Licenciatura em Engenharia Informática – DEI/ISEP
Linguagens de Programação 2006/07

Ficha 6

Bison

Objectivos:

- Introdução aos analisadores sintácticos
- Introdução à ferramenta *BISON*;
- Comunicação entre *BISON* e *FLEX*;
- Aprendizagem dos conceitos através da realização de alguns exercícios;

6.1 Analisadores sintácticos

O *BISON* é um gerador de analisadores sintácticos de âmbito genérico, que converte uma gramática independente do contexto LALR(1)[1] num programa C capaz de processar frases da linguagem reconhecida por essa gramática. O *BISON* pode ser integrado com o *FLEX* para o reconhecimento dos *tokens* da linguagem.

Para o *BISON* fazer a análise sintáctica de uma linguagem, é necessário que essa linguagem esteja descrita através de uma gramática independente do contexto. Nem todas as gramáticas independentes do contexto podem ser processadas pelo *BISON*, pois é também necessário que elas sejam LALR(1). Isto significa que tem de ser possível especificar como processar uma parte da frase a analisar, simplesmente com um *token* de avanço. Note-se que todas as gramáticas LALR(1), são obrigatoriamente LR(1), mas não o inverso. Na secção 6.6 são explicados os tipos de conflitos que podem ocorrer numa gramática *BISON*.

Numa linguagem formal, as regras gramaticais para a linguagem, são representadas por um símbolo. Existem dois tipos de símbolos:

não terminais ou regras – são construídos através do agrupamento de outros símbolos, terminais ou não terminais;

[1] Look-Ahead Left to right Rightmost derivation

terminais ou *tokens* – são símbolos que não podem ser divididos e são identificados pelo analisador léxico.

Considere a função C apresentada no excerto seguinte:

```
1 int square(int x)
2 {
3   return x * x;
4 }
```

Um analisador léxico (por exemplo o *FLEX*) é capaz de identificar os *tokens* que constituem esta frase. Assim após realizada a análise léxica, seria obtido o seguinte conjunto de *tokens*.

```
TIPO_INT ID '(' TIPO_INT ID ')'
'{'
  RETURN ID '*' ID ';'
'}'
```

A especificação da gramática a reconhecer pelo *BISON* é feita utilizando a notação BNF (ver secção 5.2.2), sendo os símbolos não terminais (regras) escritos em minúsculas e os símbolos terminais (*tokens*) escritos em maiúsculas. Quando um *token* é composto simplesmente por um carácter este é representado pelo próprio carácter delimitado por plicas. De seguida é apresentado um extracto de uma gramática *BISON* para o reconhecimento da função apresentada na listagem anterior:

```
1  funcao:   tipo ID '(' lista_parametros ')'
2            bloco_de_instrucoes
3            ;
4  bloco_de_instrucoes: '{' instrucoes '}'
5                     ;
6  instrucoes: /* vazio */
7            | instrucoes instrucao
8            ;
9  instrucao:    RETURN expressao ';'
10               ;
11 tipo:         TIPO_INT | TIPO_FLOAT | TIPO_CHAR
12               ;
13 expressao:    operando resto_expressao
14               ;
15 resto_expressao:    /* vazio */
16                   | operador expressao
17                   ;
18 operando: INTEIRO | REAL | ID
19           ;
20 operador: '+' | '-' | '*' | '/'
```

```
21              ;
22 lista_parametros: /* vazio */
23                   | lista_parametros ',' parametro
24                   ;
25 parametro: tipo ID
26          ;
```

6.2 Modo de utilização do BISON

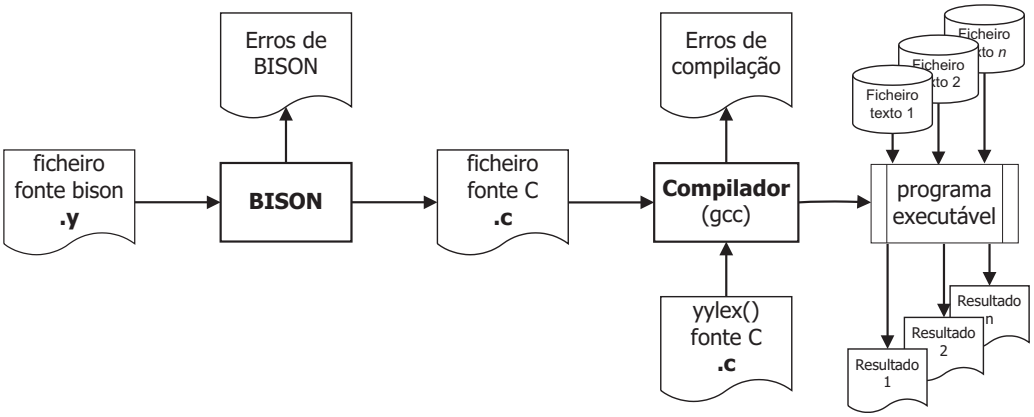

Figura 6.1: Ciclo de vida de um programa BISON

Com base num ficheiro fonte escrito de acordo com a sintaxe do *BISON* o programa `bison` gerará um analisador sintáctico descrito na linguagem C e um *header* file com a definição dos *tokens* usados na gramática. No caso de existirem erros de codificação, o *BISON* gerará uma listagem de erros. De seguida é necessário criar o analisador léxico, pois o *BISON* sempre que necessita de um novo *token* chama a função `yylex()`, esta função pode ser criada pelo programador ou gerada usando o *FLEX*. No final os ficheiros fonte em C criados com o analisador léxico e o analisador sintáctico terão de ser compilados para a plataforma destino, utilizando um compilador da linguagem C adequado (neste exemplo o GCC). O resultado final da compilação será um programa executável capaz de identificar frases que respeitem a gramática definida e executar as acções semânticas respectivas. Como entrada para o analisador sintáctico gerado podem ser fornecidos ficheiros de texto, ou alternativamente, fornecer os dados directamente pelo *standard* de entrada.

No exemplo seguinte são apresentados os passos necessários à compilação e utilização de um programa *FLEX*/*BISON*. Considere-se a existência do ficheiro `ficheiro.y` com o programa *BISON* já escrito e do ficheiro `ficheiro.flex` com o programa *FLEX*.

```
bison -d ficheiro.y
flex ficheiro.flex
gcc ficheiro.tab.c lex.yy.c -lfl
./a.out
```

O comando `bison` gera, por omissão, um ficheiro com o nome `ficheiro.tab.c` e a opção "`-d`" indica que deve ser criado o ficheiro *header* chamado `ficheiro.tab.h`. De seguida é chamado o `flex` para criar o analisador léxico, por omissão será gerado o ficheiro `lex.yy.c`. Os dois ficheiros com extensão "`.c`" deverão ser compilados, usando o `gcc`. Na utilização do `gcc` é necessário indicar a utilização da biblioteca *FLEX* adicionando o parâmetro "`-lfl`". Por sua vez, o compilador de C gera, por omissão, um ficheiro com o nome `a.out`. Por último, para a execução deste programa basta a invocação do seu nome na linha de comandos. Neste caso, a introdução dos dados terá de ser realizada via consola (terminando obrigatoriamente com `Ctrl+D`).

```
bison -d Exemplo.y
flex -oExemplo.c Exemplo.flex
gcc Exemplo.tab.c Exemplo.c -o Programa -lfl
./Programa < Dados.txt
```

Neste exemplo, o comando `bison` gera a partir do ficheiro `Exemplo.y`, o ficheiros com o nome `Exemplo.tab.c` e `Exemplo.tab.h` e o comando `flex` gera o ficheiro `Exemplo.c`. Os ficheiros com a extensão ".c" deverão de seguida ser compilados. Nesta utilização apresentada do `gcc`, é indicado o nome do executável a ser gerado, neste caso, `Programa`. Na execução do `Programa`, a introdução dos dados é realizada a partir do ficheiro `Dados.txt`.

6.3 Formato de um ficheiro BISON

Um programa em *BISON* é constituído por três secções, a saber, declarações, gramática e rotinas auxiliares. A separação entre as secções é feita através da inserção uma linha com o símbolo "`%%`".

Considere-se o seguinte exemplo que será discutido com maior detalhe nas secções seguintes.

```
1 %{
2   #include <stdio.h>
3   int numArgs=0, numErros=0;
4 %}
5
6 %token ID INT REAL
7 %start inicio
```

```
8
9 %%
10 inicio:        /* vazio */
11               | lista_args
12               ;
13 lista_args:    arg
14               | lista_args ',' arg
15               ;
16 arg:           ID          {numArgs++;}
17               | INT        {numArgs++;}
18               | REAL       {numArgs++;}
19               ;
20 %%
21
22 int main(){
23   yyparse();
24   if(numErros==0)
25     printf("Frase válida\n");
26   else
27     printf("Frase inválida\nNúmero de erros: %d\n",numErros
         );
28   printf("Número de argumentos é %d\n",numArgs);
29   return 0;
30 }
31 int yyerror(char *s){
32   numErros++;
33   printf("erro sintactico/semantico: %s\n",s);
34 }
```

6.3.1 Declarações

Esta secção compreende duas partes:

- **instruções C** – nesta parte, delimitada pelos símbolos "`%{`" e "`%}`", são colocadas as instruções da linguagem C que posteriormente serão incluídas no início do ficheiro C a gerar pelo *BISON*. Os exemplos mais comuns são a inclusão de ficheiros de cabeçalhos (*headers*, `.h`), bem como a declaração de variáveis e constantes.

```
1 /* Definição das variáveis numArgs e numErros */
2 %{
3   #include <stdio.h>
4   int numArgs=0, numErros=0;
5 %}
```

- **declarações bison** – nesta parte, são realizadas as definições *BISON* que incluem, entre outros, a declaração de terminais, não terminais, precedência de operadores e a regra inicial.

```
/* Definições do bison */
%token ID INT REAL
%start inicio
```

Definição de tipos para comunicação FLEX/BISON

O analisador léxico ao identificar alguns *tokens* tais como valores inteiros, valores reais, identificadores e cadeias de caracteres, necessita indicar ao *BISON* não só o tipo de *token*, mas também o seu valor (lexema). Para isso é usada a variável `yylval`, que é definida no *BISON* por omissão como um inteiro. Assim o analisador léxico ao encontrar um inteiro colocará o seu valor na variável `yylval`, e devolverá `INT`, tal como é apresentado no extracto seguinte de código *FLEX*.

```
[-+]?[0-9]+          yylval=atoi(yytext);return INT;
```

É possível alterar o tipo da variável `yylval` através da directiva seguinte, continuando-se limitado a um só tipo de dados.

```
#define YYSTYPE double
```

Frequentemente é necessário passar diferentes tipos de dados entre o analisador léxico e o *BISON*, assim, a alternativa a ter só um tipo de dados, é a utilização da directiva *BISON* `%union`, que permite definir uma estrutura com vários campos que depois podem ser atribuídos a *tokens* e regras, através da variável `yylval`. De seguida, é apresentado um exemplo da definição de uma `%union`. Mais adiante será explicado como atribuir os campos aos *tokens* e regras.

```
%union {
    char  *id;
    int   inteiro;
    float real;
}
```

Declaração de *tokens* e dos tipos para os *tokens* e regras

Na declaração dos *tokens* pode ser atribuído um dos campos da estrutura definida com a declaração `%union`. No extracto de código seguinte é apresentada a declaração de *tokens* e regras para os três campos da estrutura definida com a directiva "`%union`".

```
%token <id>      ID STRING
%token <inteiro> INT
%token <real>    REAL
%type  <real>    operando expressao
```

De notar que em cada declaração podem ser definidos vários *tokens* ou regras ao mesmo tempo, desde que partilhem o mesmo tipo. No exemplo anterior, esta situação verifica-se com os *tokens* "`ID`" e "`STRING`" e com as regras "`operando`" e "`expressao`".

No analisador léxico seriam guardados os valores dos *tokens*, no respextivo campo da variável `yylval`, tal como apresentado no seguinte extracto de código *FLEX*.

```
1 [-+]?[0-9]+          yylval.inteiro=atoi(yytext);return INT;
2 \"[^"\n]*\"          yylval.id=yytext;return STRING;
3 [_a-zA-Z][_a-zA-Z0-9]*   yylval.id=yytext;return ID;
```

Precedência de operadores

A precedência de operadores no *BISON* serve para definir a ordem pela qual as regras alternativas são processadas, eliminando assim os conflitos que possam surgir. Considere o exemplo seguinte.

```
1 %left     OPL
2 %right    OPD
3 %nonassoc OPNA
```

%left – Declara um operador binário com associação à esquerda, isto é o agrupamento é realizado primeiro à esquerda. Assim a frase "`x OPL y OPL z`" seria realizada primeiro a operação "`x OPL y`" e depois o resultado desta operação com "`OPL z`";

%right – Declara um operador binário com associação à direita, isto é o agrupamento é realizado primeiro à direita. Assim a frase "`x OPD y OPD z`" seria realizada primeiro a operação "`y OPD z`" e depois é realizada o operação "`x OPD`" com o resultado obtido anteriormente;

%nonassoc – Declara um operador não associativo, ou seja, um operador que não pode aparecer mais que uma vez de seguida. Assim a análise da frase "`x OPNA y OPNA z`" gera um erro.

A precedência relativa entre vários operadores, é controlada pela ordem pela qual aparecem as declarações. Isto é, quanto menor for a linha na qual aparece a declaração, menor é a precedência do operador. É possível declarar vários operadores na mesma linha tal como é apresentado no seguinte código.

```
1 %left  '<' '>' '=' DIF MEN_IG MAI_IG
2 %left  '+' '-'
3 %left  '*' '/'
4 %left  '^'
5 %nonassoc MENOS_UNARIO
```

Neste exemplo os operadores relacionais têm a menor precedência e o menos unário tem a maior precedência. A declaração de operadores é uma alternativa à declaração de *tokens*, assim não é necessário declarar os *tokens* `DIF`, `MEN_IG` e `MAI_IG` com a primitiva `%token`. O *token* `MENOS_UNARIO` é aqui declarado, mas nunca será devolvido pelo analisador léxico. Este *token* só será utilizado para atribuir a sua precedência a uma regra da gramática, usando a declaração `%prec MENOS_UNARIO`. Assim numa frase que use a gramática apresentada de seguida, será realizadas em primeiro lugar a operação "negativo" (menos unário), depois todas as multiplicações e divisões, finalmente todas as adições e subtracções.

```
1  expressao: expressao '+' expressao
2           | expressao '-' expressao
3           | expressao '*' expressao
4           | expressao '/' expressao
5           | '-' expressao %prec MENOS_UNARIO
6           | operando
7           ;
8  operando: INTEIRO
9           | REAL
10          | ID
11          ;
```

Regra inicial

A regra inicial no *BISON* é por omissão a primeira regra, no entanto existe a directiva `%start` que permite alterar este comportamento. No exemplo seguinte é indicado que a regra inicial é a regra inicio, independentemente de ser a primeira ou não.

```
1 %start inicio
```

6.3.2 Gramática

Nesta secção é definida a gramática propriamente dita e as acções semânticas a ela associadas. As regras da gramática são definidas de acordo com a notação BNF (ver secção 5.2.2), sendo a regra inicial por omissão, a primeira descrita, ou no caso de existência da declaração `%start`, a aí especificada. As regras são declaradas em minúsculas, e o símbolo "definido como" é o símbolo "`:`". Os *tokens*, se tiverem mais de um carácter, são declarados em maiúsculas, ou, se tiverem somente um carácter, são especificados entre plicas. Para a representação de regras alternativas é utilizado o símbolo "`|`", sendo a alternativa vazia, representada por uma alternativa sem conteúdo (por uma questão de legibilidade é normalmente acrescentado o comentário "`/* vazio */`"). Todas as regras terminam com o símbolo "`;`".

```
1  inicio:       /* vazio */
2              | lista_args
3              ;
4  lista_args: arg
5              | lista_args ',' arg
6              ;
7  arg:        ID          {numArgs++;}
8              | INT       {numArgs++;}
9              | REAL      {numArgs++;}
10             ;
```

Apesar do *BISON* aceitar recursividade à esquerda ou à direita, deve ser sempre usada a recursividade à esquerda, pois com a recursividade à direita, o *BISON* tem que processar todos os *tokens* (inseri-los na pilha), antes de começar a aplicar as regras. Por este motivo a recursividade à direita pode levar o analisador a gerar um erro de "*stack overflow*".

Existe uma regra especial "`error`" que instancia com qualquer coisa, e destina-se a fazer a recuperação de erros. Na regra lista_args, poder-se-iam acrescentar várias alternativas contendo a regra *error* para recuperar dos erros mais comuns. Algumas técnicas de recuperação de erros são explicadas mais adiante, na secção 6.7.1.

Sempre que o *BISON* não consegue fazer *match* em nenhuma regra com os *tokens* recebidos, invoca automaticamente a função `yyerror()` que deve ser fornecida pelo programador.

Acções semânticas

Ao longo da gramática são inseridas as acções semânticas nas regras. Essas acções semânticas são instruções C entre chavetas acrescentadas às várias alternativas da regra. Apesar de normalmente as acções aparecerem somente no final da regra, estas podem ser colocadas em qualquer parte da regra, como se pode ver no exemplo seguinte.

```
1  expressao:      {printf("antes da expressao");}
2              INT {printf("antes do operador");}
3              '+' {printf("depois do operador");}
4              INT {printf("depois da expressao");}
5              ;
```

Deve-se ter cuidado com as acções situadas antes do final das regras, pois numa regra com várias alternativas, o *BISON* é obrigado a decidir qual delas deve utilizar logo no início da regra. Se duas alternativas duma regra começarem pelo mesmo *token*, e existir uma acção semântica no início da regra, o *BISON* apesar de não saber que alternativa utilizar, tem que escolher se executa a acção semântica ou não. Este problema resolve-se tornando as acções semânticas intermédias, em

acções semânticas finais, substituindo-as por uma nova regra. Este processo é apresentado no próximo exemplo de código.

```
1  expressao: ac1 ac2 ac3 ac4;
2             ;
3  ac1:       /* vazio */ {printf("antes da expressao");}
4             ;
5  ac2:       INT {printf("antes do operador");}
6             ;
7  ac3:       '+' {printf("depois do operador");}
8             ;
9  ac4:       INT {printf("depois da expressao");}
10            ;
```

O analisador léxico, deve colocar o valor dos *tokens* (lexemas) na variável `yylval`, valor esse que é inserido na pilha aquando da inserção do *token*. Este valor pode depois ser acedido através das variáveis especiais \$1, \$2 ... \$$n$, na qual o n representa a ordem pela qual o elemento aparece na regra. Existe ainda a variável \$\$, que representa o valor que a própria regra terá (lado esquerdo da regra, equivalente a um *return* no C). Existe também a estrutura @n, que tem os campos `first_line`, `last_line`, `first_column` e `last_column`, contendo informação sobre a localização dos *tokens* e regras no documento original.

A utilização de acções semânticas no meio de regras, ocupa um "\$" por cada acção. No exemplo apresentado em cima, o primeiro *token* `INT` é o "\$2", o *token* '+' é o "\$4" e o segundo *token* `INT` é o "\$6". De notar que nas acções que se situam no meio da regra, só é possível aceder aos valores semânticos anteriores, pois o *BISON* ainda não tem valores definidos para os componentes da regra, que se situa após a acção semântica

No exemplo seguinte o regra "`expressao`" será o adição do valor semântico das duas regras "`operando`". De notar que o *token* '+', apesar de normalmente não ter valor semântico, é referenciado por \$2.

```
1  expressao: operando '+' operando {$$=$1+$3;}
2             ;
3  operando:   INTEIRO    {$$=$1;}
4            | REAL       {$$=$1;}
5            ;
```

6.3.3 Rotinas em C de suporte

Nesta secção pode ser escrito o código C que se pretende que seja adicionado ao programa a gerar pelo *BISON*. Tipicamente este código inclui o corpo do programa, designadamente, a função `main()` da linguagem C e a função `yyerror()`, usada pelo *BISON* sempre que detecta um erro sintáctico. De notar que quando é usado o *FLEX* como *scanner* só existirá função `main()` no código *BISON*, pois

o *BISON* encarregar-se-á de invocar a função `yylex()` criada pelo *FLEX* sempre que necessitar de um novo *token*.

```
1  int main(){
2    yyparse();
3    if(numErros==0)
4      printf("Frase válida\n");
5    else
6      printf("Frase inválida\nNúmero de erros: %d\n",numErros
           );
7    printf("Número de argumentos é %d\n",numArgs);
8    return 0;
9  }
10 int yyerror(char *s){
11   numErros++;
12   printf("erro sintactico/semantico: %s\n",s);
13 }
```

A função `yyparse()` é o analisador sintáctico gerado pelo *BISON* que processará a gramática anteriormente descrita (ver secção 6.3.2), devolvendo um valor diferente de zero se encontra um erro do qual não consegue recuperar ou zero se termina com sucesso. É ainda possível saber quantos erros existem na gramática através da variável global `yynerrs`.

6.4 Comunicação com o analisador léxico

É necessário criar um analisador léxico (*scanner*) que forneça os *tokens* ao *BISON*, esse *scanner* pode ser produzido no *FLEX*, sendo de seguida apresentado um exemplo do analisador léxico para a gramática apresentada na secção 6.3.

```
1  %{
2    #include"exemplo.tab.h" /* header gerado pelo bison */
3    extern int numErros;     /* vaiável criado no bison */
4  %}
5
6  %%
7
8  ,                          return yytext[0];
9  [0-9]+                     return INT;
10 [0-9]+\.[0-9]+             return REAL;
11 [_a-zA-Z][_a-zA-Z0-9]*     return ID;
12 [ \t]                      /* ignorado */
13
14 .              {
```

```
15                  printf("Erro lexico: simbolo desconhecido %s\n
                        ",yytext);
16                  numErros++;
17                 }
18 \n                              return 0;
19 <<EOF>>                        return 0;
20
21 %%
```

O analisador léxico deve ignorar todos os espaços em branco e apresentar um erro léxico para todos os símbolos que não sejam identificados como *tokens*. No exemplo anterior são identificados como fim de frase o "`\n`" e o fim de ficheiro, sendo indicado ao *BISON* devolvendo o valor 0. Normalmente o "\n" é somente usado para contar linhas, podendo em algumas gramáticas, ser passado ao *BISON* como um *token* normal. Sempre que o *token* é simplesmente um carácter, é devolvido o seu valor, guardado em `yytext[0]`, deste modo podemos inserir todos os *tokens* de um único carácter como alternativas duma expressão regular, por exemplo "`,|;|:|...  return yytext[0];`". Para todos os outros *tokens*, é devolvida a constante definida no *BISON* através das directivas `%token`.

6.5 Exemplo mais elaborado

Este exemplo e outros desta ficha, encontram-se em, `http://www.dei.isep.ipp.pt/~matos/lprog/ex_ficha6.zip`. Neste arquivo, encontra-se o *bash script* "`fl`", que recebe como parâmetro um nome (sem extensão) e gera o analisador sintáctico se existir o ficheiro "`nome.y`", gera o analisador léxico se existir o ficheiro "`nome.lex`" e em caso de sucesso, compila os ficheiros "`.c`".

6.5.1 Ficheiro *FLEX*

```
1 %{
2   #include"exemplo.tab.h" // header greado pelo bison
3   extern int numErros;
4 %}
5
6
7 %%
8
9 ,                          return yytext[0];
10 [0-9]+                    yylval.inteiro=atoi(yytext); return INT;
11 [0-9]+\.[0-9]+            yylval.real=atof(yytext); return REAL;
12 [_a-zA-Z][_a-zA-Z0-9]*    yylval.id=yytext; return ID;
13 [ \t]                     /* ignorado */
14
```

```
15 .                        printf("Erro lexico: simbolo
       desconhecido %s\n",yytext); numErros++;
16
17 \n                        return 0;
18 <<EOF>>                   return 0;
19
20 %%
```

6.5.2 Ficheiro *BISON*

```
1  %{
2      #include <stdio.h>
3      int numArgs=0, numErros=0;
4  %}
5
6  %union {
7    char  *id;
8    int   inteiro;
9    float real;
10 }
11 %token <id>      ID
12 %token <inteiro> INT
13 %token <real>    REAL
14 %start inicio
15
16 %%
17 inicio:       /* vazio */
18              | lista_args
19              ;
20 lista_args: arg
21              | lista_args ',' arg
22              | lista_args ',' error {yyerror("falta argumento");}
23              | lista_args {yyerror("falta virgula");} arg
24              ;
25 arg:          ID          {numArgs++;printf("ID:%s\n",$1);}
26              | INT         {numArgs++;printf("INT:%d\n",$1);}
27              | REAL        {numArgs++;printf("REAL:%f\n",$1);}
28              ;
29 %%
30
31 int main(){
32
33   yyparse();
34
35   if(numErros==0)
36     printf("Frase válida\n");
37   else
38     printf("Frase inválida\nNúmero de erros: %d\n",numErros);
39   printf("Número de argumentos é %d\n",numArgs);
```

```
40   return 0;
41 }
42
43 int yyerror(char *s){
44   numErros++;
45   printf("erro sintatico/semantico: %s\n",s);
46 }
```

6.6 Conflitos na gramática BISON

O funcionamento do *BISON* baseia-se num autómato de pilha, onde são inseridos os *tokens* e os lexemas. Sempre que uma sequência de *tokens* faz *match* com a regra actual, estes *tokens* são substituídos pela regra. A inserção de *tokens* na pilha é chamada de `shift` e a substituição dos *tokens* por regras é chamada de `reduce`.

A melhor maneira de verificar o funcionamento do analisador sintáctico gerado pelo *BISON*, é usando a opção "`-v`", ou incluindo no ficheiro "`.y`" a directiva `%verbose`. Esta directiva indica ao *BISON* para criar um ficheiro com a extensão "`.output`", contendo a informação sobre a gramática, os conflitos, os terminais, os não terminais e os estados do autómato gerado pelo *BISON*.

6.6.1 Conflito reduce/reduce

Sempre que o *BISON* pode fazer `reduce` de duas ou mais regras simultâneamente, diz-se que há um conflito `reduce`/`reduce`. Este erro surge normalmente da ambiguidade da gramática, e deve ser sempre corrigido.

```
1  sequencia: /* vazio */
2               { printf ("palavra vazio\n"); }
3           | talvez_palavra
4           | sequencia palavra
5                   { printf ("adicionada palavra %s\n", $2); }
6           ;
7  talvez_palavra: /* vazio */
8                   { printf ("talvez_palavra vazio\n"); }
9           | palavra
10                  { printf ("palavra simples %s\n", $1); }
11          ;
12 palavra: TOKEN
13          ;
```

Na listagem anterior há ambiguidade pois a frase vazia pode ser obtida simplesmente com opção "`vazio`" da regra "`sequencia`", ou usando a alternativa "`talvez_palavra`" que por sua vez é substituída por "`vazio`". Também a frase com uma "`palavra`", pode ser derivada como como se vê no exemplo seguinte:

<**sequencia**> ⇒ <**sequencia**><**palavra**> ⇒ </***vazio***/><**palavra**> ⇒ *TOKEN*
<**sequencia**> ⇒ <**talvez_palavra**> ⇒ <**palavra**> ⇒ *TOKEN*

6.6.2 Conflito shift/reduce

Este tipo de conflito acontece sempre que o *BISON* pode fazer o `reduce` de vários *tokens* ou inserir um novo *token* na pilha. No exemplo seguinte, quando o *BISON* encontra o *token* `ELSE`, pode fazer o `reduce` a partir da primeira alternativa, ou fazer o `shift` do *token* usando a segunda alternativa. Neste tipo de conflito, o *BISON* usa sempre o `shift`.

```
1 if_stmt:   IF expr THEN stmt
2          | IF expr THEN stmt ELSE stmt
3          ;
4 stmt:    TOKEN
5        | if_stmt
6        ;
7 expr:    TRUE
8        | FALSE
9        ;
```

<**if_stmt**> ⇒ *IF*<**expr**>*THEN*<**stmt**> ⇒
⇒ *IF*<**expr**>*THEN IF*<**expr**>*THEN*<**stmt**>*ELSE*<**stmt**>
<**if_stmt**> ⇒ *IF*<**expr**>*THEN*<**stmt**>*ELSE*<**stmt**> ⇒
⇒ *IF*<**expr**>*THEN IF*<**expr**>*THEN*<**stmt**>*ELSE*<**stmt**>

Num exemplo como este, é norma ligar o `else` ao `if` mais interior, que é obtido pelo *BISON*, realizando o `shift` em vez do `reduce`.

6.7 Tópicos avançados

6.7.1 Recuperação de erros

A recuperação de erros permite ao *BISON* continuar a análise mesmo quando encontra um erro de sintaxe. Há duas maneiras mais comuns de recuperar de erros no *BISON*. A primeira consiste na construção de alternativas às regras com os erros

mais comuns, por exemplo a falta de uma virgula, de um ponto e virgula ou de fechar um parêntesis. A segunda consiste na criação de alternativas que incluam a regra especial "`error`" que instancia com qualquer coisa. Na regra "`lista_args`", podem-se acrescentar várias alternativas contendo, ou não, a regra "`error`" para recuperar dos erros mais comuns.

```
1 lista_args: arg
2         | lista_args ',' arg
3         | lista_args     arg   {yyerror("falta virgula");}
4         | lista_args ',' error {yyerror("falta argumento");}
5         | error {yyerror("erro grave, sem recuperação");}
6         ;
```

6.7.2 Geração de um *parser*/*scanner* numa classe

Se os nomes dos ficheiros *FLEX* e *BISON* terminar em "`+`", os analisador léxico e o analisador sintáctico serão classes C++. Nos ficheiros de ajuda do *FLEX* e do *BISON* são incluídas secções especiais sobre este assunto, nomeadamente como escolher o nome para a classe e alguns exemplos de *parsers* C++.

6.7.3 *Parsing* a partir de *strings*

Para processar um ficheiro, é necessário redirecciona a variável `yyin` do *FLEX* para o ficheiro pretendido. No caso de se pretender processar uma `string`, é necessário criar um `buffer` usando a função do *FLEX* `yy_scan_string`. Como a criação do buffer é realizado no *BISON* é necessário ao executar o *FLEX* gerar o ***`header file`*** para inclusão no *BISON* usando a opção "`flex -header-file=nome.h`". De seguida é apresentado um extracto de um ficheiro *BISON* para analisar uma *string*.

```
1 %{
2     #include "exemplo.h"
3 %}
4
5 ...
6
7 int main(){
8   char my_char_ptr[] = "123 asda, 123.23, 134,
        asd1231asd121";
9   YY_BUFFER_STATE str_buffer = yy_scan_string(my_char_ptr);
10  yyparse();
11  yy_delete_buffer(str_buffer); /* free up memory */
12  ...
13 }
```

6.8 Propostas de Exercícios

1. Crie o programa "Hello World" com o *BISON* e o *FLEX*. Os *tokens* existentes são `HELLO` e `WORLD`. Sempre que o texto a analisar tiver os dois *tokens* pela ordem certa, deve imprimir a frase "Hello World!!!".

2. Implemente um analisador sintáctico para reconhecimento duma expressão aritmética, utilizando o *BISON* e o *FLEX*. A gramática é a seguinte:

   $\mathbf{S} \rightarrow \mathit{ID}$ *'='* $\mathbf{E} \mid \mathbf{E}$
   $\mathbf{E} \rightarrow \mathbf{E}$ *'+'* $\mathbf{E} \mid \mathbf{E}$ *'-'* $\mathbf{E} \mid \mathbf{E}$ *'*'* $\mathbf{E} \mid \mathbf{E}$ *'/'* $\mathbf{E} \mid$ *'-'*$\mathbf{E} \mid$ *'('* $\mathbf{E}$ *')'* $\mid \mathit{ID} \mid \mathit{INT} \mid \mathit{REAL}$

   Em que `ID` é um identificador (letra de 'a' a 'z'), `INT` um número inteiro e `REAL` um número real.

   O *parser* deve analisar múltiplas expressões e obter resultados, apresentando-os. Sempre que haja uma atribuição esse valor deve ser guardado, para ser utilizado com o identificador respectivo em outras expressões. Como tabela de símbolos, utilize um vector com uma posição para cada letra.

   (a) Implemente este exercício sem usar precedências de operadores;

   (b) Implemente este exercício usando precedências de operadores;

3. Considere um simulador de uma máquina de venda automática que dispõe de um conjunto de produtos e aceita moedas em euros (€0.01, €0.02, €0.05, €0.10, €0.20, €0.50, €1.00, €2.00).

   O objectivo é seleccionar um produto, introduzir o respectivo valor, receber o troco (se existir) e receber o produto. Considere os seguintes produtos: café (€0.35), pingo (€0.35), chá (€0.35), chocolate (€0.40), copo (€0.05) e leite (€0.30).

   O formato de entrada de dados deve obedecer à seguinte regra: <**produto**>, <**moeda1**>, ...<**moeda***n*>. O formato de saída deve obedecer à seguinte regra: <**produto**>, <**moeda1**>, ...<**moeda***n*> | "*dinheiro insuficiente*".

   Exemplo:

   Entrada – café, €0.01, €0.10, €0.05, €0.20

   Saída – café, €0.10, €0.01

   Defina a gramática de modo a que a máquina funcione ininterruptamente e implemente-a utilizando o *FLEX* e o *BISON*.

4. Implemente um analisador sintáctico para reconhecimento duma expressão aritmética, utilizando o *FLEX* e o *BISON*. A gramática é a seguinte:

   $\mathbf{S} \rightarrow \mathit{DECL}\ \mathbf{L} \mid \mathit{VAR}\text{'='}\mathbf{E} \mid \mathbf{E} \mid \mathit{HELP}$
   $\mathbf{E} \rightarrow \mathbf{E}\ \text{'+'}\ \mathbf{E} \mid \mathbf{E}\ \text{'-'}\ \mathbf{E} \mid \mathbf{E}\ \text{'*'}\ \mathbf{E} \mid \mathbf{E}\ \text{'/'}\ \mathbf{E} \mid \text{'-'}\ \mathbf{E} \mid \text{'('}\ \mathbf{E}\ \text{')'}\mathit{VAR} \mid$
   $\mid \mathit{FUNCAO}\ \text{'('}\ \mathbf{E}\ \text{')'} \mid \mathit{INT} \mid \mathit{REAL}$
   $\mathbf{L} \rightarrow \mathit{VAR}\mathbf{R}$
   $\mathbf{R} \rightarrow \text{','}\ \mathit{VAR} \mid \varepsilon$

   Em que:

   DECL – é a palavra "decl" para declaração de uma variável;

   VAR – é um identificador de uma variável;

   HELP – é a palavra "help" ou o símbolo ?;

   INT – um número inteiro;

   REAL – um número real;

   FUNCAO – é o identificador para chamada a uma função predefinida (sqrt, factorial, etc.).

   O *parser* deve analisar múltiplas expressões e obter resultados, apresentando-os. Devem ser sempre definidas as variáveis que queremos utilizar, se uma variável tentar ser redefinida deve ocorrer um erro. Sempre que for inserido `HELP` devem ser apresentadas as variáveis e funções definidas. Sempre que haja uma atribuição, esse valor deve ser guardado para ser utilizado com o identificador respectivo. Se uma variável não estiver definida deve ser apresentado um erro.

   Como tabela de símbolos, utilize uma lista ligada, e use como tipo de *token* para a variável um apontador para essa lista.

5. Defina uma gramática que reconheça endereços URL e implemente-a utilizando o *FLEX* e o *BISON*. Para além do reconhecimento da validade de um endereço, deve ser apresentado um resumo dos seus componentes.

   Exemplo 1: para o endereço `http://user@pass:www.dei.isep.ipp.pt:8080/nova/index.html`, a resposta deverá ser:

   endereço válido
   protocolo: http
   utilizador: user
   palavra-chave: pass

máquina: www.dei.isep.ipp.pt

porta: 8080

caminho: nova

página: index.html

Exemplo 2: para o endereço `mailto:dei@isep.ipp.pt`, a resposta deverá ser:

endereço válido

protocolo: mailto

utilizador: dei

domínio: isep.ipp.pt

Considere como válidos os seguintes protocolos: `http`; `https`; `ftp`; `ftps`; `mailto`. Lembre-se que o utilizador, a password, a porta, o caminho, e a página podem não existir, caso em que, a sua indicação deve ser ignorada. O nome da máquina pode ser substituído pelo respectivo endereço IP. Considere ainda que os nomes das máquinas e dos ficheiros são apenas constituídos pelos caracteres de `a` a `Z`.

6. Suponha um simulador para cálculo das notas finais da disciplina de Linguagens de Programação. Pretende-se verificar, apenas, se um determinado aluno obteve aproveitamento ou não. Assim, o simulador deve estar constantemente a receber dados traduzidos por frases do tipo:

<**tipo**>(<**exame época normal**>|<**exame recurso**>
|<**exame Setembro**>)<**trabalho prático**>
<**nº aluno**><**turma**>

Os formatos de cada um dos campos é o seguinte:

<**nº aluno**> – 7 algarismos

<**tipo**> – N ou D conforme o aluno esteja inscrito no regime nocturno ou diurno

<**turma**> – 1 algarismo e 2 letras

<**exame época normal**> – $0 \leq$ inteiro ≤ 20

<**exame recurso**> – $0 \leq$ inteiro ≤ 20

<**exame Setembro**> – $0 \leq$ inteiro ≤ 20

<**trabalho prático**> – $0 \leq$ inteiro ≤ 20

Considere definida a função `procura(N)` que procura num ficheiro de alunos o nome do aluno com o número `N` e retorna uma cadeia com 60 caracteres contendo o nome do aluno.

Pretende-se que o simulador retorne informação sobre o aproveitamento do aluno, indicando todos os dados do mesmo (incluindo o nome), bem como a classificação final da disciplina (`CF`), obtida de acordo com a seguinte fórmula (`NFREQ` é a nota do trabalho prático e `PE` é a nota do exame):

$$\text{CF} = \frac{x\text{NFREQ} + y\text{PE}}{x + y} \quad \text{Em que} \begin{cases} x = 40\% \\ y = 60\% \\ \text{Min NFREQ} = 8.0 \\ \text{Min PE} = 9.0 \end{cases}$$

Pode utilizar o formato de saída que considerar mais adequado, tendo em atenção que se não forem atingidas as notas mínimas indicadas a classificação final deverá ser `SM`.

Defina a gramática de modo a que a máquina funcione ininterruptamente e implemente-a utilizando o *FLEX* e o *BISON*.