Ficha prática 4

Compiladores

2018/2019

YACC (continuação)

1 Introdução

Na aula anterior, construiu-se uma pequena calculadora, transferindo do *lex* para o *yacc* os valores identificados como sendo numéricos (sempre que não eram números, transferia simplesmente o caracter encontrado, p.e., um sinal `+`).

Em linguagens de programação, mais do que a transmissão de valores, é por vezes importante a transmissão de símbolos, normalmente identificadores de variáveis (ou funções). Assim, quando o *lex* encontra uma variável, poderá enviar ao *yacc* o respectivo identificador.

Lembre-se também que é a variável `yylval` que contém tipicamente o valor do token encontrado num dado momento. Assim, a um token `NUMBER` pode corresponder um valor numérico, contido na variável `yylval`. Por exemplo, quando identifica a string `"15"`, o *lex* poderá enviar ao *yacc* o token `NUMBER` (com `return NUMBER`), guardando o valor 15 em yylval (com `yylval=atoi(yytext)`).

Temos então, na variável `yylval`, uma espécie de stream de transmissão de valores entre o *lex* e o *yacc*. Claro que, para enviar mais do que inteiros, teremos que permitir que `yylval` receba valores de vários tipos.

Na linguagem C, existe uma estrutura apropriada a este efeito, a `union`. Uma `union` permite partilhar, no mesmo espaço de memória, vários tipos diferentes. Por exemplo:

```
typedef union{
    char c;
    int i;
    char string[10];
    double d;
} exemplo;
```

A `union` anterior representa um `char`, um `int`, uma `string` ou um `double`. Note que uma variável do tipo `exemplo` só poderá ser de um tipo de cada vez (ou é `char`, `int`, `string` ou `double`).

Para permitir que `yylval` contenha vários tipos de valores, deverá então incluir o seguinte código na especificação *yacc*:

```
%union{
/* vários tipos */
}
```

De seguida, para associar o tipo de valor a cada token deverá também alterar as declarações de tokens na seguinte sintaxe:

```
%token <tipo de token> TOKEN
```

Também pode associar um tipo a símbolos não terminais, com a seguinte expressão:

```
%type <tipo> simboloNaoTerminal
```

É normalmente importante fazê-lo, pois é comum os símbolos não terminais representarem resultados intermédios que é importante transmitir para outras regras.

Um exemplo é o símbolo `expression` da ficha anterior: cada expressão contém o valor que resulta do seu cálculo aritmético associado, por isso `expression` era também `int`. Abaixo vemos uma versão muito reduzida desta ideia.

Com estas associações de tipos, o *yacc* irá associar os valores correctos de `yylval` às referências de pilha. Por exemplo:

```
%union{
char cval;
int intval;
}
%token <cval> CARACTER
%token <intval> NUMBER
%type <intval> expression
%%
statement:  expression                {printf("%d", $1);}
;
expression: NUMBER                    {$$=$1;}
|           CARACTER                  {$$=(int)$1;}
|           expression '+' expression {$$=$1+$3;}
;
```

Nesta pequena gramática, quando se detectar um `NUMBER`, o *yacc* assumirá o valor `yylval.intval` para `$1`. Quando detectar um `CARACTER`, assumirá o valor `yylval.cval`.

Para se perceber quando é necessário declarar o tipo de um símbolo não terminal é simples. Se nas suas regras (aquelas em que esse símbolo aparece no lado esquerdo) houver acções que referenciem o topo da pilha (`$$`), isso implica que é obrigatório definir o tipo (com uma declaração `%type ...`).

Para esta ficha, iremos abordar um pequeno interpretador (não é um compilador!) em que se poderá acrescentar símbolos novos (variáveis) a uma tabela de símbolos. Este interpretador aceita apenas dois comandos:

```
    VARIAVEL=VALOR    (guarda valor inteiro na variavel)
    VARIAVEL          (apresenta valor da variavel)
```

Uma interacção possível seria:

```
[utilizador] a=9
[utilizador] c=897
[utilizador] a
[programa  ] o valor da variavel a é 9.
[utilizador] c
[programa  ] o valor da variavel c é 897.
```

Para poder guardar variáveis, deveremos utilizar uma tabela de símbolos. Esta tabela conterá então estruturas com o nome e valor de cada símbolo. Abaixo, damos um exemplo da estrutura `symtab`:

```
typedef struct _symtab{
    char *name;
    int value;
} symtab;
```

Vamos assumir que esta declaração passará a estar no ficheiro `symtab.h` e que a tabela de símbolos se chamará `ubc` e que conterá um máximo de 100 símbolos. Precisamos também de permitir que `yylval` transmita valores (inteiros) e identificadores de variáveis (strings). Para isso, na secção de definições da especificação *yacc*, colocamos o código correspondente às `union` e `tokens` necessárias. Abaixo, mostramos o ficheiro `ficha4.exemplo.y`, já contendo as regras gramaticais necessárias ao nosso pequeno programa:

```
%{
#include <stdio.h>
#include "symtab.h"
#define NSYMS 100
```

```
symtab tab[NSYMS];
symtab *symlook(char *varname);
%}
%token <id> VAR
%token <value> NUMBER
%union{
int value;
char* id;
}
%%
statement: expression '\n'
|          statement expression '\n'
;
expression:
           VAR '=' NUMBER {symlook($1)->value = $3;}
|          VAR            {printf("o valor da variável %s é %d\n",
                                   symlook($1)->name, symlook($1)->value);}
;
%%
...
```

É claro que, de cada vez que encontrarmos uma variável, teremos que obter o seu ponteiro na tabela (ou acrescentar uma nova entrada). Isso será feito pela função `symlook`, que recebe o nome da variável. Se ela existir na tabela, devolve o seu ponteiro, caso contrário, cria uma nova entrada (devolvendo também o seu ponteiro).

Repare em cima nas duas chamadas a `symlook` (na primeira para criar/guardar o valor na variável; na segunda para imprimir o valor da variável). Observe também atentamente um excerto do ficheiro `ficha4.exemplo.l` abaixo.

```
%{
#include "y.tab.h"
%}
%%
[0-9]+              {yylval.value=atoi(yytext);
                     return NUMBER;}
[a-zA-Z][A-z0-9 ]*  {yylval.id=(char*)strdup(yytext);
                     return VAR;}
[ \t]               ;
\n | .              return yytext[0];
%%
...
```

2 Exercícios

Exercício 1 *Altere o programa que realizou na aula anterior de forma a incluir:*

- *Utilização de variáveis, na forma descrita nesta ficha.*

  Isto é, a criação e inclusão na tabela de símbolos de uma variável acontece na sua primeira utilização (e.g. `a=9`*), podendo depois ser utilizada nos cálculos (e.g.* `c=a+1`*). Assuma que uma variável tem que ser começada por uma letra, podendo depois conter números, letras ou underscore;*

- *Utilização de valores* `double` *nas expressões (não necessariamente nas variáveis, que se poderão manter inteiras se assim pretender);*

- *Possibilidade de utilizar os operadores* `++` *e* `--` *(como em C ou em Java, e.g.,* `x++` $\Leftrightarrow$ `x=x+1`*);*

- *Um comando* `varlist`*, que apresenta os valores de todas as variáveis, e um comando* `save`*, que devera escrever os valores das variáveis no disco, no seguinte formato:*

```
a    0
b    55
...  ...
```

- *Utilização das funções* `sqrt`*,* `exp`*,* `log` *e do operador* `^` *(de exponenciação).*

Teste o seu programa com o ficheiro `test4.1.txt`*. Deverá obter um resultado semelhante ao contido no ficheiro* `result4.1.txt`*.*

□

Exercício 2 *Considere uma linguagem de programação simplificada, de cuja seguinte sintaxe se apresenta o seguinte exemplo:*

```
let
    integer  n
    double   x
    char     z
in
    write n
    write x
end
```

- *Analise o seguinte excerto de uma possível especificação yacc para processar programas na linguagem acima.*

  Note que esta especificação é dada no material de apoio no ficheiro `ficha4.2.y`*, bem como a correspondente especificação lex, no ficheiro* `ficha4.2.l`*, bem como todos os ficheiros auxiliares de que estas dependem. Todos devem ser analisados com todo o cuidado.*

```
...
program: LET vardeclist IN statementlist END
                                {$$=myprogram=insert_program($2, $4);}
    ;

vardeclist: /*empty*/           {$$=NULL;}
    | vardeclist vardec         {$$=...}
    ;

vardec: INTEGER IDENTIFIER      {$$=...}
    | CHARACTER IDENTIFIER      {$$=...}
    | DOUBLE IDENTIFIER         {$$=...}
    ;

statementlist: /*empty*/        {$$=...}
    | statementlist statement   {$$=...}
    ;

statement: WRITE IDENTIFIER     {$$=...}
    ;
    ...
```

- *Complete os espaços marcados com* `functions.c`*, de modo a que, utilizando as funções de construção da árvore de sintaxe abstrata (AST) disponibilizadas em* `functions.c`*, seja possível construir uma AST para cada programa na linguagem de programação simplista.*

- *Implemente uma função que percorra a AST e imprima o seu conteúdo. A ideia é utilizar esta função depois da invocação de* `yyparse()` *para confirmar que a AST construída é a correcta.*

□

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