

Compiladores

Ficha prática 3 – YACC

Licenciatura em Engenharia Informática
Universidade de Coimbra

Ano Letivo 2015/2016

O YACC (Yet Another Compiler-Compiler) é um “compilador de compiladores” que, juntamente com um analisador lexical (o lex, tipicamente), permite gerar automaticamente um compilador. Como veremos, com o YACC é possível criar compiladores extremamente potentes.

Para além de outros dados, o ficheiro de especificação do YACC deverá conter a gramática correspondente à linguagem que se quer compilar, que o YACC processa usando análise ascendente (*Shift/Reduce parsing*).

O YACC foi concebido para ser usado com o Lex. O Lex identifica os tokens e envia-os (juntamente com o seu valor, caso sejam diferentes¹) ao YACC, que os processa de acordo com o especificado na gramática. Para assim suceder, é necessário respeitar um conjunto de normas na construção das especificações Lex e YACC. Tal como o Lex, o ficheiro de especificação do YACC tem o seguinte formato:

```
...definições...
%%
...regras...
%%
...subrotinas...
```

Suponhamos que vamos fazer uma pequena calculadora com inteiros. Esta calculadora funcionará na linha de comandos (ou a partir de um ficheiro) recebendo expressões simples e imprimindo os resultados. Por exemplo:

```
[input] 2+4*2+1*2
[output] = 12
```

Na parte das definições, interessa-nos por agora declarar os tokens esperados. Os tokens possíveis serão números e operadores. O YACC atribui a cada token um valor inteiro, que corresponderá a uma constante definida no ficheiro *y.tab.h* (gerado automaticamente). Por exemplo, o token NUMBER poderá ter o valor 258. Note-se que os números de 0 a 256 são **sempre** atribuídos aos respectivos caracteres ASCII (+EOF), pelo que, quando o lex processa padrões de apenas um carácter, pode enviar simplesmente o próprio padrão ao YACC (é o

¹Por exemplo, suponhamos que encontra número 15. O lex identifica o token “INTEIRO”, mas o seu valor é 15.

exemplo dos operadores, que só têm um caracter). Assim, teremos apenas que declarar o token NUMBER:

```
%token NUMBER
```

Depois, podemos descrever as regras gramaticais que representam as expressões aritméticas. Um regra em YACC tem sempre um símbolo não terminal do lado esquerdo, seguido de “:” e de símbolos terminais e não terminais do lado direito (eventualmente utilizando “|” para representar disjunção de regras). Como convenção, tem-se assumido a representação dos símbolos terminais (tokens) em maiúsculas e os não terminais em minúsculas. Aqui vai um exemplo da gramática para a calculadora simples:

```
%%
expression: expression '/' expression {$$=$1/$3;}
           | expression '*' expression {$$=$1*$3;}
           | expression '+' expression {$$=$1+$3;}
           | expression '-' expression {$$=$1-$3;}
           | NUMBER                    {$$=$1;}
           ;
```

Nesta gramática, temos uma definição recursiva de expression que permite um tamanho infinito de operações aritméticas. Repare que, em cada regra, existe uma acção que corresponde à respectiva semântica. Por exemplo, uma soma consiste na soma dos valores já processados. Atenção que \$\$ significa o valor que irá ser colocado no topo da pilha (do analisador), enquanto \$1,\$2,\$3,...,\$i correspondem aos valores dos argumentos índice 1,2,3,...,i na regra (que são retirados da pilha do analisador, ou seja correspondem ao primeiro, segundo, terceiro,..., i-ésimo elementos a contar do topo da pilha). Por exemplo, na regra

```
frase:  sujeito verbo complemento
```

O símbolo “sujeito” é identificado por \$1, o símbolo “verbo” é identificado por \$2 e o símbolo “complemento” é identificado por \$3.

Repare-se também na última regra (`\NUMBER {$$=$1;}`). Em linguagem informal, podemos interpretar da seguinte forma: “Se o lex encontrar um token NUMBER, o valor correspondente deve ser colocado directamente no topo da pilha”. Nesta situação, pretende-se passar o valor de NUMBER para as outras regras. Estes pormenores serão clarificados nos exemplos.

Na parte das subrotinas, poderemos colocar as funções *main* e *yyerror*².

```
%%
int main() {
  yyparse();
  return 0;
}
```

²Dependendo da versão do Yacc que utilize, pode omitir a inclusão explícita destas funções desde que compile posteriormente com o argumento -ly.

```
void yyerror (char *s) {
printf ("%s\n", s);
}
```

Do lado do lex, terá então que identificar os tokens. Se definiu tokens no ficheiro YACC (como mostrámos atrás), deverá incluir o *header* “y.tab.h” nas definições do lex (`#include "y.tab.h"`). De cada vez que identifica um token, o lex deverá fazer um “return” desse token.

Para o envio do valor do token para o YACC, o Lex tem que se socorrer de uma variável que é partilhada (pelo Lex e pelo YACC). No exemplo mais simples, imaginemos que queremos identificar inteiros (tokens NUMBER, que correspondem a padrões de um ou mais dígitos). Assim que o Lex descobre um padrão de inteiro, pode enviar ao YACC essa informação (`return NUMBER`). No entanto, para além de saber que foi encontrado um inteiro, o YACC pode querer o seu *valor*, tal como acontece com a regra `\NUMBER {$$=$1;}` referida acima. Aí, entra a variável *yylval*.

Por default, *yylval* é um inteiro (definido no ficheiro *y.tab.h*). Cá está o ficheiro *lex* para o nosso exemplo:

```
%{
#include "y.tab.h"
}%
%%
[0-9]+      {yylval=atoi(yytext); /*Guarda valor em yylval e*/
              return NUMBER;}      /*envia token reconhecido ao
                                   YACC*/
\n          {return 0;}           /*Fim = sinal de EOF para
                                   YACC */
[\\t]       ;                    /*Ignorar espaço e tab*/
.           return yytext[0];    /*Caso seja qualquer outro */
                                   /*caracter (por exemplo um */
                                   /*operador), enviar para */
                                   /*o YACC*/

%%
int yywrap()
{
return 1;
}
```

Assumindo que os ficheiros criados são *mycalc.l* e *mycalc.y*, para criar então a nossa mini calculadora, terá que executar as seguintes instruções:

```
lex mycalc.l
yacc -d mycalc.y
cc -o mycalc y.tab.c lex.yy.c -ll -ly
```

Esta é a sequência necessária para criar um programa com o lex e yacc.

Exercícios:

1. (a) Processe os ficheiros acima descritos (estão dentro de ficha4.zip) utilizando o Lex e o YACC. Repare no aviso mostrado pelo YACC. Compile e teste o comportamento do programa. Efectivamente, a gramática é ambígua e os resultados não respeitam as regras de prioridade (teste por exemplo $3*2+1$)! Temos quatro regras em que não é clara a prioridade dos operadores...

O YACC permite definir prioridades com as instruções `%left` e `%right` (na parte das definições) que correspondem respectivamente a associatividade à esquerda (por exemplo $2+3+4 \rightarrow (2+3)+4$) ou à direita (por exemplo $x=y=z \rightarrow x=(y=z)$). Existe também a instrução `%nonassoc`, que significa que o operador não tem associatividade. A prioridade do operador é tanto maior quanto posterior for a sua definição. Por exemplo,

```
%right '='
```

```
%left '+'
```

indica que a soma tem associatividade à esquerda e tem maior prioridade do que a igualdade (prioridade à direita). Resolva então os problemas de ambiguidade referidos acima.

- (b) Acrescente a possibilidade de utilização de parêntesis, que permitem a alteração de prioridades pelo utilizador.
- (c) Se fizer uma divisão por zero, verá que aparece uma mensagem de erro (floating point exception). Altere o programa para passar a dar a mensagem “Divide by zero!”.
- (d) Acrescente a possibilidade de utilização do sinal ‘-’ unário, permitindo operações do género “ $2--2$ ”.
- (e) Como terá verificado, assim que introduz uma expressão, o programa sai para a shell do linux. Altere o programa para sair apenas quando se escrever a palavra “end”.
2. Faça um pequeno parser que verifica a correcção (ou incorrecção) de Lisp S- Expressions que contenham apenas operadores aritméticos e valores numéricos. Por exemplo, as expressões seguintes estão correctas:
- ```
(+ 3 2)
(+ 1.3 (/ 7 6))
(- 4 1)
(+ (+ (/ 1 3) (* 4 5) 8))
```

As seguintes estão incorrectas:

```
(3 + 2)
```

```
(+ 3 2
```

```
(+)
```

```
(4 3)
```

Imprima uma mensagem “CORRECTO” ou “INCORRECTO” conforme os resultados do seu parser.

Nota: Para detectar que o parser encontrou um erro pode fazer de duas formas:

- A função `yyparse()` devolve um valor diferente de 0
- É chamada função `yyerror(char* msg)`

## Referências

- [1] Anexo A de Processadores de Linguagens. Rui Gustavo Crespo. IST Press. 1998
- [2] A Compact Guide to Lex & Yacc. T. Niemann. <http://epaperpress.com/lexandyacc/epaperpress>
- [3] Manual do yacc em Unix (comando “man yacc” na shell)
- [4] Lex & Yacc. John R. Levine, Tony Mason and Doug Brown. O'Reilly. 2004