext) ; return (INTEIRO);}
[\(\)\*\+]      {return (yytext[0] );}
.|\n            ;
%%
--------------------------------------------------------------------------
```

yylval : elemanto do tipo **%union** do texto yacc, usado para passar o valor associado ao símbolo terminal reconhecido (INTEIRO)

INTEIRO: %token do texto yacc

1.2.2 Utilização

Usando makefile: criar um ficheiro makefile com as seguintes regras:

```
> cat makefile
        calc : calc.o
                cc -o calc calc.o -ly -ll

        calc.o : lex.yy.c

        lex.yy.c: analex.l
                lex analex.l

> make
        lex analex.l                    cria lex.yy.c
        yacc  calc.y                    cria y.tab.c
        cc -O -c y.tab.c                compila y.tab.c
        rm y.tab.c
        mv y.tab.o calc.o
        cc -o calc calc.o -ly -ll       linka usando as lib. do lex e yacc
```

Exercício: acrescentar as funções exp log sin cos. Exemplo:
(+ 3 (sin 2) (cos 1)) = 3 + sin(2) + cos(1)

1.3 Exemplo: construção duma mini shell

Como foi referido, uma *shell* é um programa que interpreta um texto (introduzido interactivamente ou não) e executa os comandos indicados usando as primitivas do sistema operativo.

Neste exercício pretendemos algo ligeiramente mais simples: ler comandos, traduzi-los para "csh" e executá-los através da "csh".

Seguem-se exemplos da sintaxe pretendida e respectiva tradução:

```
+--------------------------------+-----------------------------------+
| TLCshell                       | Cshell                            |
+--------------------------------+-----------------------------------+
| a                              | cat a                             |
| a + b                          | (cat a ; cat b)                   |
| c = a + b + d                  | ((cat a ; cat b);cat d) > c       |
| sort(a)                        | cat a | sort                      |
| lp(pr(banner("tit")+nl(f)))    | (banner "tit";cat f| nl) | pr | lp |
+--------------------------------+-----------------------------------+
```

Para tal construa uma gramática tradutora e respectiva implementação usando Lex e Yacc.

Tlcshell : texto Yacc

```
%{
#include <stdio.h>
FILE *fp;
#define iniciaPipeParaCsh  fp=popen("csh","w")
#define closePipe             pclose(fp)
#define enviaStrParaCsh(st)   printf("%s\n",st);\
                     fprintf(fp,"%s\n",st);fflush(fp)
%}
%union{ char stri[200];}
%token BANNER GREP
%token <stri> STR ID
%type  <stri> comando Exp

%%
tlcshell : {iniciaPipeParaCsh;}
           comandos
           {closePipe;}
         ;
comandos : comando {enviaStrParaCsh($1);} ';' comandos
         |
         ;
comando  : ID '=' Exp               {sprintf($$," %s > %s ", $3,$1);}
         | Exp                      {sprintf($$," %s ",$1);}
         ;
Exp      : ID                       {sprintf($$," cat %s ",$1);}
         | BANNER '(' STR ')'       {sprintf($$," banner %s ",$3);}
         | Exp '+' Exp              {sprintf($$,"(%s ; %s ) ",$1,$3);}
         | ID '(' Exp ')'           {sprintf($$," %s | %s ",$3,$1);}
         | GREP '(' STR ',' Exp ')' {sprintf($$," %s |grep %s ",$5,$3);}
         | '(' Exp ')'              {sprintf($$," %s ",$2);}
         ;
%%
#include "lex.yy.c"
```

Tlcshell: texto lex

```
%%
grep                                   return GREP;
banner                                 return BANNER;
[(),=+;]                               return yytext[0];
[a-zA-Z][0-9a-zA-Z.]*     {strcpy(yylval.stri,yytext);
                                       return ID;}
\"[^"]*\"                 {strcpy(yylval.stri,yytext);
                                       return STR;}
```

```
" "|\n                                  ;
%%
```

Exemplo

A seguinte expressão:

```
--------------------------------------------------------------------------
| x=pr(qua(banner("tlc:lex"))+nl(x.l)+qua(banner("tlc:yacc"))+nl(x.y));  |
--------------------------------------------------------------------------
```

seria traduzida para:

```
--------------------------------------------------------------------------
| (  banner "tlc:lex"  | qua  ; (  cat x.l  | nl  ; (  banner "tlc:yacc" |
| | qua  ;   cat x.y  | nl  )  )  )  | pr  > x                           |
--------------------------------------------------------------------------
```

e executada através da Csh (com que resultados? suponha que "qua" é um comando que enquadra um texto).