Ficha prática 3

Compiladores

2022/23

YACC

1 Introdução

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 diferentes[1]) 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
```

[1] Por exemplo, suponhamos que encontra número 15. O *lex* identifica o token `INTEIRO`, mas o seu valor é 15.

Na parte das definições, começamos por incluir a biblioteca `stdio.h` e declarar algumas funções e variáveis definidas pelo *lex* que serão precisas.

```
%{
  #include <stdio.h>
  int yylex(void);
  void yyerror(const char*);
  char* yytext;
%}
```

De seguida, interessa-nos 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 caracter, pode enviar simplesmente o próprio padrão ao YACC (é o caso 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). Por convenção, representam-se os símbolos terminais (tokens) em maiúsculas e os não terminais em minúsculas. Um exemplo de gramática para a calculadora simples é o seguinte:

```
%%
calc       : expression                {printf("%d", $1);}
           ;

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

Nesta gramática, temos uma definição de `calc` como sendo uma `expression`, e uma definição recursiva de `expression` que descreve operações aritméticas com um número arbitrário de operações.

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 por ordem inversa, ou seja correspondem aos i primeiros 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 interpretá-la 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;
}

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

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 enviar o 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, utiliza-se a variável `yylval`.

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

```
%{
  #include "y.tab.h"
%}
%%
```

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

```
[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
```

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

2 Exercícios

Exercício 1 *Processe os ficheiros acima descritos 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 precedência (teste por exemplo 3*2+1)!*

Temos quatro regras em que não é clara a precedência dos operadores.

O yacc permite definir precedências com as diretivas `%left` *e* `%right` *(na parte das definições) que correspondem respectivamente a associatividade à esquerda (por exemplo 2+3+4 → (2+3)+4 ou à direita (por exemplo x=y=z → x=(y=z)). Existe também a diretiva* `%nonassoc`*, que significa que o operador não tem associatividade.*

Para além disso, a ordem das diretivas especifica a precedência dos operadores.

Por exemplo,

```
%right '='
%left '+' '-'
```

indica que a soma e a subtração têm associatividade à esquerda e têm maior precedência do que a atribuição (que associa à direita).

- *Resolva então os problemas de ambiguidade referidos acima.*
- *Acrescente a possibilidade de utilização de parêntesis, que permitem ao utilizador alterar a ordem das operações.*
- *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!`.
- *Acrescente a possibilidade de utilização do sinal* `-` *unário, permitindo operações do tipo 2 - -2.*

□

Exercício 2 *Faça um pequeno analisador sintático que verifique a correcção (ou incorrecção) de Lisp S-Expressions que contenham apenas operadores aritméticos e valores numéricos.*[3]

Por exemplo, as expressões seguintes estão correctas:

```
(+ 3 2)
(+ 1.3 (/ 7 6))
(- 4)
(+ (+ (/ 1 3) (* 4 5)) 8)
```

As seguintes estão incorrectas:

```
(3 + 2)
(+ 3 2
(-)
(4 3)
```

O analisador deve imprimir uma mensagem, `CORRECTO` *ou* `INCORRECTO`*, conforme o caso.*

Note que pode detectar que o parser encontrou erro de duas formas:

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

□

[3]Ver por exemplo `http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node125.html` para mais informação sobre expressões deste tipo.

Referências

[1] Anexo A de Processadores de Linguagens. Rui Gustavo Crespo. IST Press. 1998

[2] A Compact Guide to Lex & Yacc. T. Niemann.
`https://www.epaperpress.com/lexandyacc/`

[3] Manual do lex/flex em Unix (comando `man lex` na shell)

[4] Lex & Yacc. John R. Levine, Tony Mason and Doug Brown. O'Reilly. 2004