

Compiladores

Aulas prácticas

© 2008 Pedro Freire

Este documento tem alguns direitos reservados:



Atribuição-Uso Não-Comercial-Não a Obras Derivadas 2.5 Portugal
<http://creativecommons.org/licenses/by-nc-nd/2.5/pt/>

Isto significa que podes usá-lo para fins de estudo.

Para outras utilizações, lê a licença completa. Crédito ao autor deve incluir o nome ("Pedro Freire") e referência a "www.pedrofreire.com".

REQUISITOS.....	6
PARA A ESCOLA.....	6
PARA O ALUNO.....	6
AULA 01.....	7
PRIMEIRO EXEMPLO.....	7
COMPILAR E CORRER.....	7
INTRODUÇÃO AO FORMATO DE FICHEIROS FLEX.....	8
FUNCIONAMENTO DAS REGRAS FLEX.....	8
AULA 02.....	10
AULA 03.....	12
EXERCÍCIOS.....	12
AULA 04.....	13
AULA 05.....	15
EXERCÍCIOS.....	16
AULA 06.....	17
START CONDITIONS.....	17
AULA 07.....	19
FORMATO DE FICHEIROS FLEX EM DETALHE.....	19
USAR FICHEIROS EM VEZ DE STDIN E STDOUT.....	21
REJECT, YYMORE() E OUTRAS FUNCIONALIDADES FLEX.....	22
EXEMPLO FINAL 1: CALCULADORA DE INTEIROS, PÓS-FIXA.....	22
EXEMPLO FINAL 2: CONTADOR DE IDENTIFICADORES.....	22
EXERCÍCIO FINAL FLEX.....	23
AULA 08.....	24
PRIMEIRO EXEMPLO BISON.....	24
A GRAMÁTICA DA CALCULADORA.....	25
ANÁLISE GRAMATICAL: YYPARSE().....	25
SÍMBOLOS TERMINAIS E VALORES SEMÂNTICOS.....	26
INTERLIGAÇÃO ENTRE OS FICHEIROS FLEX E BISON.....	27
FORMATO DO FICHEIRO BISON.....	28
EXERCÍCIO.....	28
AULA 09.....	29
REGRAS BISON.....	29
COMO FUNCIONAM AS REGRAS NO PRIMEIRO EXEMPLO.....	29
EXEMPLO DE ANÁLISE GRAMATICAL.....	30
CONFLITOS SHIFT/REDUCE: %LEFT, %RIGHT E %EXPECT.....	31
EXERCÍCIO 1.....	32
EXERCÍCIO 2.....	33
EXERCÍCIO 3.....	33
AULA 10.....	34

MÚLTIPLOS TIPOS DE DADOS SEMÂNTICOS: %UNION	34
EXERCÍCIO 1	35
EXERCÍCIO 2	35
BIBLIOGRAFIA.....	36

Requisitos

Para a escola

Requisitos para as salas das aulas práticas de Compiladores:

- Um qualquer Linux instalado, desde que suporte o `flex` e o `bison`, nativo, em *dual-boot* ou máquina virtual (VirtualPC ou VMware). Pode ainda ser acedido remotamente via Telnet/SSH. Pode também ser usado Windows (versão 95 ou acima), desde que funcional em relação aos pontos seguintes.
- `flex` e `bison` instalados no sistema operativo, e assegurar a existência do compilador de C (`gcc` ou `cc`) assim como do seu correspondente utilitário `make`, acessíveis a partir de qualquer directório (ou seja, instalados no *path*).
- No caso do Linux, assegurar que este tem ambiente visual instalado de fácil acesso (par utilizador+senha), assim como editor de texto visual (à semelhança do Bloco de Notas do Windows).
- Acesso à Internet com um *browser*.

Deve haver 1 PC por aluno.

Cada aula está programada para uma duração de 1,5h.

Para o aluno

Comparência nas aulas. Este guião tem propositadamente omissos certos elementos importantes para a compreensão total da matéria (notas históricas, relações entre partes diferentes da matéria, avisos sobre erros comuns, etc., ou seja, elementos para uma nota 20), embora seja suficiente para passar com nota bastante acima de 10.

Deves ter instalado o `flex` e o `bison` em computador próprio se quiseres acompanhar a matéria em casa. Consulta www.pedrofreire.com para uma pequena introdução de como o fazer. Vê também a secção acima para requisitos. Não é no entanto de todo necessário que tenhas estes sistemas em casa para conseguires passar à cadeira (podes usá-los na escola).

Esta cadeira assume que já tens experiência de programação na linguagem C.

Aula 01

Introdução e contextualização: o `flex` e o `bison`. Hiperligações.

Seu uso no desenho de compiladores.

Expressões regulares e gramáticas.

Referências e avaliação.

Primeiro exemplo

Primeiro exemplo `flex` (encontra números inteiros):

```
%option main
#include <stdio.h>
%%
[0-9]+ puts( yytext );
.|\\n|\\r
```

Grave este exemplo como `aula01.l` (a extensão é por convenção um “L” minúsculo).

Compilar e correr

Para compilar este programa precisa de ter o `flex` instalado, assim como um compilador de C qualquer. Então, numa linha de comandos / consola / terminal, execute:

```
flex aula01.l
```

Isto irá criar um ficheiro `lex.yy.c` que deverá ser compilado na linha de comandos ou no seu compilador favorito. Por exemplo, com o compilador da GNU, faça:

```
gcc lex.yy.c
```

Isto cria um ficheiro executável, cujo nome depende do compilador. Com o compilador da GNU (e a maior parte dos compiladores Unix) é criado um ficheiro `a.out`. Corra-o:

```
./a.out
```

Em Windows seria apenas

```
a
```

já que o compilador da GNU cria um ficheiro `a.exe` neste sistema.

O programa parece não fazer nada. É normal: está a ler dados do teclado. Escreva qualquer coisa que inclua números: ele repete os números (cada inteiro numa linha diferente) e ignora o resto. Quando tiver terminado use Ctrl+C para sair. Este é o 1º exemplo flex.

Em programas futuros vai ser prático testar o mesmo texto vezes sem conta. Para evitar termos de escrever sempre o mesmo texto no teclado, podemos colocá-lo num ficheiro separado (e.g.: teste.txt) e depois fazer:

```
./a.out < teste.txt
```

ou

```
a < teste.txt
```

Desta vez o programa irá mostrar imediatamente os resultados e sair, sem pausa.

Introdução ao formato de ficheiros flex

Os ficheiros flex têm o seguinte aspecto mínimo:

```
Definições flex
%%
Regras flex
```

No primeiro exemplo, acima, a secção de definições tem uma directiva `%option` do flex para criar uma função `main()` básica que avalia as expressões regulares a partir do teclado, e uma directiva `#include` para o código C usado. Os espaços atrás do `#include` são importantes!

A secção de regras flex tem sempre várias linhas com o seguinte aspecto:

```
expressão-regular      acção
```

Cada acção pode ser vazia (não fazer nada), ser só o caracter “|” (que significa “a mesma acção da expressão regular seguinte”) ou ser código em C. Pode-se ter mais de uma linha de código em C se este estiver envolvido em `{ }`.

Este ficheiro será analisado em mais detalhe em aula posterior.

Funcionamento das regras flex

O código gerado pelo flex, vai analisar os dados de entrada (tipicamente o teclado) à procura de qual expressão regular se aplica:

- Assim que uma se aplica, corre a acção correspondente, “consome” os caracteres dos dados de entrada que foram “encontrados” pela expressão regular, e repete todo o processo.
- Se mais do que uma se aplica, usa a que “encontra” mais caracteres, ou a primeira do ficheiro se “encontrarem” a mesma quantidade de caracteres.
- Se nenhuma se aplica, exhibe um caracter dos dados de entrada no ecrã, “consome-o” e repete o processo.

No exemplo acima, queremos exibir números inteiros no ecrã, e mais nada. A primeira regra exibe os números: a expressão regular

```
[0-9]+
```

detecta números inteiros (como veremos mais tarde), e a acção

```
puts( yytext );
```

exibe o número que foi encontrado no ecrã, uma vez que a variável `yytext` do `flex` contém a *string* de dados de entrada que foi encontrada pela expressão regular.

Para evitar que “outros” caracteres sejam exibidos no ecrã (só queremos exibir os números inteiros, e o comportamento do `flex` quando não encontra uma expressão é exibir esse caracter no ecrã), temos de anular o comportamento habitual do `flex`.

Para isso criamos uma regra que se aplique a qualquer caracter (mas só 1). Essa expressão regular é:

```
.\n|\r
```

e associamos a essa expressão regular uma acção vazia.

Aula 02

Expressões regulares comuns:

Caracteres	
Cada caracter "não especial" representa-se a si mesmo	
.	(ponto) Representa qualquer caracter (só 1), excepto o fim-de-linha (EOL)
[c]	Conjuntos ("classes de caracteres") – representam 1 só caracter; c pode conter caracteres avulso (sem separadores, e.g.: [abc]) ou gamas (<i>ranges</i>) (e.g.: [a-z]); se o 1º caracter de c for ^, o conjunto é todos os caracteres excepto aqueles em c
Repetições	
c*	O caracter c repetido zero ou mais vezes
c+	O caracter c repetido uma ou mais vezes
c?	"Opcional" – o caracter c zero ou uma vez
c{#}	O caracter c repete-se exactamente # vezes
c{#,}	O caracter c repete-se # ou mais vezes ({0,} = *, {1,} = +)
c{#1,#2}	O caracter c repete-se no mínimo #1 vezes, e no máximo #2 vezes ({0,1} = ?)
Agrupamentos	
(r)	Agrupamento de expressões (e.g.: para fazer com que uma repetição se aplique a mais do que um caracter)
r s	"ou"

Todas as expressões de repetição (*, +, ? e { }) são gananciosas (*greedy*).

O "ou" avalia as expressões da esquerda para a direita e assim que uma é encontrada (*matched*), avança com essa sem testar as restantes.

Exemplos, assumindo que o utilizador escreve

Copacabana

quando corre o programa flex:

- Com a expressão: `a.a`
são encontradas (*matched*) as seguintes ocorrências marcadas com fundos verde e amarelo (cores intercaladas para cada ocorrência):

Copacabana

Note-se que a sequência sobreposta "aba" não é encontrada.

- Com a expressão: `[abc]`
são encontradas: Copacabana
- Com a expressão: `[A-Z]`
é encontrada: Copacabana
- Com a expressão: `a.*a`
é encontrada: Copacabana (* é ganancioso)
- Com a expressão: `(a.)*`
é encontrada: Copacabana
- Com a expressão: `a.?`
são encontradas: Copacabana

Mais exemplos:

- Expressão para detectar um número de PIN ("####"):
`[0-9]{4}`
- Expressão ingénua para detectar um dos 4 números de um número IP:
`[0-9]{1,3}`
- Expressão ingénua para detectar uma *string*:
`["].*["]`
(as aspas têm de ser representadas como `["]` – veremos porquê mais tarde).
Problema: com o texto: `var="a"+x+"b";`
esta expressão encontra: `var="a"+x+"b";` (porque o * é ganancioso)
- Expressão melhorada para detectar uma *string*:
`["][^"]*["]`
Com o texto: `var="a"+x+"b";`
esta expressão encontra: `var="a"+x+"b";`

É um erro comum não reparar que `[ana] ≠ ana`.

Aula 03

Exemplos resolvidos:

1. Lista de inteiros separados por vírgulas (e.g.: "1,2,3,4").
Se vamos ter uma lista, precisamos de usar uma expressão de repetição (*, +, etc.), mas aquilo que se repete (um inteiro seguido de uma vírgula) já não se repete no último elemento (que deixa de ter vírgula à direita).

A solução é exprimi-lo outra vez:

$([0-9]+,)*[0-9]+$ ou
 $[0-9]+(,[0-9]+)*$

2. Número inteiro entre 1 e 20.
Não conseguimos validar gamas numéricas (apenas gamas de caracteres).
Então validamos gamas numéricas pelo seu aspecto (em termos de caracteres):

$20|1[0-9]|[1-9]$

Ou seja:

20 ou 10-19 ou 1-9

Note-se no "ou" a ordem dos mais longos para os mais curtos!

Exercícios

Exercícios flex com expressões regulares:

1. Matrículas portuguesas
2. Idade (1 a 150 inclusive)
3. Nº de cartão de crédito
(opcionalmente separado por "-" ou espaço a cada 4 dígitos)
4. IPv4
5. Número real
(inteiro, com parte decimal opcional e parte científica opcional)

Os 2 primeiros serão resolvidos no quadro nos últimos 20min.

Correcção na aula seguinte.

Aula 04

Resolução dos exercícios da aula anterior:

1. As matrículas têm os seguintes aspectos válidos:

##-##-LL
##-LL-##
LL-##-##

onde # é um dígito numérico e L uma letra maiúscula.

Isto não funciona porque permite combinações inválidas:

$(([0-9]\{2\} | [A-Z]\{2\}) -)\{2\} ([0-9]\{2\} | [A-Z]\{2\})$

Então temos mesmo de ter as 3 opções diferentes na expressão:

$([0-9]\{2\} - [0-9]\{2\} - [A-Z]\{2\}) |$
 $([0-9]\{2\} - [A-Z]\{2\} - [0-9]\{2\}) |$
 $([A-Z]\{2\} - [0-9]\{2\} - [0-9]\{2\})$

(na mesma linha e sem espaços)

2. Validamos gamas numéricas pelo seu aspecto (em termos de caracteres):

$[1-9] | [1-9][0-9] | 1[0-4][0-9] | 150$

Ou seja:

1-9 ou 10-99 ou 100-149 ou 150

Simplificando e colocando pela ordem correcta (devido ao "|"):

$150 | 1[0-4][0-9] | [1-9][0-9] ?$

3. Os números de cartão de crédito têm os seguintes aspectos válidos:

####-####-####-####
####

onde # é um dígito numérico. O enunciado permite maior flexibilidade, e.g.:

#####-####

Logo a expressão será uma lista de 4 grupos de 4 dígitos numéricos cada:

$([0-9]\{4\} [-]?)\{3\} [0-9]\{4\}$

Note-se que o espaço separa uma expressão regular da acção correspondente:

como tal, ele deve ser *escaped* para ser usado numa expressão regular. Neste caso, ao ser usado num conjunto, ele é automaticamente *escaped*.

4. Um número IP versão 4 tem o seguinte aspecto:

.#.#.#

onde # é um número de 0 a 255 inclusive.

A expressão será então uma lista de 4 números:

$(r[.])\{3\}r$

onde r é a expressão regular para um número entre 0 e 255:

$[0-9] | [1-9][0-9] | 1[0-9]\{2\} | 2[0-4][0-9] | 25[0-5]$

que, simplificada, pela ordem correcta e com os parêntesis, fica:

$(25[0-5] | (2[0-4] | 1[0-9] | [1-9])?[0-9])$

5. Um número real pode ter os seguintes aspectos válidos, e.g.:

15
+8
-0.5
.7
2.
3.001
3E8
+8.6E-8

Se verificarmos, isto significa que, à exceção de “.7” e “2.”, o formato de um número real é:

$\pm\#(\. \#)?(E\pm\#)?$

onde # representa um número inteiro. Então, em expressão regular:

$[+-]?[0-9]+([\.,][0-9]+)?([Ee][+-]?[0-9]+)?$

Mas e o “.7” e “2.”? Não podemos simplesmente substituir os dois primeiros + de repetição por * porque isso permitiria que “.” e “+.”, entre outros, fossem considerados números reais. Então temos de exprimir os 3 casos possíveis de forma a excluir o 4º caso inválido:

$[+-]?([0-9]+([\.,][0-9]*)?|[\.,][0-9]+)([Ee][+-]?[0-9]+)?$

onde o que está a cor mais clara é a parte modificada da expressão anterior.

Aula 05

Expressões regulares específicas do flex:

Caracteres e escapes	
<code>[[:c:]]</code>	Conjunto pré-definido (“expressões de classes de caracteres”) – ver manual do flex
<code>"s"</code>	<code>s</code> literal – para usar em <code>s</code> o próprio caracter <code>"</code> , fazer <code>\"</code> , e para usar o próprio <code>\</code> , fazer <code>\\</code>
<code>\c</code>	Se <code>c</code> for <code>θ</code> , <code>a</code> , <code>b</code> , <code>f</code> , <code>n</code> , <code>r</code> , <code>t</code> ou <code>v</code> , então é o caracter ANSI-C correspondente, caso contrário é um <code>c</code> literal (usado para fazer <i>escaping</i> de caracteres especiais)
<code>\###</code>	Caracter com o código ASCII octal <code>###</code>
<code>\x##</code>	Caracter com o código ASCII hexadecimal <code>##</code>
Expressões	
<code>{n}</code>	Expansão de <code>n</code> , sendo <code>n</code> um nome (ver exemplo, abaixo)
<code>r/s</code>	Encontra <code>r</code> apenas se for seguido por <code>s</code> (<code>s</code> não fará parte de <code>yytext</code>); nem todas as expressões regulares podem ser usadas em <code>s</code> – ver manual do flex
Âncoras	
<code>^</code>	Neste sítio tem de existir o início de uma linha
<code>\$</code>	Neste sítio tem de existir o fim de uma linha
<code><<EOF>></code>	Neste sítio tem de existir o fim do ficheiro / dos dados de entrada

As âncoras não representam caracteres. Representam sim, posições no ficheiro ou nos dados de entrada (ver exemplo abaixo).

Exemplo:

```
%option main
#include <stdio.h>
DIGITO      [0-9]
%%
{DIGITO}+   printf( "Encontrei o inteiro %s\n", yytext );
"aluno[s]"  puts( "Encontrei a expressão \"aluno[s]\"" );
^Ze$       puts( "Encontrei \"Ze\" sozinho numa linha" );
<<EOF>>    puts( "Encontrei o fim do ficheiro" );
.|\\n|\\r
```

Para testar o EOF, faça Ctrl+D em Unix ou Ctrl+Z em Windows.

Questão: porquê %s no printf()?

Exercícios

Faça um pequeno programa flex para:

1. Comentários C++ //...
2. Comentários C /*...*/
3. Strings C/C++ "...\"..."

Correcção na aula seguinte.

Aula 06

Resolução dos exercícios da aula anterior:

1. `"/" . * ou [/] [/] . * ou \ / \ / . *`
2. Primeira hipótese:
`"/" . "*" /"`
Para apanhar EOL:
`"/" (. | \n | \r) "*" /"` (usar Ctrl+D/Z ao testar)
Para evitar comportamento ganancioso:
`"/" ([^ *] | [*] + [^ /]) "*" /"`
Para aceitar `****/`:
`"/" ([^ *] | [*] + [^ /]) * [*] + [/]`
3. `["] ([^ \ \] | \ \ ["]) * ["]`
Para aceitar qualquer código de escape oferecido pelo C:
`["] ([^ \ \] | \ \ ([^ \ 0 - 7 x X] | \ 0 | [0 - 7] { 3 } | [x X] [0 - 9 a - f A - F] { 2 })) * ["]`

Start Conditions

Start condition (condição de arranque) é um estado em que o flex se encontra. Dependendo desse estado, uma ou mais regras deixam de ter efeito (deixam de estar activas).

Declaramos as *start conditions* na secção de definições:

Declarar <i>start conditions</i>	
<code>%s s</code>	<i>Start condition</i> inclusiva: quando estiver activa, as regras sem <i>start condition</i> definida, ou com as <i>start conditions</i> <code>s</code> ou <code>*</code> serão as únicas activas
<code>%x s</code>	<i>Start condition</i> exclusiva: quando estiver activa, apenas as regras com as <i>start conditions</i> <code>s</code> ou <code>*</code> estarão activas

A *start condition* `INITIAL` (ou `0`) está pré-definida.

Troca-se de *start condition* com a chamada `BEGIN(s)` em qualquer bloco de código em C. Pode-se ler a *start condition* actual num bloco C lendo a macro `YY_START`. Existem ainda funções para gerir uma pilha de *start conditions* (ver manual do flex e `%option stack`).

Especificamos que regras pertencem a que *start conditions* com:

<i>Start conditions</i>	
<code><*>r</code>	A expressão regular <code>r</code> estará activa em qualquer <i>start condition</i> , mesmo uma exclusiva
<code><s>r</code>	A expressão regular <code>r</code> estará activa apenas na <i>start condition</i> <code>s</code>
<code><s₁, s₂>r</code>	A expressão regular <code>r</code> estará activa apenas nas <i>start conditions</i> <code>s₁</code> e <code>s₂</code>
<code><s>{ r₁ r₂ r₃ }</code>	As expressões regulares <code>r₁</code> , <code>r₂</code> e <code>r₃</code> estarão activas apenas no âmbito (<i>scope</i>) da <i>start condition</i> <code>s</code>

Exemplo: Programa `flex` que recebe um ficheiro ASP ou PHP e “retira-lhe” o HTML e CSS, exibindo apenas todo o código *server-side*:

```
%option main
#include <string.h>
%x ASP
%x PHP
%%
"<%"?          BEGIN( ASP );
"<?"(=[a-z]*)  { if( strlen(yytext) <= 2    ||
                  !strcmp(yytext, "<?=")  ||
                  !strcmp(yytext, "<?php") )
                  BEGIN( PHP );
                }
.|\n|\r
<ASP>"%>"      BEGIN( INITIAL );
<PHP>"?>"      BEGIN( INITIAL );
<ASP,PHP>.\n|\r ECHO;
```

`ECHO` é uma macro definida pelo `flex` que simplesmente exhibe `yytext` e então equivale a:

```
fputs( yytext, yyout );
```

onde `yyout` (variável do `flex`) é normalmente `stdout` (ecrã). Veremos em aula posterior como podemos dizer ao `flex` para exibir os resultados num ficheiro.

Exercício: Repetir os exercícios da aula anterior, mas recorrendo a *start conditions*.
Correcção na aula seguinte.

Aula 07

Resolução dos exercícios da aula anterior:

C++

```
%option main
%x COMENT
%%
"/"          BEGIN( COMENT );
.|\\n|\\r
<COMENT>.    ECHO;
<COMENT>\\n|\\r BEGIN( INITIAL );
```

C

```
%option main
%x COMENT
%%
"/*"        BEGIN( COMENT );
.|\\n|\\r
<COMENT>"/" BEGIN( INITIAL );
<COMENT>.|\\n|\\r ECHO;
```

Strings

```
%option main
#include <stdio.h>
%x STR
%%
["]          BEGIN( STR );
.|\\n|\\r
<STR>["]     BEGIN( INITIAL );
<STR>\\\\"["]   putchar( '\"' );
<STR>.      ECHO;
```

Exercícios de revisão (na aula).

Formato de ficheiros flex em detalhe

Como foi visto na 1ª aula, os ficheiros flex têm o seguinte aspecto geral:

```
Definições flex
%%
Regras flex
%%
Código de utilizador em C
```

A secção de definições pode ter:

```
%{
  Inicializações de variáveis C e funções/protótipos C
  (opcional, podem também ser linhas indentadas, sem %{ %})
}%

%directiva

/* Comentário, copiado para o ficheiro C final */

nome expressão-regular
```

No último exemplo da aula anterior, temos as directiva `%option` e `%x` do `flex`, e uma directiva `#include` para o código C usado.

A secção de regras `flex` tem o seguinte aspecto:

```
%{
  Inicializações de variáveis e código C locais à função
  yylex() (opcional, podem também ser linhas indentadas,
  sem %{ %})
}%

expressão-regular acção
```

Cada acção pode ser vazia (não fazer nada), ser só o caracter “|” (que significa “a mesma acção da expressão regular seguinte”) ou ser código em C. Pode-se ter mais de uma linha de código em C se este estiver envolvido em `{}`.

A expressão regular está separada da sua acção por um ou mais espaços ou *tabs*. Isto significa que estes caracteres devem estar *escaped* na expressão regular para serem usados como parte da expressão (*escaped* significa usar `[]` à volta do espaço, por exemplo – mais sobre isso em aula anterior).

A função `yylex()` é uma função criada pelo `flex` no código C gerado, função essa que corre as expressões regulares e as acções que lhes correspondem. Note-se que no primeiro exemplo da aula 01, `%option main` gera a seguinte função `main()`:

```
int main( void )
{
  yylex();
  return 0;
}
```

Estas regras vão ser convertidas para código C dentro de uma função `yylex()`. É por esta razão que a 3ª secção, que o `flex` simplesmente copia para o ficheiro final, tem a função `main()` a chamar `yylex()`.

Na secção de código de utilizador costuma ter-se a nossa função `main()` e outras funções, variáveis e estruturas necessárias ao programa. Devido à `%option main` usada, estas são criadas automaticamente pelo que o exemplo acima não tem esta secção.

Usar ficheiros em vez de stdin e stdout

O flex permite trabalhar com ficheiros em vez da entrada e saídas padrão (stdin e stdout respectivamente):

Trabalhar com ficheiros	
yyin	FILE* de onde o flex lê os dados
yyout	FILE* para onde a macro ECHO do flex escreve dados
yyrestart(FILE*)	Função flex para alterar yyin – alterar yyin directamente não funciona como se espera porque o flex faz <i>caching/buffering</i> de yyin.

Para exemplificar, vamos criar um programa flex que converte ficheiros com terminações de linha Unix (apenas \n) e MacOS antigo (apenas \r) em Windows (\r\n). Ele recebe o nome de dois ficheiros na linha de comandos: o primeiro é o ficheiro origem, e o segundo é o nome destino para esse ficheiro, após conversão.

```
%option nostdinit noyywrap
    #include <stdio.h>
%x EOL_MAC

%%

\r          fputs("\r\n", yyout); BEGIN(EOL_MAC);
\n          fputs("\r\n", yyout);
[^\r\n]+    ECHO;
<EOL_MAC>\n BEGIN(INITIAL);
<EOL_MAC>\r fputs("\r\n", yyout);
<EOL_MAC>.  ECHO; BEGIN(INITIAL);

%%

int main( int argc, char *argv[] )
{
    if( argc >= 3 )
    {
        yyrestart( fopen(argv[1], "r") );
        yyout = fopen( argv[2], "w" );
        yylex();
        fclose( yyout );
        fclose( yyin );
    }
    return 0;
}
```

Neste exemplo, como estamos a pedir ao flex para não inicializar as variáveis yyin e yyout (no %option nostdinit), podíamos substituir a linha yyrestart() por

```
yyin = fopen( argv[1], "r" );
```

REJECT, ymore() e outras funcionalidades flex

O flex tem algumas outras macros e funções definidas para nos auxiliarem a criar programas. Estas devem ser consultadas no manual do flex. Entre as funções que poderão ser de mais interesse e que deves consultar encontram-se o REJECT e o ymore().

Exemplo final 1: Calculadora de inteiros, pós-fixa

Este exemplo flex pretende ser uma calculadora de inteiros com suporte às 4 operações básicas, em modo pós-fixa (*postfix*, também chamada de calculadora RPN – *reverse polish notation*, ou notação polaca inversa), onde o operador segue os dois operandos (e.g.: “2 5 +” seria 7).

```
%option main
#include <stdio.h>
#include <stdlib.h>
int a = 0;
int b = 0;

%%
int r;

[+-]?[0-9]+ a = b; b = atoi(yytext);
[+]         r=a+b; printf("%d+%d=%d\n", a,b,r); a=b; b=r;
[-]         r=a-b; printf("%d-%d=%d\n", a,b,r); a=b; b=r;
[*]         r=a*b; printf("%d*%d=%d\n", a,b,r); a=b; b=r;
[/]         r=a/b; printf("%d/%d=%d\n", a,b,r); a=b; b=r;
.|\\r|\\n
```

Exemplo final 2: Contador de identificadores

Este exemplo flex pretende ser um contador de identificadores e inteiros num código fonte em qualquer linguagem. Um identificador é definido como uma sequência de um ou mais caracteres alfanuméricos ou *underscore* (sublinhar). O identificador não pode começar com um carácter numérico.

```
%option noyywrap
#include <stdio.h>
enum { TOKEN_IDENT, TOKEN_INT, TOKEN_MISC, TOKEN_EOF };

%%

[a-zA-Z_][a-zA-Z0-9_]*      return TOKEN_IDENT;
[+-]?[0-9]+                return TOKEN_INT;
.|\\r|\\n                  return TOKEN_MISC;
<<EOF>>                    return TOKEN_EOF;

%%
```

```

int main()
{
int num_idents = 0;
int num_ints   = 0;
int token;

do    {
      switch( token=yylex() )
      {
        case TOKEN_IDENT:    ++num_idents;
                             break;
        case TOKEN_INT:      ++num_ints;
                             break;
        /* outro TOKEN_*: ignorar */
      }
    }
    while( token != TOKEN_EOF );

printf( "Identificadores: %d\n", num_idents );
printf( "Inteiros: %d\n",      num_ints );
return 0;
}

```

Repare-se que aqui as acções fazem `return`, saindo assim da função `yylex()`. Esta é uma forma comum de permitir que a função `yylex()` seja apenas auxiliar em detectar *tokens*, e não tenha o trabalho central de processamento dos mesmos. Esta será a forma de usar o `flex` quando avançarmos para o `bison`.

Exercício final flex

Combine os dois exemplos anteriores, criando uma calculadora de números reais, infixa, mas onde a leitura de cada *token* (elemento) da calculadora é feita pela função `main()`, chamando o `yylex()` para ler cada *token*, como o exemplo 2, acima.

Correcção no final da aula seguinte.

Aula 08

Correcção do exercício da aula anterior.

Primeiro exemplo bison

Descarrega o primeiro exemplo `bison`. É uma calculadora RPN tal como o nosso exemplo final 1 do `flex`, da aula anterior. Para o compilar só precisas de escrever o comando `make`*. Em alternativa podes compilá-lo “à mão” com os seguintes comandos:

```
bison -d rpn.y -o rpn.tab.c
flex rpn.l
gcc rpn.tab.c lex.yy.c -Wall -o rpn
```

Ele consiste de 5 ficheiros, que são:

- `bison.hairy` e `bison.simple` – necessários para algumas versões de `bison`, tipicamente em Windows. Fazem parte do `bison` em si e não devem ser alterados.
- `Makefile` ou `Makefile.mak` – ficheiro que tem os comandos necessários para construir (*make*) o nosso executável. Nota que podes ter de fazer algumas alterações para o conseguires correr em Linux (vê comentários ao topo do ficheiro).
- `rpn.l` – ficheiro `flex` que detecta os *tokens* da nossa calculadora.
- `rpn.y` – ficheiro `bison` que tem as regras gramaticais da nossa calculadora.

Este programa consiste de uma série de variáveis e funções definidas separadamente pelo `flex` e `bison`, mas que devem trabalhar em conjunto. Ao nível mais abstracto (da gramática) são as regras `bison` que trabalham, enquanto que ao nível mais baixo (detecção de *tokens*) são as regras `flex` que trabalham.

Vamos começar pelo abstracto, pela visão global de como fazer esta calculadora.

* O teu compilador de C tem de ter este executável, caso contrário deves descarregá-lo de outro compilador (e.g.: da GNU) ou executar os comandos individuais à mão.

A gramática da calculadora

Uma operação em RPN tem o seguinte aspecto, e.g.:

```
2 5 +
```

Mas podemos combinar operações, e.g.:

```
2 3 2 + +
```

ou

```
1 1 + 5 +
```

Em ambos os casos chamamos a estas combinações de operações de “expressões”. Isto é um nome comum em linguagens de programação e Matemática. Ambas as expressões acima fazem a conta “2+5”.

Então, embora ingenuamente nós possamos definir uma operação RPN como tendo o seguinte aspecto:

```
operando operando operador
```

na realidade cada operando pode ser o resultado de uma operação, e pode então ser uma expressão em si. Então o aspecto de uma operação RPN é:

```
expressão expressão operador
```

desde que cada expressão possa também ser definida como simplesmente uma constante numérica. É exactamente isto que encontras ao final do ficheiro `rpn.y`, na definição de `expr`.

`expr` é algo abstracto no ficheiro `rpn.y`. Não tem um único correspondente real nas operações que escreves na calculadora. É um nome que serve apenas de auxiliar abstracto para descrever o aspecto geral das operações. Chama-se então a este nome de “símbolo não-terminal”.

Explicar como fazer gramáticas não faz parte do âmbito das aulas práticas, mas sim das aulas teóricas. Nas aulas práticas vamos estudar apenas como implementar as gramáticas no `bison`.

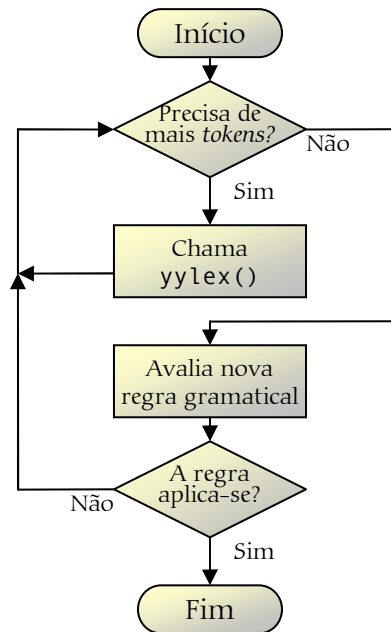
Análise gramatical: `yyparse()`

O `bison` (ou melhor, a função `yyparse()` de análise gramatical gerada pelo `bison`) precisa de reconhecer qual regra se aplica a cada instante. Para isso, para além da gramática precisar de ter certas características que irás aprender nas aulas teóricas, também precisa de ocasionalmente se referir aos elementos que o utilizador escreve (elementos aos quais se chama de “símbolos terminais” ou *tokens*).

Esta referência tem o detalhe necessário para cada gramática. Por exemplo, na definição de `expr` no ficheiro `rpn.y`, vemos que não quero nem preciso de distinguir o número 1 do número 2 do número 3 e do número 5 (ou outros). Preciso só de saber que encontrei um `NUMERO`.

Por outro lado, nesta gramática dá-me jeito distinguir as 4 operações suportadas, pelo que tenho os 4 *tokens* `OP_SOMA`, `OP_SUB`, `OP_MULT` e `OP_DIV`.

A descrição do “aspecto” de cada símbolo terminal (*token*) não é do âmbito de `yyparse()`. Enquanto está a distinguir qual regra se aplica, esta função pode estar a comparar os *tokens* que já encontrou (mas ainda não “consumiu”) com as várias regras disponíveis. Mas quando estes *tokens* não são suficientes, ela precisa de obter “o próximo *token* disponível”. Isto faz-se chamando automaticamente uma função (`yylex()`) que deverá resolver esse problema. No nosso caso estamos a usar o `flex` para criar essa função.



O processamento de `yyparse()` é então semelhante ao diagrama exibido acima. Ele vai avaliando as regras gramaticais uma a uma até encontrar uma que se aplique, e conforme vai precisando de mais *tokens*, chama `yylex()`.

Símbolos terminais e valores semânticos

Definimos o aspecto dos símbolos terminais no ficheiro `rpn.l`, em formato `flex`. Esse ficheiro deve-te ser familiar e quase tudo deve ser óbvio.

Repara nas expressões regulares triviais que estamos a usar para definir cada *token*. Repara também como estamos a guardar o valor do número detectado (o chamado valor semântico do *token* `NUMERO`) numa variável global, tal como o tivemos de fazer para resolver o exercício final `flex` da aula anterior. Neste caso usamos uma variável global pré-definida pelo `bison` para esse efeito: `yyval`. O tipo de dados que esta variável guarda é o definido na macro `YYSTYPE*`.

* É possível que esta variável global seja uma união de vários tipos para quando a linguagem com que estamos a trabalhar suporta vários tipos (tal como a maior parte das linguagens de programação). Mais detalhes em aula posterior.

No `bison` temos um atalho para nos referirmos a esta variável: os `$$`, `$1`, `$2`, etc.:

Aceder a valores semânticos	
<code>\$\$</code>	Valor semântico da regra em si.
<code>\$#</code>	Valor semântico de cada um dos elementos da regra: <code>\$1</code> refere-se ao primeiro elemento da regra, <code>\$2</code> ao segundo, etc.. Ver exemplo.

Cada um dos `$#` estará a referir-se à respectiva variável `yyval`:

- Tal como atribuída na função `yylex()` que retornou esse *token*, ou
- Tal como definida pela atribuição `$$=...` na acção da regra gramatical correspondente.

É precisamente por cada regra também ter o seu valor semântico, que o `yyparse()` consegue encadear expressões umas nas outras.

Interligação entre os ficheiros `flex` e `bison`

Ambos os ficheiros se referem às mesmas constantes `NUMERO`, `OP_SOMA`, `OP_SUB`, `OP_MULT`, `OP_DIV` e `EOL`. Podíamos defini-las em ambos os ficheiros, mas se por engano `NUMERO` tem o valor 1 no `rpn.l` e o valor 2 no `rpn.y` (por exemplo) então o código não irá funcionar bem.

O `bison` tem uma directiva `%token` que nos permite definir estes *tokens* e posteriormente exportá-los (assim como algumas outras opções) para um ficheiro `rpn.tab.h` que devemos incluir no nosso ficheiro em C gerado pelo `flex`. Ficamos assim com um único ponto de definição de *tokens*: o ficheiro `bison*`.

Mas embora não tenhamos o inverso (um ficheiro `.h` do `flex` para o `bison`), precisamos de usar no ficheiro `bison` a função `yylex()` definida pelo `flex`. Como ela é sempre definida como recebendo nada (`void`) e devolvendo um inteiro, isto não é problema – temos simplesmente de nos lembrar de a definir como `extern` no ficheiro `bison` para evitar erros de ligação (*linkage*) após a compilação.

* Há outra razão para estarmos a usar o `%token` do `bison`: é que em certas situações precisamos de estabelecer no `bison` certas características dos *tokens*, ao mesmo tempo que os “definimos”. Mais sobre isto em aula posterior.

Formato do ficheiro bison

O formato do ficheiro `bison` é quase idêntico ao dos ficheiros `flex` que temos visto até agora:

```
Definições bison
%%
Regras bison
%%
Código de utilizador em C
```

A secção de definições pode ter:

```
%{
Inicializações de variáveis C e funções/protótipos C
(opcional)
%}

%directiva
```

As regras `bison` serão definidas em mais detalhe em aula posterior, mas pode ter-se uma ideia do aspecto normal destas regras neste primeiro exemplo.

Exercício

Altera o primeiro exemplo `bison` para passar a funcionar como uma calculadora infixa.

Simplifica depois o código: as directivas `%token` geram sempre números acima de 255 (`0xFF`) para os *tokens* definidos, isto para permitir que um *token* simples de um só carácter seja representado pelo seu próprio código ASCII, como em C (e.g.: `'c'`), sem necessitar de directiva `%token`. Simplifica então os ficheiros `rpn.l` e `rpn.y` para que as operações sejam especificadas desta forma.

Aula 09

Correcção do exercício da aula anterior. A resolução necessitava apenas de colocar os operadores entre os operandos na regra `expr`. Mas isto gera um aviso sobre 16 erros *shift/reduce*. Para os resolver precisamos de conhecer mais sobre o `bison`.

Regras `bison`

As regras `bison` são definidas da seguinte forma geral:

```
resultado: símbolo1 símbolo2 ... { acção1 }  
|          símbolo3 símbolo4 ... { acção2 }  
...  
;
```

Cada regra tem um nome associado (“`resultado`” no exemplo acima), e pode ter várias variantes (definidas em linhas diferentes e separadas por um caracter “|”). As várias variantes também podem ser definidas com o nome da mesma regra à esquerda, como em:

```
resultado: símbolo1 símbolo2 ... { acção1 }  
;  
resultado: símbolo3 símbolo4 ... { acção2 }  
;
```

Cada variante da regra tem zero ou mais componentes (símbolos terminais ou não-terminais) e, opcionalmente, uma acção (código C) envolvida em `{ }`. Os componentes determinam o aspecto desta variante da regra, em termos de posição e tipo de símbolos.

Cada regra deve terminar com um “;” numa linha independente.

Como funcionam as regras no primeiro exemplo

A regra `input` do primeiro exemplo diz que os dados oferecidos à calculadora podem ser vazios, ou uma sequência de linhas. Este tipo de regras diz-se “recursivas à esquerda” (devido à parte “`input linha`”) e é preferido em relação às equivalentes “recursivas à direita” (que seriam “`linha input`” – ver manual do `bison`).

A segunda regra determina o aspecto válido de cada linha: ou vazia, ou com uma única expressão (em cujo caso exibimos o seu valor).

A terceira e última regra define o aspecto válido de cada expressão em termos de si mesma. Conforme a função `yyparse()` gerada pelo `bison` vai subindo

semanticamente das expressões mais simples às mais complexas, as acções das regras respectivas correm e as contas parciais são feitas.

Exemplo de análise gramatical

Descarrega o exemplo “calc-sr” (Calculadora *shift-reduce*) e compila-o. Quando o correres, escreve:

1+2*3

Este exemplo mostra-te os passos que o `yyparse()` dá para analisar esta (ou qualquer outra) expressão. Estes são de três tipos:

- *Shift*: ler um novo *token* dos dados de entrada e colocá-lo numa pilha interna para processamento;
- *Reduce*: reduzir uma combinação de símbolos (terminais e/ou não-terminais) na pilha, a um seu equivalente gramático;
- *Accept*: Se já não há mais *tokens* a ler da entrada e se a pilha se reduz ao símbolo não-terminal inicial (neste caso, `input`), então termina o processamento com sucesso.

Todas as acções do `flex` correspondem então a *shifts* e todas as do `bison` correspondem a *reduces*. Com este programa vemos a seguinte sequência de acções:

Pilha	Tokens	Acção
(vazia)	1+2*3 EOL	<i>Reduce</i> : input ← (vazio)
input	1+2*3 EOL	<i>Shift</i> : 1
input 1	+2*3 EOL	<i>Reduce</i> : expr ← 1
input expr	+2*3 EOL	<i>Shift</i> : +
input expr +	2*3 EOL	<i>Shift</i> : 2
input expr + 2	*3 EOL	<i>Reduce</i> : expr ← 2
input expr + expr	*3 EOL	<i>Shift</i> : *
input expr + expr *	3 EOL	<i>Shift</i> : 3
input expr + expr * 3	EOL	<i>Reduce</i> : expr ← 3
input expr + expr * expr	EOL	<i>Reduce</i> : * expr ← expr * expr

* Porque não há operadores com maior precedência do que o * nesta calculadora. Ver secção seguinte sobre conflitos.

input expr + expr	EOL <i>Shift:</i> * EOL
input expr + expr EOL	<i>Reduce:</i> expr ← expr + expr
input expr EOL	<i>Reduce:</i> linha ← expr EOL
input linha	<i>Reduce:</i> input ← input linha
input	<i>Accept</i>

Este exemplo mostra-te dois pontos importantes:

- Existe sempre uma regra inicial. Ela é sempre a 1ª a ser especificada no ficheiro `bison` (regra `input` neste caso), mas podemos especificar outra com a declaração:

```
%start símbolo-não-terminal;
```

- Uma gramática `bison` só estará correcta se puder ser reduzida à regra/símbolo inicial em todas as combinações possíveis da sua linguagem.

Conflitos `shift/reduce`: `%left`, `%right` e `%expect`

Este tipo de conflitos surgem quando mais do que uma regra se pode aplicar em certas situações e o `yyparse()` não sabe se deve aplicar (*reduce*) a regra mais curta ou aceitar o novo *token* (*shift*) para aplicar a regra mais longa (que é o que fará por omissão).

Estes conflitos podem ser ignorados, mas podemos resolvê-los de duas formas:

1. Associatividade e precedência de operadores, ou
2. Dizer ao `bison` que já contamos com certa quantidade destes conflitos.

O primeiro caso usa-se quando podemos vir a ter ambiguidade com o uso de operadores. E.g., na nossa calculadora infixa:

```
x + y + z
```

Isto resolve-se alterando a declaração `%token` dos operadores em questão por:

Associatividade de operadores	
<code>%left</code>	Associatividade à esquerda, ou seja, fazemos <code>x+y</code> primeiro.
<code>%right</code>	Associatividade à direita, ou seja, fazemos <code>y+z</code> primeiro.
<code>%nonassoc</code>	Sem associatividade: a expressão acima é um erro de sintaxe.

* Na realidade aqui não ocorreu um *shift*, mas um “*look-ahead*” (ver manual do `bison`). Mas não vamos entrar em muitos detalhes e assumir que as coisas correm como é aqui descrito, por simplicidade.

Cada uma destas declarações pode levar à frente mais do que um operador. Todos esses terão a mesma precedência. Operadores declarados em declarações diferentes têm maior precedência (são executados primeiro) se forem declarados mais tarde (mais abaixo) no ficheiro `bison`.

Ou seja, a precedência correcta para as quatro habituais operações matemáticas é:

```
%left OP_SOMA OP_SUB
%left OP_MULT OP_DIV
```

A segunda forma de resolver os conflitos surge quando eles são esperados. Por exemplo:

```
regra:      IF expr THEN decl
|           IF expr THEN decl ELSE decl
;
```

Neste caso existe ambiguidade na aplicação das duas variantes da regra. Mas podemos acrescentar

```
%expect 1
```

ao topo do ficheiro `bison` para que ele espere este conflito, e só gere avisos se encontrar uma quantidade diferente de conflitos.

Exercício 1

Acrescenta à calculadora a capacidade de atribuir valores (expressões) a variáveis, criando-as se necessário. Cada variável é inteira, pode ter um nome arbitrário de até 32 caracteres alfanuméricos ou *underscore* (mas não pode começar num dígito numérico), e podem haver no máximo 100 variáveis definidas pelo utilizador. A atribuição de variáveis é ela mesma uma expressão, devolvendo o valor atribuído.

Acrescenta também a capacidade da calculadora fazer cálculos com base em decisões, ou seja fazer algo parecido a:

```
if( expr ) expr else expr
```

sendo a parte do `else`, opcional.

Dicas: as limitações às variáveis são para te facilitar o trabalho. Para cada variável preciso de guardar duas coisas: o nome dela e o valor que ela tem. Existem no máximo 100 variáveis. Então crio um vector de 100 estruturas com:

```
struct s_variavel
{
    char nome[32+1];
    int valor;
}
vector[100];
```

Pergunta: porquê 32+1?

É útil criares duas funções auxiliares: uma que pesquisa por uma variável com um dado nome, e outra que a cria se ela não existir. Ambas devem devolver um apontador para o vector, ou `NULL` se a variável não for encontrada ou o vector estiver cheio e a função não a puder criar.

O ficheiro `bison` vai precisar de receber do `flex` o nome da variável para além da indicação que encontrou um nome de variável. Como `yyval` é um inteiro, faço isso com uma nova variável global que crio. Não te esqueças que ela será definida normalmente num ficheiro e como `extern` no outro.

Para adicionares a decisão às regras da gramática, não te esqueças de adicionar a detecção dos seguintes *tokens* no ficheiro `flex`:

```
"if", "(", ")" e "else"
```

Exercício 2

Acrescenta à calculadora o comando

```
print expr
```

que exhibe o resultado de `expr` no ecrã.

Com isto, altera os níveis superiores abstractos da gramática (regras `input` e `linha`) para em vez dela ser “uma sequencia de linhas com expressões” passar a ser “uma sequência de expressões separadas por ‘;’”.

Naturalmente que agora os resultados dos cálculos só são exibidos no ecrã quando o utilizador o pedir explicitamente com `print`. Isto também significa que o *token* `EOL` é agora desnecessário.

Exercício 3

Acrescenta uma nova regra `afirm` à calculadora. Ela servirá para distinguir expressões de afirmações. Ajusta a calculadora para passar a usar esta regra para `if` e `print`. A principal distinção entre expressões e afirmações (*statements*) é que estes últimos não nos devolvem um valor que possamos usar em cálculos.

Cria a expressão condicional “?” que, tal como em C, serve para fazer um condicional dentro de uma expressão.

Aula 10

No exercício 1 da aula anterior vimos que para passar o nome da variável do flex para o bison precisámos de criar uma nova variável global porque a nossa variável de valor semântico (`yylval`) já era um inteiro. Mas como podemos então compilar linguagens que podem lidar com vários tipos de dados? Como passamos do flex *strings*, reais e inteiros para o bison?

Múltiplos tipos de dados semânticos: %union

Este problema resolve-se com recurso à directiva `%union`. Quando a usamos, devemos deixar de usar o `#define YYSTYPE` pois `yylval` vai passar a ser um tipo de dados composto com uma union de C, e vai passar a fazer parte de `rpn.tab.h`.

Usamos `%union` tal como em C, da seguinte forma:

```
%union {
    int valor_inteiro;
    char nome_de_variavel[32+1];
    ...
}
```

Podemos ter a quantidade e tipo de membros (variáveis) que desejarmos dentro desta union. Nota no entanto que esta `%union` não termina com `“;”` ao contrário de C.

Claro que agora precisamos de informar o bison e o flex de qual membro do `yylval` nos referimos em cada regra.

No flex isto é trivial. `yylval` é uma union C e então:

```
[0-9]+ { yylval.valor_inteiro = atoi(ytext); ... }
```

No bison, como nos referimos aos valores semânticos com `$$`, `$1`, `$2`, etc., podemos fazer também:

```
$1.valor_inteiro
```

ou definimos qual o tipo de dados a cada símbolo terminal ou não-terminal para o bison determinar o tipo de cada `$$`, etc., automaticamente:

```
%token <valor_inteiro> símbolo-terminal
%type <valor_inteiro> símbolo-não-terminal
```

`%type` é usado apenas para definir o membro da `%union` em símbolos não-terminais.

A partir daqui podemos usar `$$`, `$1`, `$2`, etc., sem especificar o membro da `%union`, que o `bison` saberá fazê-lo sozinho.

O seguinte exemplo define e distingue operações de soma (entre inteiros) e concatenação (entre *strings*):

```
(...)  
%union {  
    int inteiro;  
    char string[1000+1];  
}  
  
%token <inteiro> INTEIRO  
%token <string> STRING  
%left <inteiro> '+'  
%left <string> '.'  
%type <inteiro> expr_int  
%type <string> expr_str  
  
%%  
  
(...)  
expr_int: INTEIRO '+' INTEIRO { $$=$1+$3; }  
;  
expr_str: STRING '.' STRING {  
    strcpy($$, $1); strcat($$, $3); }  
;
```

Tem atenção que este exemplo tem uma forma ingénuo de lidar com *strings* (impedindo *strings* maiores de 1000 caracteres, mas não validando este limite). Os teus programas deveriam ser mais robustos.

Exercício 1

Altera o exercício 1 da aula anterior para passar a usar `%union` e passar o nome da variável também em `yylval`.

Exercício 2

Adiciona suporte a tipos de dados à tua calculadora. Ela deve passar a suportar variáveis reais para além de inteiras.

Não convertas todos os números lidos para reais. O comportamento da calculadora deve ser como em C: uma operação que tenha ambos os argumentos inteiros deve fazer o cálculo em inteiro (mesmo uma divisão). Uma operação que tenha pelo menos um dos argumentos reais, deve converter ambos para reais e fazer o cálculo em número real. O `print expr` também deve exibir um inteiro ou real dependendo do tipo de dados do resultado da expressão.

Bibliografia

[FLEX]

"flex: The Fast Lexical Analyzer", site oficial

No site da SourceForge em:

<http://flex.sourceforge.net/>

[BISON]

"Bison – GNU parser generator", site oficial

No site da GNU em:

<http://www.gnu.org/software/bison/bison.html>

[GNUWIN32]

"GnuWin32", Utilitários da GNU compilados para Windows.

Inclui flex, bison e make.

No site da SourceForge em:

<http://gnuwin32.sourceforge.net/>

[C-PROG]

"The C Programming Language", Prentice Hall Software Series,

Brian W. Kernighan e Dennis M. Ritchie

Na Amazon do Reino Unido, em:

<http://www.amazon.co.uk/C-Programming-Language-2nd/dp/0131103628/>

(mais em breve)