Licenciatura em Engenharia Informática – DEI/ISEP
Linguagens e Programação 2024/2025
Aula Prática-Laboratorial

Ficha PL 3

Flex e Bison

Objetivos:

- Introdução aos analisadores léxicos e sintáticos;
- Introdução às ferramentas FLEX e BISON;
- Comunicação entre BISON e FLEX;
- Tópicos avançados BISON e FLEX;
- Implementação de analisadores sintáticos.

1. Analisadores Léxicos

Um analisador léxico (scanner) é um programa que permite ler os caracteres de um ficheiro de texto (e.g., programa-fonte) e produzir uma sequência de componentes léxicos (tokens) que serão utilizados pelo analisador sintáctico (*parser*) e/ou identificar erros léxicos na entrada. Além da sua função básica, o analisador léxico está, geralmente, encarregue de realizar algumas tarefas secundárias, nomeadamente, a eliminação de comentários, espaços em branco e "tabulações".

Um *token* representa um conjunto de cadeias de entrada possível e por sua vez, um lexema é uma determinada cadeia de entrada associada a um *token*. Considere os exemplos apresentados na tabela 3.1.

Tabela 3.1: *Tokens* e lexemas

Tokens	Lexemas
FOR	for
IF	if
WHILE	while
NÚMERO	1089, 142857, 0.2, 3.14159
IDENTIFICADOR	i, j, contador, nomeAluno
OP_SOMA	+
OP_MAIOR_IGUAL	>=
ABRE_PAR	(
FECHA_PAR	)

O FLEX é uma ferramenta que permite gerar analisadores léxicos. Estes analisadores são capazes de reconhecer padrões léxicos em texto (e.g., números, identificadores e palavras-chave de uma determinada linguagem de programação). O analisador é construído com base num conjunto de regras. Uma regra é constituída por um par, padrão-ação, o padrão (expressão regular) descreve o elemento a reconhecer e ação (ou conjunto de ações) define o procedimento que será realizado no caso de identificação positiva do padrão. Uma expressão regular constrói-se com base num conjunto de meta-caracteres que são interpretados de forma especial, escritos numa dada sequência e

possivelmente intercalados com sequências de caracteres sem significado especial que não seja a sua verificação. O conjunto de palavras geradas por uma expressão regular designa-se uma linguagem regular.

3.1. *Modo de utilização do FLEX*

O ciclo de vida de um programa FLEX obedece à estrutura apresentada na figura 3.1.

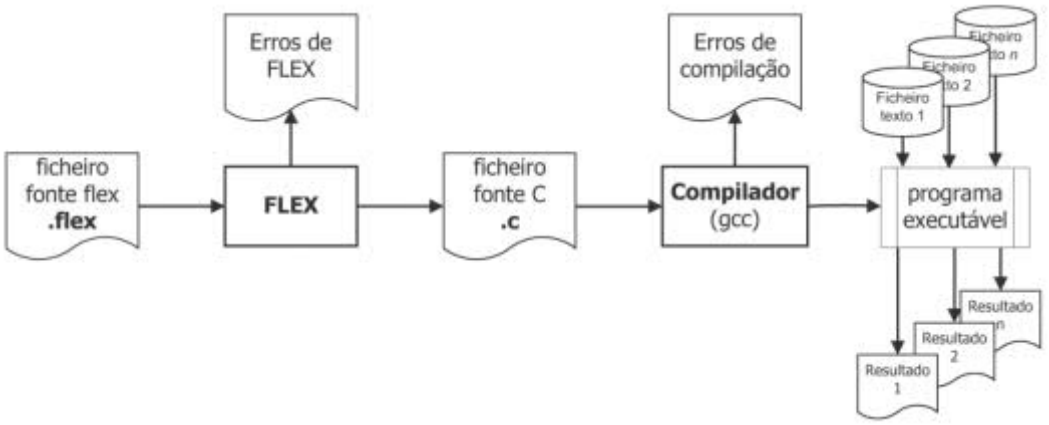

Figura 3.1: Ciclo de vida de um programa FLEX

Com base num ficheiro fonte escrito de acordo com a sintaxe do FLEX, o programa FLEX gerará um analisador léxico descrito na linguagem C. Em caso de existirem erros de codificação, o FLEX gerará uma listagem de erros. O ficheiro fonte em C terá de ser compilado para a plataforma em utilização utilizando um compilador da linguagem C adequado (neste caso o GCC). O resultado final da compilação será um programa executável capaz de identificar os padrões definidos pelo programado e levar o conjunto de ações previsto. Como entrada para o analisador 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. Considere-se a existência do ficheiro **`ficheiro.flex`** com o programa FLEX já escrito.

```
flex ficheiro.flex
gcc lex.yy.c -lfl
./a.out
```

O comando **`flex`** gera, por omissão, um ficheiro com o nome **`lex.yy.c`** que deverá ser compilado, por exemplo com 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`**).

```
flex -oExemplo.c Exemplo.flex
gcc Exemplo.c –o Programa -lfl
./Programa < Dados.txt
```

Neste exemplo, o comando **`flex`** gera a partir do ficheiro **`Exemplo.flex`**, o ficheiro com o nome **`Exemplo.c`** que deverá ser compilado. 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`**.

3.2. *Formato de um Ficheiro FLEX*

Um programa em FLEX é constituído por três secções, a saber, declarações, regras e rotinas auxiliares. A separação entre as secções é feita inserindo uma linha com o símbolo "%%". Considere-se o seguinte exemplo que será discutido nas secções seguintes.

```
1 %{
2     int numChars=0;
3 %}
4
5 %%
6
7 . {
8     numChars++;
9     printf("%s",yytext);
10  }
11
12 \n {
13      numChars++;
14      printf("\n");
15    }
16
17 %%
18
19 main()
20 {
21    yylex( ) ;
22    printf("Número de caracteres %d\n" , numChars);
23
24 }
```

Declarações

Regras (Definição de padrões e regras)

Rotinas em C

Declarações

Esta secção compreende duas partes:

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

```
1  /* Definição da variável numChars */
2 %{
3 int numChars=0;
4 %}
```

- Expressões regulares - podem ser declaradas macros para as expressões regulares mais comuns como por exemplo algarismo ou letra do alfabeto.

```
1  /* Definição de macros */
2 ALGARISMO [0-9] /* Algarismo */
3 ALFA [ a-zA-Z ] /* Letra do alfabeto */
```

A utilização de macros para expressões regulares será explicada com detalhe mais adiante.

Regras (definição de padrões e ações)

Nesta secção, são definidas as expressões regulares (padrões) e as respetivas ações que se pretendem realizar no caso da identificação positiva (*pattern matching*) do referido padrão.

No caso de um qualquer carácter excepto mudança de linha (representado por "**.**") é incrementada a variável **num_chars** e impresso o referido carácter no *standard* de saída. A mudança de linha (representado por "\n") é também contabilizada como um caráter e escrita no *standard* de saída.

As expressões regulares têm de ser obrigatoriamente escritas na primeira coluna do ficheiro.

```
1 . {
2      numChars++;
3      printf ("%s",yytext);
4   }
5
4
6 \n {
7      numChars++;
8      printf("\n") ;
9    }
```

Na secção, 1.6 são apresentados alguns dos padrões mais relevantes utilizados pelo *FLEX*.

O analisador léxico gerado funciona de acordo com as seguintes regras:

- Apenas uma regra é aplicada à entrada de dados;
- A ação executada corresponde à expressão que consome o maior número de caracteres;
- Caso existam duas ou mais expressões que consumam igual número de caracteres, tem precedência a regra que aparece em primeiro no ficheiro.

Quando um padrão é reconhecido, a sequência de caracteres consumida (*token*) na identificação do padrão pode ser obtida usando a variável **yytext** (do tipo **char ***). Para além disso, o comprimento da referida sequência é guardado na variável **yyleng** [1](do tipo **int**).

Rotinas em C de suporte

Nesta secção, pode ser escrito o código C que se pretende adicionar ao programa a gerar pelo FLEX. Tipicamente este código inclui o corpo do programa a função **main()** da linguagem C.

```
1 main ()
2 {
3    yylex () ;
4    printf ("Número de caracteres %d\n", numChars) ;
5 }
```

A função **yylex()** invoca o analisador léxico gerado pelo **Flex** que processará as expressões regulares anteriormente descritas (ver secção 1.2.2).

3.3. *Exemplo mais elaborado*

Considere o seguinte exemplo, no qual é contabilizada a quantidade de números e de linhas existentes noficheiro. Recorre-se à utilização de uma macro para a definição de algarismo (`ALGARISMO [0-9]`).

[1] O valor desta variável poderia ser obtido através da função da linguagem C strlen (yytext)

```
1 %{
2    int qtdNumeros=0, nLinhas=0;
3 %}
4
5 ALGARISMO [0-9]
6
7 %%
8
9  /* Se a ação for descrita numa só linha as chavetas podem ser omitidas */
10
11 \n            nLinhas++;
12 {ALGARISMO}+  {printf("d %s \n",yytext);qtdNumeros++;}
13 .
14
15 %%
16 main ()
17 {
18    yylex();
19    printf("#linhas=%d\n",nLinhas);
20    printf("#numeros=%d\n",qtdNumeros);
21 }
```

Todos os carateres não processados pelas duas primeiras expressões regulares são consumidos pela última à qual não corresponde nenhuma ação particular.

3.4. *Ambiente de trabalho*

O *FLEX* pode ser usado nas máquinas virtuais *LINUX*, ou utilizando o programa `putty` que está instalado em **`c:\putty\`** em modo **SSH** (ver figura 3.2). As máquinas a utilizar deverão ser **`ssh`**, **`ssh1`**, **`ssh2`** e **`ssh3`**.

A edição dos ficheiros fonte pode ser realizada a partir de qualquer editor de texto básico (*e.g.*, no ambiente `Windows` o **`Notepad++`**) desde que os ficheiros sejam gravados em formato *Unix* na área do utilizador.

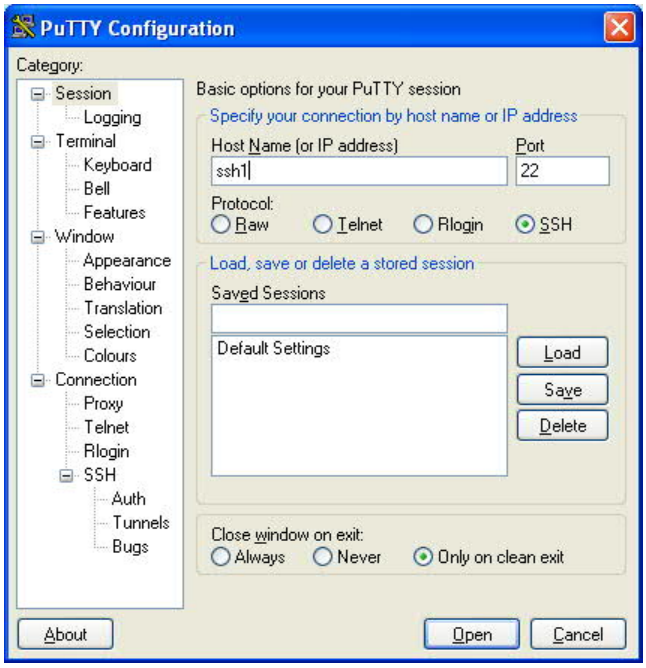

Figura 3.2: Acesso SSH via `putty`

3.5. *Resumo de comandos UNIX úteis*

Nesta secção, pretende-se apenas disponibilizar um resumo de comando UNIX. Na tabela 3.2 são apresentados os comandos que permitem fazer mudança de diretório e mostrar qual o diretório atual.

Tabela 3.2: Mudar e mostrar diretório

Comando	Descrição
`cd`	Mudar para o diretório *home*
`cd ..`	Mudar para o diretório pai
`cd <DIR>`	Mudar para o diretório `DIR`
`pwd`	Mostra o caminho completo do diretório atual

Na tabela 3.3 são apresentados os comandos que permitem listar o conteúdo de um determinado diretório.

Tabela 3.3: Listagem do conteúdo de um diretório

Comando	Descrição
`ls`	Mostra conteúdo do diretório atual
`ls –a`	*idem* incluindo ficheiros escondidos
`ls –la`	*idem* incluindo ficheiros escondidos sob a forma de lista
`ls <DIR> -l`	Mostra conteúdo do diretório `DIR` sob a forma de lista

Na tabela 3.4 são apresentados os comandos que permitem alterar as permissões de acesso a ficheiros e/ou diretórios. As permissões podem ser de três tipos:

- Leitura (***r***) - permite visualizar o conteúdo quando de se trate de ficheiros, no caso de diretórios, permite fazer um **`ls`**;
- Escrita (***w***) - permite alterar o conteúdo quando de se trate de ficheiros e no caso de diretórios permite criar ficheiros/diretórios no referido diretório;
- Execução (***x***) - permite executar um programa, quando de se trate de ficheiros, no caso de diretórios, permite aceder aos seus conteúdos;

Tabela 3.4: Alteração de permissões de acesso de ficheiros/diretórios

Comando	**Descrição**
`chmod u+rwx,gorw<FICH>`	Concede as permissões `rwx` para o utilizador; retira as permissões `rw` para utilizadores do mesmo grupo e outros; as outras permissões não são alteradas.
`chmod u+rwx,go+rx <DIR>`	Concede as permissões `rwx` para o utilizador; concede as permissões rx para utilizadores do mesmo grupo e outros; as outras permissões não são alteradas.
`chmod 751 <DIR>`	Concede somente as permissões: `rwx` (7=4+2+1) para o utilizador, `rx` (5=4+1) para utilizadores do mesmo grupo e `x(1)` para outros utilizadores.

As permissões de acesso a um ficheiro ou diretório estão subdivididas por três grupos de utilizadores:

- o próprio (*owner*);
- o grupo (*group*);
- restantes (*other*).

Considere o seguinte extracto resultado da execução do comando **`ls -la`** no qual o primeiro campo faz a codificação das permissões leitura, escrita e execução (**`rwx`**) para cada grupo. Estas permissões podem ser convertidas para um valor numérico somando 4 para **`r`**, 2 para **`w`** e 1 para **`x`**.

```
-rwxr    1 user     profs     25 Jan 27  2002 fich
drwxx    4 user     profs   4096 Jan 17 16:31 dir/
```

Neste exemplo, o ficheiro **`fich`** tem permissões de leitura, escrita e execução para o próprio, leitura para o grupo e nenhuma para os restantes utilizadores. Por sua vez, o diretóório **`dir`** (o primeiro carácter é um d no caso de diretóórios) tem permissões de leitura, escrita e execução para o próprio, execução para o grupo e nenhuma para os restantes utilizadores.

3.6. *Padrões utilizados no FLEX*

Na tabela 3.5 são apresentados alguns dos padrões mais relevantes utilizados pelo *FLEX*.

Tabela 3.5: Padrões utilizados no *FLEX*

Padrão	**Descrição**
x	O carácter “x”
.	Qualquer carácter excepto mudança de linha
\n	Mudança de linha
[xyz]	Um dos caracteres “x”, “y” ou “z”
xyz	A cadeia de caracteres “xyz”
[a-zA-Z]	Um dos caracteres no intervalo de “a” a “z” ou de “A” a “Z”
[-+*/]	Qualquer um dos operadores “-“, “+”, “*” ou “/”, sendo que o símbolo “-“ deve aparecer em primeiro lugar dada a possibilidade de ambiguidade com a definição de intervalo
[abj-oZ]	Um dos caracteres “a”, “b”, ou de “j” a “o” ou “Z”
[^A-Z\n]	Qualquer caracter exceto no intervalo de “A” a “Z” ou mudança de linha
r*	O carácter “r” zero ou mais vezes
r+	O carácter “r” uma ou mais vezes
r?	O carácter “r” zero ou uma vez
r{#$_1$,#$_2$}	O carácter “r” repetido no mínimo #$_1$ vezes, e no máximo #$_2$ vezes
r{#,}	O carácter “r” repetido pelo menos # vezes
r{#}	O carácter “r” repetido exactamente # vezes
{macro}	Substituição/Expansão da macro definida anteriormente
(r)	O carácter “r” sendo que os parêntesis permitem estipular precedências
xyz*	A sequência “xy” seguida de zero ou mais “z”s
(xyz)*	A sequência “xyz” repetida zero ou mais vezes
(r|s)	O carácter ”r” ou “s” (alternativa)
^r	O carácter ”r” apenas se no início da linha
r$	O carácter ”r” apenas se no final da linha (não consome o \n)
^xyz$	Uma linha que contém apenas a cadeia de caracteres “xyz”
<<EOF>>	Fim de ficheiro

3.7. *Exercícios propostos*

1. Escrever um programa que permite contar o número de ocorrências de uma cadeia de caracteres.
2. Escrever um programa que permite substituir as seguintes cadeias: “FEUP” por “ISEP” e “2007” por “2008”.
3. Escrever um programa que permite validar matrículas portuguesas.

4. Escrever um programa que dado um ficheiro de texto, mostra:
   - número de algarismos;
   - número de letras do alfabeto;
   - número de linhas de texto;
   - número de espaços ou tabulações (\t);
   - número de caracteres não identificados nos pontos anteriores.
5. Escrever um programa que permite identificar números naturais;

Entrada	Saída
123 abc 12.45 s 245 xyz	123 12 45 245
xyz 2 abc 45 cc	2 45

6. Escrever um programa que permite identificar números inteiros (com ou sem sinal).
7. Escrever um programa que permite identificar números com parte decimal (com ou sem sinal).

Bibliografia:

[1]. Manual do Flex (http://flex.sourceforge.net/)

[2]. Jeffrey D. Ullman, E. Hopcroft, Rajeev Motwani, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 2nd Edition, 2001.

[3]. Rui Gustavo Crespo, Processadores de Linguagens - da concepção à implementação, IST Press, 2001.

2. Analisadores Sintáticos

O BISON é um gerador de analisadores sintáticos de âmbito genérico, que converte uma gramática independente do contexto *LALR(1)*[2] 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 5.6 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;
- 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 squar e ( 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, 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
```

[2] Look-Ahead Left to right Rightmost derivation

```
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            ;
```

2.1 *Modo de utilização do 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 ações 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.

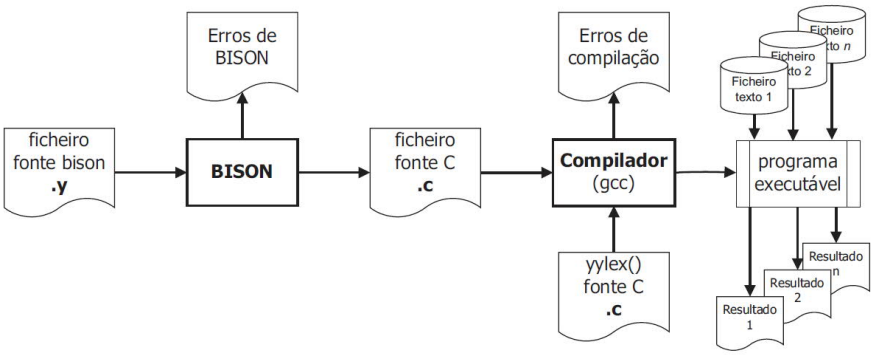

Figura 3.3: Ciclo de vida de um programa BISON

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 fi cheiro **`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**, os 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**.

2.2 *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 de 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 }
```

2.3 *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.

```
1 /* Definições do bison */
2 %token ID INT REAL
3 %start inicio
```

2.4 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 fará o que 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. De seguida, é apresentado um exemplo da definição de uma %union. Mais adiante será explicado como atribuir os campos aos *tokens* e regras.

```
1 %union
2 {     char   * id ;
3       int    inteiro ;
4       float real ;
5 }
```

2.5 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 na secção 5.3.1.

```
1 %token <id>        ID STRING
2 %token <inteiro>   INT
3 %token <real>      REAL
4 %type <real>       operando expressão
```

De notar que em cada declaração podem ser definidos vários *tokens* ou regras ao mesmo tempo, desde que partilhem o mesmo tipo. Neste caso, 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*, num dos campos 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;
```

2.6 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 na frase "x OPD y OPD z" seria realizada primeiro a operação "y OPD z" e depois 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 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 só será utilizado para atribuir uma precedência igual a esse *token*, com a declaração %prec MENOS_UNARIO que será acrescentada na gramática, à regra que pretendida com essa precedência.

Assim numa frase que use a gramática apresentada de seguida, serão realizadas em primeiro lugar as operações "negativo", depois todas as multiplicações e divisões, finalmente todas as adições e subtrações.

```
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 início, independentemente de ser a primeira ou não.

```
1 %start inicio
```

Gramática

Nesta secção é definida a gramática propriamente dita e as ações semânticas a ela associadas. As regras da gramática são definidas de acordo com a notação BNF, sendo a regra inicial a primeira ou a que foi especificada com a declaração %start.

As regras são declaradas em minúsculas, e o símbolo "definido como" é o símbolo ":". Os *tokens*, se tiverem mais de um caracter, são declarados em maiúsculas, ou, se tiverem somente um caracter, 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.

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.

Ações semânticas

Ao longo da gramática são inseridas as ações semânticas nas regras. Essas ações semânticas são instruções C entre chavetas acrescentadas às várias alternativas da regra. Apesar de normalmente as ações aparecerem somente no final da regra, estas podem ser colocadas em qualquer parte da regra com 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 ser tomado cuidado com as ações antes das regras, pois o *BISON* tem de saber qual é a alternativa a utilizar logo no início da regra. Se duas alternativas duma regra começarem pelo mesmo *token*, e existir uma ação semântica antes, o *BISON* apesar de não saber que alternativa utilizar, tem que escolher se executa a ação semântica ou não. Este problema resolve-se tornando as ações semânticas intermédias em ações semânticas finais, substituindo-as por uma 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 macros $1, $2 . . . $n, na qual o n representa a ordem pela qual o elemento aparece na regra. Existe ainda o $$, que representa o valor que a própria regra terá (lado esquerdo da regra).

A utilização de ações semânticas no meio de regras, ocupa um "$". 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".

No exemplo seguinte o regra "expressão" será a 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          ;
```

De notar que nas ações 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 ação semântica.

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. 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() invoca o analisador sintáctico gerado pelo *BISON* que processará a gramática anteriormente descrita (ver secção 5.3.2), devolvendo um valor diferente de zero, se encontrar um erro do qual não consegue recuperar ou zero se terminar com sucesso. É ainda possível saber quantos erros existem na gramática através da variável global yynerrs.

2.7 Comunicação com o analisador léxico

É necessário criar um analisador léxico (*scanner*) que forneça os *tokens* ao *BISON*, esse analisador léxico pode ser produzido no *FLEX*, sendo de seguida apresentado um exemplo para a gramática apresentada na secção 3.7.

```
1 %{
2   #include " exemplo . tab . h" /* header gerado pelo bison */
3   extern int numErros ; /* variável criada 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, isso é indicado ao *BISON* devolvendo o valor 0. Normalmente o "\n"é somente usado para contar linhas, podendo também ser passado ao *BISON* como um *token*. Sempre que o *token* é simplesmente um caracter, é devolvido o 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.

2.8 Exemplo mais elaborado

Este exemplo está disponível no moodle, com o nome ex_ficha5TP.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".

Ficheiro FLEX

```
1 %{
2   #include " exemplo . tab.h" // header greado pelo bison
```

```
3   extern int numErros ;
4 %}
5
6
7 %%
8
9 ,                          return yyt ext [ 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 %%
```

Ficheiro BISON

```
%{
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
37
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 }
```

2.9 Conflitos na gramática BISON

O funcionamento do *BISON* baseia-se numa pilha, onde são inseridos os *tokens* e os lexemas. Sempre que uma sequência de *tokens* faz *match* com a regra atual, 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 sintático gerado pelo *BISON*, é usando a opção "-v" , ou incluindo no ficheiro ".y" a directiva %verbose. Esta opção 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*.

Conflito reduce/reduce

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

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

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:

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

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

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 sempre ao if mais interior, que é o que o *BISON* faz realizando o shift em vez do reduce.

4. Tópicos avançados

3.1. *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 vírgula, 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   ;
```

- Geração de um *parser/scanner* numa classe

Se os nomes dos ficheiros *FLEX* e *BISON* terminarem em "+", o analisador léxico e o analisador sintático serão classes C++. Nos ficheiros de ajuda do *FLEX* e do *BISON* são incluídas secções especiais sobre este assunto.

- Parsing a partir de strings

Para processar um ficheiro, é necessário redireccionar 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 headerfile=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 }
```

3.2. *Contextos do FLEX (start conditions)*

O *FLEX* permite estabelecer contextos, para a aplicação das expressões regulares. Um contexto, é um estado especial, no qual só está activo um conjunto restrito de expressões regulares. Um dos exemplos mais conhecidos, implementado recorrendo a contextos, é o processamento de comentários multi-linha da linguagem de programação C. O contexto em que o *FLEX* se encontra por omissão, é o contexto "0" ou "INITIAL". Neste contexto todas as expressões regulares que não pertencem a nenhum contexto e as marcadas como "INITIAL" são usadas. Para definir novos contextos são usadas as directivas "%s" e "%x" na primeira secção do ficheiro *FLEX*, em que:

- A directiva "%s" define um contexto *inclusivo*, em que são aplicadas as expressões regulares desse contexto e as que não têm contexto;
- A directiva "%x" define um contexto exclusivo, isto é, quando o FLEX está nesse contexto, somente as expressões regulares marcadas com esse contexto estão activas.

A indicação dos contextos a que se aplica uma expressão regular é realizada, colocando um estado ou uma lista de estados, delimitado pelos símbolos "<" e ">", antes da expressão regular. Para expressões activas em todos os contextos, pode-se usar a notação "<*>".

A mudança de contextos é realizada nas ações com a macro BEGIN, indicando o contexto para o qual queremos mudar. De seguida é apresentado um exemplo que ignora os comentários da linguagem C, continuando a contar as mudanças de linha.

```
1 %x coment
2
3 %%
4
5 "/*"                  BEGIN coment; /* inicia comentário */
6 <coment>[^*\n ]*      /* ignorar tudo até um '* ' ou '\n ' */
7 <coment>"*/"          BEGIN INITIAL; /* finaliza comentário */
8 <coment>"*"           /* ignorar '*' sozinho */
9
10 /* a regra seguinte está activa nos dois contextos */
11<INITIAL , coment>\n numLinhas++; // equivalente a <*> verifica se não é
                                              "equivalente"
12
13 [0-9]+               return INTEIRO;
14 [0 -9]*\. [0-9]+     return REAL;
15
16 <*>[ \ t \ r ]       /*ignorar em todos os contextos */
17 <<EOF>>              return 0 ;
18 %%
```

O processamento de um comentário HTML é semelhante, sendo somente necessário substituir as expressões regulares de início e fim de comentário, pelas expressões regulares "''<!—'" e "''→''" respectivamente, e fazer algumas alterações às restantes expressões regulares.

No caso de processamento das linguagens XML e HTML, pode ser necessária a criação de contextos *FLEX*, para garantir que tudo que se situa fora das etiquetas é tratado como um único *token*. É necessário também permitir que apareçam palavras reservadas fora das etiquetas, e sejam tratadas como texto normal. No exemplo de código seguinte é apresentado o extracto de um ficheiro XML com alguns problemas de análise com o *FLEX* e o *BISON*, sem recorrer à utilização de contextos no *FLEX*.

```
1 <?xml version=" 1 . 0 "?>
2 <language>
3   <extension>xml</ extension>
4   <name>extensible markup language</name>
5   <update>2005-04-12 12:10:22</update>
6 </ language>
7 <language>
8   <extension>html</ extension>
9   <name>hypertext markup language</name>
```

```
10  <update>2003-09-24 19:4 4:0 1</update>
11 </ language>
```

No exemplo anterior a palavra xml aparece como etiqueta e como texto, é necessário identificar esta diferença recorrendo a dois contextos no *FLEX*, um quando se está dentro de uma etiqueta, e outro quando se está fora.

Também fora de uma etiqueta é necessário diferençar vários casos, dependendo da etiqueta a tratar. Por exemplo na etiqueta " extension" existe somente uma palavra, já na etiqueta "name" existe qualquer coisa (até encontrar o "<" seguinte) e na etiqueta "update" existe uma data que tem de ser validada do lado do *BISON*, necessitando dum contexto especial *FLEX* que identifique os inteiros e os símbolos "-" e ":" até encontrar o símbolo "<".

De seguida apresenta-se uma maneira de identificar a etiqueta "name" e "update" no *FLEX* e no *BISON*.

```
1 ...
2 language: '<' LANGUAGE '>'
3         dados_language
4         TAG_FIM LANGUAGE '>' { printf("language ...\n");}
5 ;
6 dados_language: extension name update
7                 |error {yyerror ("unrecognized language"); yynerrs++;}
8 ;
9 ...
10 name: '<'NAME'>' beginStrxml
11        strxml
12 TAG_FIM NAME '>' {printf ( "name ...%s \n",$5 ); }
13 ;
14 beginStrxml : {printf ("begin stringxml ...\n");
15               BEGIN stringxml;}
16 ;
17 update: '<' UPDATE '>' beginData
18        dataerr
19        TAG_FIM UPDATE '>' {printf ("data ... \n");}
20 ;
21 beginData: {printf ("begin data ... \n" );BEGIN data;}
22 ;
23 dataerr: data hora
24        |error hora {yyerror ("malformed data ... "); yynerrs++;}
25        |data error {yyerror ("malformed hora ... "); yynerrs++;}
26 ;
27 data:INTEIRO '-' INTEIRO '-' INTEIRO {printf("dia ...%d-%d-%d\n",$1 ,$3 ,$5);}
28 ;
29 hora:INTEIRO ':' INTEIRO ':' INTEIRO {printf("hora...%d:%d:%d\n",$1,$3,$5);}
30 ;
31 …
```

Sempre que é necessário usar a macro BEGIN no *BISON* para mudar o contexto do *FLEX*, deve ser criada uma regra vazia com a respectiva ação semântica, para esse fim (neste exemplo, são usadas as regras `beginStrxml` e `beginData`). Tal facto, deve-se à necessidade do *FLEX* mudar imediatamente para este contexto, o que pode não acontecer com a inclusão de ações semânticas no meio de regras.

Para poder usar a macro BEGIN e os contextos dentro do *BISON*, torna-se necessário incluir o ficheiro C gerado pelo *FLEX* no *BISON* e compilar somente o ficheiro *BISON*.

Um analisador léxico para esta gramática seria:

```
1
2 %{
3     extern int coluna , linha ;
4 %}
```

```
5
6 %x tag data stringxml
7
8 ID   [ a-zA-Z]+
9 INTEIRO    [0-9]+
10 REAL      [0-9]*\.[0-9]+
11
12 %%
13
14 <*>"<"           BEGIN tag; coluna+=yyleng; return yytext [0];
15 <*>"</"          BEGIN tag; coluna+=yyleng; return TAG_FIM;
16 <tag>">"         BEGIN INITIAL; coluna+=yyleng; return yytext [0];
17 <tag>xml         coluna+=yyleng; return XML;
18 <tag>extension   coluna+=yyleng; return EXTENSION;
19 <tag>language    coluna+=yyleng; return LANGUAGE;
20 <tag>name        coluna+=yyleng; return NAME;
21 <tag>update      coluna+=yyleng; return UPDATE;
22 <tag >[=?/]      coluna+=yyleng; return yytext [0];
23 <tag>\"{REAL}\" |
24 <tag >'{REAL}'   coluna+=yyleng;yylval.real=atof(yytext+1);returnREAL_STR;
25
26 <tag , data , INITIAL>[\t\r]  coluna+=yyleng ;
27 <tag , data , INITIAL>\n      coluna=0; linha++;
28
29 <data>{INTEIRO} coluna+=yyleng; yylval.inteiro=atoi(yytext);return INTEIRO;
30 <data >[:-] coluna+=yyleng; return yytext [0] ;
31
32 <stringxml>[^>]* BEGIN INITIAL; coluna+=yyleng;yylval.str=strdup(yytext); return
STRING_XML;
33
34 <*>[ \t\r ]            coluna+=yyleng;
35 <*>\n                  coluna=1; linha++;
36 <*>.                   coluna+=yyleng; printf("Erro lexico(%d-%d):simbolo
                        desconhecido (%c)\n", linha, coluna, yytext [0]);
37 <<EOF>>               return 0 ;
```

3.3. *Validação da existência de elementos por ordem arbitrária numa lista*

Para a validação da existência ou não de elementos numa lista e se eles estão repetidos, é necessário recorrer a ações semânticas para evitar a criação exponencial, de regras alternativas.

O exemplo seguinte aceita dentro da etiqueta uma lista de três campos obrigatórios, mas que podem aparecer por qualquer ordem. Para a sua validação é necessário usar um vector (ou uma estrutura) com um elemento para cada campo. A regra `lista` é a regra que permite inserir os vários campos, sendo na ação da condição de paragem (a primeira ação a ser realizada) iniciado o vector.

```
1 regra :
2   '<' TAG lista {
3         if ( vecCampos [0]==0)
4               yyerror ( " Falta campo 0" ) ;
5         else
6               if (vecCampos[0]>1) {
7                 sprintf ( str , "Campo 0 repetido %d vezes ",vecCampos[0]);
8               yyerror ( str ) ;
9                }
10        if ( vecCampos [1]==0)
11                    yyerror(" Falta campo 1" ) ;
12        else
13              ...
14              }
15        '/' '>
16        |
17        '<' TAG error ' / ' '> { yyerror( "Erro dentro da tag");}
18  ;
19
20  lista: /* Vazio */ {vecCampos[0]= vecCampos[1]= vecCampos[2]=0;}
```

```
21          |lista campo
22  ;
23
24 campo: CAMPO_0 {
25          if (vecCampos [0]>0) yyerror ("Campo 0 repetido" );
26                vecCampos [0]++;
27  }
28  |CAMPO_1 {
29          if (vecCampos [1]>0) yyerror ("Campo 1 repetido" );
30                vecCampos [1]++;
31  }
32  |CAMPO_2 {
33           if (vecCampos [2]>0) yyerror ("Campo 2 repetido" );
34                vecCampos [2]++;
35  }
36  ;
```

3.4. *Passagem de strings entre o FLEX e o BISON*

Na passagem de *strings* entre o *FLEX* e o *BISON* deve-se ter o cuidado de reservar espaço e copiar a *string* no *FLEX* (por exemplo usando a função strdup). No *BISON* deve ser libertado o espaço quando o *token* já não é utilizado. Isto pode ser realizado de duas formas diferentes, a primeira é incluindo a instrução free directamente nas ações semânticas, a outra é incluindo a instrução free na directiva do *BISON* "%destructor" (somente disponível a partir da versão 1.9 do *BISON*).

De seguida são apresentados dois pequenos exemplos do *FLEX* e do *BISON* que utilizam este mecanismo.

```
1 ...
2 %%
3 ...
4 [a-zA-Z] [ a-zA-Z0-9_]* yylval.str=strdup(yytext); return ID;
5 ...
6 %%
```

```
1
2 %union{
3   int inteiro ;
4   double real ;
5   char * str ;
6 }
7
8 %token <str> ID
9
10 %type <str> id
11
12 %destructor {free ($$);} ID id
13
14 %%
15 inicio : /* vazio */
16         |inicio id {printf( "id:%s\n",$2 ) ;
17               /* se não usar %destructor, fazerfree ($2) ; aqui */}
18 ;
19
20 id : ID {$$=$1 ; }
21 ;
22
23 %%
```

3.5. *Pure parsers e Reentrant scanners, para programas com threads*

Em virtude dos *parsers* e *scanners* gerados "por omissão" com o *BISON* e o *FLEX*, recorrerem a variáveis globais para a comunicação entre si, não podem ser usados em programas que utilizem

threads ou que necessitem fazer análise simultânea a dois ficheiros. Neste tipo de programas, as variáveis de comunicação, têm que ser passadas como argumentos para as funções yylex e yyparse. Isso é conseguido usando *parsers* puros no *BISON* e "*Reentrant scanners*" no *FLEX* (só é possível com as versões linux).

Reentrant scanners

Para tornar o analisador léxico compatível com um *parser* puro, é necessário no *FLEX* realizar as seguintes alterações:

```
1   /* para não necessitar de -lfl */
2 %option noyywrap
3   /* para o bison passar 2 argumentos com yyloc e yylval */
4 %option bison-bridge bison-locations
5   /* para o flex não usar variáveis globais */
6 %option reentrant
7   /* para criar a função yy_push_state que substitui a macroBEGIN
                                ( só para quem usa o BEGIN no bison )*/
8 %option stack
9   /* para criar os ficheir o scanner.c e scanner.h */
10 %option outfile=" scanner.c " header-file="scanner.h"
```

Nas linhas que usam a variável yylval, deve ser alterado o "." para "->", visto a variável yylval ser agora um apontador para a estrutura. A contagem das linhas manual, deve ser substituída pela instrução:

```
1   yyset_lloc(yylloc, yyscanner);
```

Em caso de erro léxico, podem ser usados os campos da estrutura apontada por yylval para informar do local do erro:

```
1   printf ( "Erro lexico (%d,%d): simbolo desconhecido (%c)\n" ,
          yylloc->first_line, yylloc->first_column, yytext[0]);
```

Quando se usa o BEGIN para mudar contextos a partir do *BISON*, deve ser criada uma nova função no ficheiro *FLEX* (como a apresentada de seguida), e definir novas macros para os contextos que vai utilizar no *BISON*, e que serão copiadas também para o ficheiro *BISON*.

```
1 %%
2 /* inserir no final do ficheiro flex */
3
4 // copiar também para o bison
5 // definir uma macro para cada contexto usado no bison
6
7 #define MACRO_STRXML 1
8 #define MACRO_OUTRO 2
9
10 // fim de cópia para o bison
11
12 void push_state (int contexto, yyscan_t scanne r )
13 {
14  switch (contexto){
15        case MACRO_STRXML:
16          yy_push_state (stringxml, scanner); //stringxml é o contexto flex
17          break ;
18        case MACRO_DATA:
19          yy_push_state (data,scanner);// data é o contexto flex
20          break ;
21  }
22 }
```

Pure parser

Para tornar o analisador sintático puro, no ficheiro *BISON* devem ser realizadas as seguintes alterações:

```
%{
  #include "parser.tab.h"
  #include "scanner.h"
  #include<stdlib.h>
  #include<stdio.h>

  // macros para cada contexto copiadas do flex

  #define MACRO_STRXML 1
  #define MACRO_OUTRO 2

  // prototipo da função criada no flex
  void push_state ( int , yyscan_t ) ;

  // Função de erro bison o primeiro e último parâmetros ,
  // os dois do meio são os especifica dos nas opções %parse-param abaixo
  void yyerror (YYLTYPE * loc , yyscan_t scanner , int * erros , char* s ) {
        (* nerros )++
        printf ("Erro sintáctico (%d,%d):
       %s\n", loc->first_line, loc->first_column, coluna, s );
  }
%}

 /* parâmetros a passar à função yyparse , podem ser acrescentados outros ,
 mas também têm que ser acrescentados na função yyerror */
%parse-param {yyscan_t scanner }
%parse-param {int *erros }
 /* parâmetros a passar à função yylex */
%lex-param {yyscan_t scanner }
 /*para passar a estrutura yylloc à função yylex,
                            onde podem ser controladas as linhas/colunas */
%locations
 /* para a função yyparse usar somente variáveis locais */
%pure_parser
```

A todas as chamadas à função yyerror, devem ser acrescentadas no início os parâmetros &yylloc e todos os parâmetros passados para a função yyparse, devendo ser declarada no bloco inicial do ficheiro *BISON*:

```
1   void yyerror (YYLTYPE * loc , yyscan_t scanner , int * erros , char* s ) {
2           (* erros )++;
3           printf ( "Erro sintactico (%d.%d,%d.%d) : %s \n" , loc->first_line ,
                        loc->first_column , loc->last_l ine , loc->last_column , s ) ;
4   }
```

Todas as ações para mudança do contexto no *FLEX*, devem ser substituídas pela chamada à função push_state criada no *FLEX*, com a respectiva macro.

```
beginStrxml : { printf ( " begin stringxml . . . \ n" ) ;
                push_state (MACRO_STRXML, scanner ) ;
                }
;
```

Integração

Para a integração, no ficheiro onde será usado o *parser*, deverá ser realizada a inclusão dos *headers* do *BISON* e do *FLEX*, por esta ordem, e definido o protótipo da função yyparse.

```
1   #include " parser.tab.h"
2   #include " scanner.h"
3
4   // protótipo da yyparse
5   int yyparse (yyscan_t scanner, int * erros);
```

Na função onde necessitarem realizar a análise da mensagem recebida, devem incluir o seguinte código:

```
1   // utilização do parser :
2
3   int nerros;
4   yyscan_t scanner;
5   YY_BUFFER_STATE buf;
6
7   nerros=0;
8   // criar o scanner
9   yylex_init (&scanner);
10  // definir o buffer do scanner
11  buf = yy_scan_string (mensagem, scanner);
12  // chamar o parser
13  yyparse (scanner, &nerros ) ;
14  // apagar o buffer
15  yy_delete_buffer (buf , scanner) ;
16  // apagar o scanner
17  yylex_destroy (scanner) ;
```

No final compilar tudo separado, usando o seguinte exemplo:

```
1 flex scanner.lex
2 bison -d parser.y
3 g++ parser.tab.c scanner.c main.c
```

O g++ pode ser trocado pelo gcc e os nomes dos ficheiros a gerar pelo *FLEX* estão definidos pela seguinte directiva, incluída no ficheiro *FLEX*:

```
1 %option outfile="scanner.c" header-file="scanner.h"
```

Neste exemplo só são passados para a função yyparse dois argumentos, o scanner e um contador de erros. Podem no entanto ser passados mais argumentos como foi descrito anteriormente fazendo as seguintes alterações:

- Acrescentar as linhas %parse-param no ficheiro *BISON*;
- Acrescentar os parâmetros na função yyerror e nas respectivas chamadas;
- Acrescentar os parâmetros no protótipo e na chamada à função yyparse, nos ficheiros que a usam.

Os exemplos usados nestes tópicos avançados, estão disponíveis no moodle com o nome topicos.zip.

- exemplo.lex, exemplo.y, exemplo.xml e exemplo1.xml

- acoessem.lex e acoessem.y

- scanner.lex, parser.y e main.c

5. Exercícios Propostos

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) Crie um analisador usando o FLEX e o BISON, que reconheça frases constituídas por dois inteiros separados por um operador relacional (=, <, >, <=, >=, <>). O analisador deve indicar se a frase está de acordo com a sintaxe, e se a comparação é verdadeira ou falsa.

   10 <= 20 '\n' - verdadeiro

   5 = 10 '\n' -falso

   120 <> 130 '\n' - verdadeiro

   > 10 '\n' - erro de sintaxe

3) Reescreva a gramática da alínea anterior, de modo a:

   i. aceitar inteiros e letras (a-z e A-Z);

   ii. aceitar várias comparações na mesma linha;

   iii. testar a incompatibilidade de tipos entre inteiros e letras;

   iv. fazer a recuperação dos erros ocorridos.

   10 <= 20 = 20 '\n' - verdadeiro verdadeiro

   5 < 10 >= 5 < 2'\n' - verdadeiro verdadeiro falso

   120 <> A '\n' - incompatível

   z <> A '\n' - verdadeiro

   > 10 '\n' - erro de sintaxe

4) Implemente a gramática do exercício 10 da ficha PL2 (declaração de variáveis em C) usando o FLEX e o BISON. Implemente estratégias de recuperação de erros.

5) Implemente um analisador sintático para reconhecimento duma expressão aritmética, utilizando o BISON e o FLEX. A gramática é a seguinte:

   S →ID ‘=‘ E|E

   E → E ‘+‘ E|E ‘-‘ E|E ‘*‘ E|E ‘/‘ E| ‘-‘E| ‘ (‘ E ‘) ‘ | ID | INT |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 respetivo em outras expressões. Como tabela de símbolos, utilize um vetor com uma posição para cada letra.

   i. Implemente este exercício sem usar precedências de operadores;
   ii. Implemente este exercício usando precedências de operadores.

6) 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 objetivo 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>, . . .<moedan>.

O formato de saída deve obedecer à seguinte regra:

<produto>, <moeda1>, . . .<moedan> | *"dinheiro insuficiente".*

Exemplo:
Entrada - café, €0.01, €0.10, €0.05, €0.20
Saída - café, €0.01

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

7) 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 ≤ inteiro ≤ 20

<exame recurso> - 0 ≤inteiro ≤ 20

<exame Setembro> - 0 ≤ inteiro ≤ 20

<trabalho prático> - 0 ≤ 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):

$$CF = \frac{xNFREQ + yPE}{x + y} \begin{cases} x = 45\% \\ y = 55\% \\ \min NFREQ = 8.0 \\ \min PE = 8.0 \end{cases}$$

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.

Exercícios complementares

1. O gabinete de e-learning fazOScursos, pretende realizar a análise da quantidade de alunos e tipo de cursos lecionados por cada formador. Num ficheiro está reunida a informação de todos os cursos lecionados. O registo de cada formador é constituído por uma linha com a informação do formador (código e nome), seguido de linhas com a informação dos cursos leccionados (código, nome, ano curricular, alunos inscritos e carga horária). A informação está registada da seguinte forma, no formato EBNF (campos entre [ ] são opcionais e os campos entre { } repetem-se 0 ou mais vezes):

   {<cod_formador> [<nome_ formador>] '\n'

   {<cod_curso> [<nome_curso>] <ano_curricular> <alunos_inscritos> <carga_horaria>' \n'}}

   Em que o conteúdo de cada campo é o seguinte:

   - <cod_formador> – Sigla com 3 letras minúsculas seguido de 2 algarismos
   - <nome_ formador> – String entre aspas
   - <cod_curso> – Sigla com 3 letras maiúsculas seguida de 2 algarismos
   - <ano_curricular> – Inteiro maior que 0 e menor que 10
   - <nome_curso> – String entre aspas
   - <alunos_inscritos > – Inteiro maior que 0
   - <carga_horaria> – Inteiro maior que 0 e menor que 20

   Defina a gramática para o ficheiro anteriormente descrito, e crie utilizando o Flex e o Bison um programa que:

   1. Reconheça a validade do ficheiro;
   2. Indique para cada um dos formadores, o código do formador, o código do curso e o número de alunos inscritos, com o maior número de alunos;
   3. A quantidade de alunos de cada formador;
   4. Gere um erro sempre que encontre um erro sintático/léxico no ficheiro.

   O programa só deve parar no final do ficheiro, independentemente dos erros encontrados. O programa deve validar o ficheiro de dados e imprimir os resultados como no seguinte exemplo:

Input:	Output:

`ans23 "António Silva"`
`AAA32 3 125 12`
`MAT33 "Matemática I" 1 445 8`
`ANF42 "Análise Financeira" 5 12 4`
`abc34`
`ABC33 2 55 12`
`XYZ "Materiais" 125`
`NHL96 "Circuitos Elétricos" 2 178 9`
`FMI69 "Teoria de Empréstimos" 5 22 3`
	`ans23 MAT33 445 aluno(s)`
	`Total: 582 aluno(s)`
	`Erro:` `registo incompleto`
	`abc34 NHL96 178 aluno(s)`
	`Total: 255 alunos`

2. Crie e defina uma gramática que reconheça endereços URL e implemente-a utilizando FLEX e BISON. Para além disso, o reconhecimento de um endereço deve ser seguido pela indicação dos seus componentes.

   Exemplo para: `http://www.dei.isep.ipp.pt/nova/index.html`, a resposta deverá ser:

   endereço válido
   protocolo: http
   máquina: www.dei.isep.ipp.pt
   caminho: nova
   página: index.html
   Exemplo 2: para 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 quer o caminho, quer a página podem não existir, caso em que a sua indicação deve ser ignorada e que a máquina pode ser substituída pelo respectivo endereço IP. Considere ainda que os nomes das máquinas e dos ficheiros apenas são constituídos pelos caracteres de *a* a *Z*.

3. Crie Suponha um simulador de uma loja do Cidadão que pretende verificar apenas se a duração entre o horário de chegada e partida das pessoas em determinada loja corresponde àquele que foi previamente estimado de forma a optimizar o processo. Assim, o simulador deve estar constantemente a receber dados traduzidos por frases do tipo:

   <nº senha><data><hora chegada><hora partida>

   <nº loja>(<nome loja>)?<nº loja>

   Os dados estão armazenados num ficheiro de texto em que cada linha tem a forma acima indicada. Os formatos de cada campo são os seguintes:

   <**nº senha**> - inteiro maior de 0 e menor de 1000
   <**data**> - dd.mm.aa
   <**hora chegada**> - hh:mm
   <**hora partida**> - hh:mm
   <**nº loja**> - yyxx em que y é uma letra e x é um algarismo
   <**nome loja**> - cadeia com máximo de 35 caracteres delimitados por aspas

   Considere que a função procura(N), que procura num ficheiro com os dados referentes a previsão estimada de duração de atendimento do número da senha N e retorna o tempo estimado de duração de atendimento, já está definida.

   Pretende-se que o simulador retorne informação sobre eventuais erros de estimação de atendimento de pessoas, indicando todos os dados e o número de minutos de atraso (ou adiantamento) ou a informação "OK".

   Pode utilizar o formato de saída que considerar mais adequado. Defina a gramática de modo que a máquina funcione ininterruptamente e implemente-a utilizando o FLEX e o BISON.