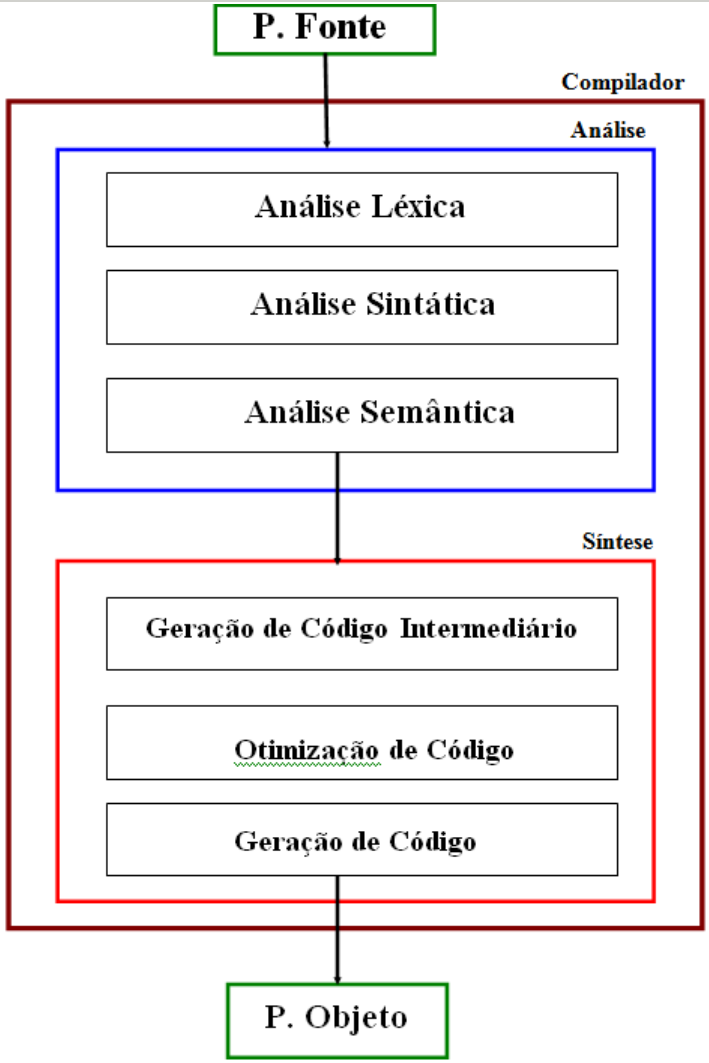
Análise Léxica - flex

Profa. Fabrícia Damando

Estrutura de um compilador



Análise Léxica

- Analisador léxico:
  - lê o arquivo fonte em busca de unidades significativas (os tokens) instanciadas por lexemas ou átomos.
  - contar as linhas de um programa
  - eliminar comentários
  - contar a quantidade de caracteres de um arquivo
  - tratar espaços

- *Tokens* – padrões de caracteres definido por um alfabeto, símbolos.
- *Lexemas* – são as ocorrências de um token em um código-fonte – conhecido como átomos

Muito utilizado em compiladores

USANDO E DESENVOLVENDO LEX/FLEX

- Um analisador léxico (scanner) é um programa que permite:
  - ler os caracteres de um ficheiro de texto (e.g., programa-fonte)
  - produzir uma sequência de componentes léxicos (tokens)
    - serão utilizados pelo analisador sintático (parser) e/ou identificar erros léxicos na entrada.

- Portanto:
  - O analisador léxico simplesmente varre a entrada (arquivo fonte) em busca de padrões pertencentes a uma linguagem.
  - A única possibilidade de ocorrer erro é aparecer um caracter que não pertence ao alfabeto da linguagem

Exemplo

Token	Padrão	Lexema	Descrição
`<const, >`	Sequência das palavras c, o, n, s, t	const	Palavra reservada
`<while, >`	Sequência das palavras w, h, i, l, e	while, While, WHILE	Palavra reservada
`<if, >`	Sequência das palavras i, f	If, IF, iF, If	Palavra reservada
`<=, >`	<, >, <=, >=, ==, !=	==, !=	
`<numero, 18>`	Dígitos numéricos	0.6, 18, 0.009	Constante numérica
`<literal, "Olá">`	Caracteres entre ""	“Olá Mundo”	Constante literal
`<identificador, 1>`	Nomes de variáveis, funções, parâmetros de funções.	nomeCliente, descricaoProduto, calcularPreco()	Nome de variável, nome de função
`<=, >`	=	=	Comando de atribuição

```
printf("Total = %d\n", score)
```

onde:

- `printf` e `score` são lexemas que casam com o padrão `identificador`.
- `Total = %d\n` é um lexema que casa com o padrão `literal`.
- `(` `)` lexemas que auxiliam a identificação de uma função.

TOKENS

Os tokens são símbolos léxicos reconhecidos através de um padrão.

Os tokens podem ser divididos em dois grupos:

•**Tokens simples: são** tokens que não têm valor associado pois a classe do token já a descreve.
• Exemplo: palavras reservadas, operadores, delimitadores: `<if,>`, `<else>`, `<+,>`.

•**Tokens com argumento:** são tokens que têm valor associado e corresponde a elementos da linguagem definidos pelo programador.

•Exemplo: identificadores, constantes numéricas - `<id, 3>`, `<numero, 10>`, `<literal, Olá Mundo>` .

Um token possui a seguinte estrutura:
`<nome-token, valor-atributo>`

Expressões Regulares

- Expressões regulares ou **regex** são uma forma simples e flexível de ***identificar cadeias de caracteres*** em palavras.

- Elas são escritas em uma linguagem formal que pode ser interpretada por um processador de expressão regular que examina o texto e identifica partes que casam com a especificação dada

- São muito utilizadas *para validar entradas de dados, fazer buscas, e extrair informações de textos*.

- As expressões regulares não validam dados, apenas verificam se um texto está em uma determinado padrão

- As expressões regulares estão diretamente relacionadas a autômatos finitos não determinísticos.

- As **regex** são utilizadas por editores de texto, linguagem de programação, programas utilitários, IDE de desenvolvimento e compiladores e seu padrões são independentes de linguagem de programação.

- As expressões regulares dão origem a algoritmos de autômatos finito determinísticos e autômatos finitos não determinísticos que são utilizados por analisadores léxicos para reconhecer os padrões de cadeias de caracteres.

https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/lexical-analysis.html

- As expressões regulares são formadas por metacarateres que definem padrões - obter o casamento entre uma **regex** e um texto.

- **Metacaracteres**
  - São caracteres que tem um significado especial na expressão regular.

Metacaracteres

Meta	Descrição	Exemplo
.	Curinga	Qualquer caractere
[]	Lista	Qualquer caractere incluído no conjunto
	Lista negada	Qualquer caractere não incluído no conjunto
\d	Dígito	O mesmo que [0-9]
\D	Não-digito	O mesmo que 0-9
\s	Caracteres em branco	
\S	Caracteres em diferentes de branco	
\D	Alfanumérico	O mesmo que [a-zA-Z0-9_]
\W	Não-alfanumérico	
\	Escape	Faz com que o caracteres não sejam avaliados na Regex
(...)	Grupo	É usado para criar um agrupamento de expressões
|	OU	casa | bonita – pode ser casa ou bonita

Quantificadores

São tipos de metacaracteres que definem um número permitido de repetições na expressão regular.

Expressão	Descrição	Exemplo
{n}	Exatamente n ocorrências	
{n,m}	No mínimo n ocorrências e no máximo m	
{n,}	No mínimo n ocorrências	
{,n}	No máximo n ocorrências	
?	0 ou 1 ocorrência	car?ro – caro ou carro.
+	1,ou mais ocorrência	ca*ro –carro, carrro, carrrro, nunca será caro.
*	0 ou mais ocorrência	ca*ro – caro, carro, carro, carrrro

Casar

Tem o significado de combinar uma expressão regular com texto.
É quando os metacaracteres especificados na expressão regular correspondem aos caracteres dos textos.

- A regex `\d,\d` casa com `9,1` já `\d,\d` não casa com `91`.

- A regex `\d{5}-\d{3}` é utilizada pra validar CEP. Essa regex casa com os padrões de texto `89900-000` e `87711-000` mas não casa com os padrões `87711-00077` e `89900000`. A regex é formada pelo metacaractere `\d` e o quantificador `{5}`

- A regex `[A-Z]{3}\d{4}` é utilizada para validar a placa de um automóvel e casa com o padrão `ACB1234` mas não casa com o padrão `ACB12345`.

- O FLEX é uma ferramenta que permite gerar analisadores léxicos.

- São capazes de reconhecer padrões léxicos em texto (números, identificadores e palavras-chave de uma determinada linguagem de programação).

- LEX é uma ferramenta Unix para gerar analisadores lexicais sob-demanda.
- – Lesk & Schmidt, Bell Laboratories, 1975.
  - Versão free: FLEX (Fast Lex)
    - http://www.gnu.org/software/flex

Geradores de analisadores léxicos - lex

- Os geradores de analisadores léxicos e automatizam o processo de criação do autômato finito e o processo de reconhecimento de sentenças regulares a partir da especificação de expressões regulares.

- São chamadas de ***lex***

- Flex – http://flex.sourceforge.net/
- JFlex – http://jflex.de/download.html
- Turbo Pascal Lex/Yacc - http://www.musikwissenschaft.uni-mainz.de/~ag/tply/
- Flex++ - http://www.kohsuke.org/flex++bison++/
- CSLex – versão C#, derivada do Jlex - http://www.cybercom.net/~zbrad/DotNet/Lex

Como criar uma especificação – lex?

- O ponto de partida para a criar uma especificação usando a **linguagem lex** é criar uma especificação de expressões regulares que descrevem os itens léxicos que são aceitos.
- Este arquivo possui 3 seções:
  - Declarações
  - Regras de tradução
  - Procedimentos auxiliares

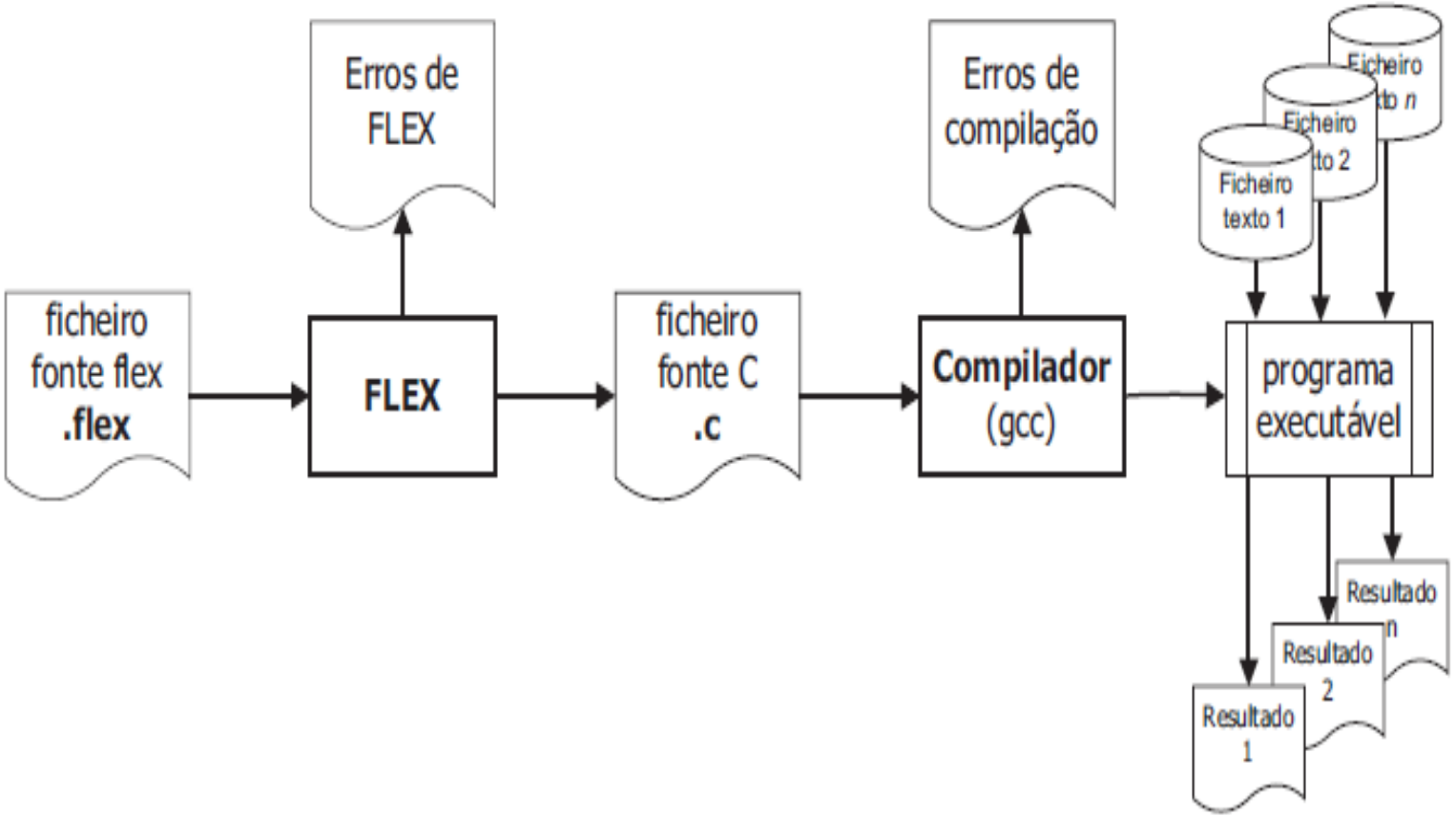
- Declarações: declarações de variáveis que representam definições regulares dos lexemas.

- Regras de Tradução: estão vinculadas as regras que correspondentes a ações em cada expressão regular válida na linguagem.

- Procedimentos Auxiliares: são colocadas as definições de procedimentos necessários para a realização das ações especificadas ou auxiliares ao analisador léxico

FLEX

Ciclo de vida de um programa FLEX



1 - Com base num programa fonte escrito de acordo com a sintaxe do FLEX, o programa FLEX gerará um analisador léxico descrito na linguagem C.

2 - Em caso de existirem erros de codificação, o FLEX gerará uma listagem de erros.

3 - O ficheiro fonte em C terá de ser compilado para a plataforma em utilização (neste caso o GCC).

4 - O resultado final da compilação será um programa executável

5 - Como entrada para o analisador gerado podem ser fornecidos ficheiros de texto ou alternativamente fornecer os dados diretamente pelo standard de entrada

Funcionamento

1. Especificação LEX (arquivo fonte .l)
   – Declarações, Padrões, regras para aplicar ações
2. Compilação LEX
   – Obtenção de um programa C (lex.yy.c)
3. Compilação do programa C
   – Obtenção de um analisador lexical

Definições
%%
Regras
%%
Subrotinas opcionais(código C)

- Lex define várias variáveis globais:
  – yytext e yyleng - Lexema corrente e seu tamanho
  – yylval
    De tipo YYSTYPE, a partir de #defines
    Depende do parser, você escolhe

Passos necessários

- Considere-se a existência de um 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

- 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.

Passos necessários

- 1. FLEX -o<arq_saida.c> <arq_def>.l

- *.l => Arquivos que contêm as definições das unidades léxicas a serem identificadas e *.c => Arquivo do programa “C” que implementa o analisador léxico especificado.

- Principais opções do FLEX:

  -i => Case sensitive (ignora diferença entre maiúscula/minúsculas)

  --version => Exibe a versão atual do programa flex em uso

  -+ => Geração de código de saída em C++

- 2. GCC <arq_saida.c> -o <arq_executavel> -lfl

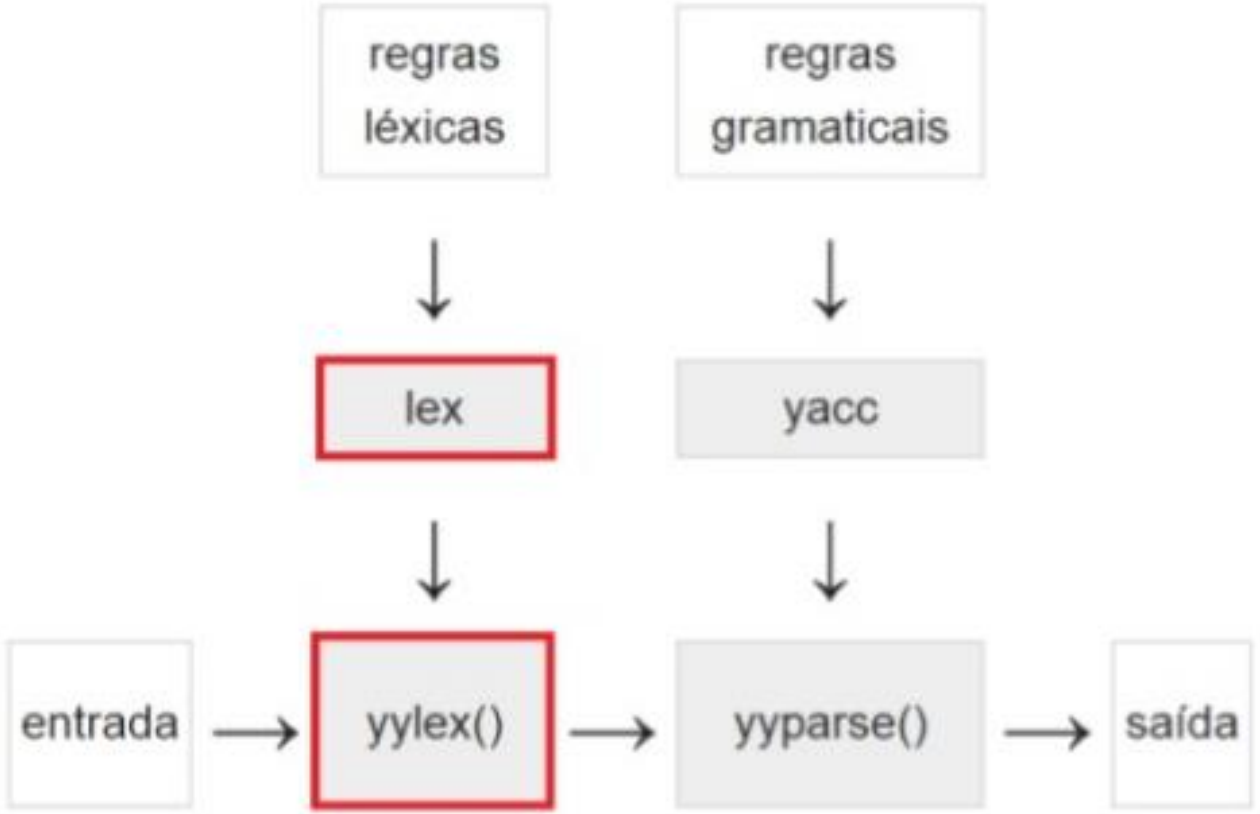
- Observações importantes sobre a geração do programa de análise léxica:

  - É necessário linkar (-l) uma bibliotec (“lib”) do analisador léxico na compilação do código gerado. O FLEX usa a lib “fl” e o LEX usa a lib “l”.

  - O programa gerado pode ser executado, onde usualmente a entrada do texto a ser analisado é feita pela “stdin” (teclado).

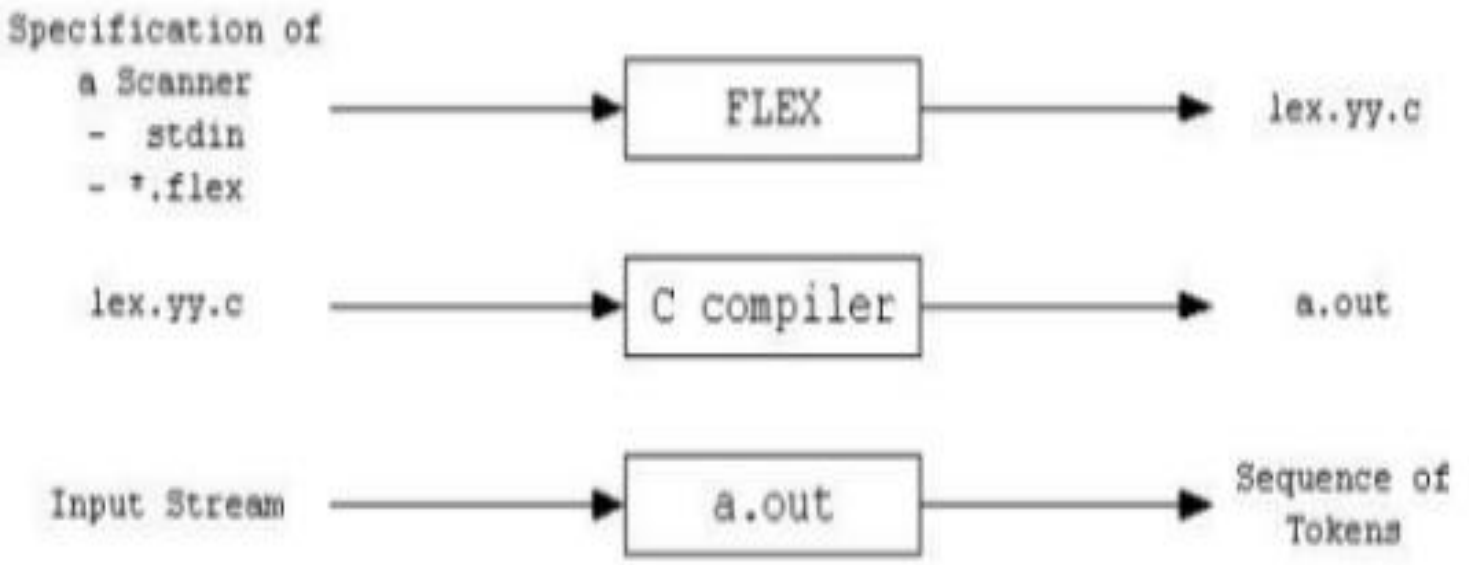
- 3. Executar o programa gerado...

Lex para C



http://pt.slideshare.net/aldisiogm/compiladoranalisador-lxico-com-flex-linguagem-c

Flex para C


Specification of
a Scanner
- stdin
- *.flex
FLEX
lex.yy.c
lex.yy.c
C compiler
a.out
Input Stream
a.out
Sequence of
Tokens

Formato do FLEX

- Formado por três seções:

  1 - declarações,

  2 – regras

  3 - rotinas auxiliares.

  %% - separa as seções

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

Definição da variável numChars

Definições
%%
Regras
%%
Subrotinas opcionais(código C)

Nesta seção são definidas as expressões Regulares e as ações, no caso de qualquer Caracter, será incrementado a variável numChars.
A mudança de linha (representado por “\n”) É contabilizada como um caracter

E posteriormente será impresso.
O valor desta variável poderia ser obtido através da instrução da linguagem C strlen(yytext)

Corpo do programa.
A função yylex() evoca o analisador léxico gerado pelo flex que processará as expressões regulares anteriormente Descritas.

Declarações e expressões regulares

- 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 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

- Expressão regular é uma forma compacta de denotar todos os possíveis strings que compõe uma determinada linguagem

- Nesta parte, podem ser declaradas macros para as expressões regulares, sendo as 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 */
```

```
main ( )
2 {
3 yylex ( ) ;
4 p r i n t f ( "Número de  c a r a c t e r e s  %d\n" ,
    numChars ) ;
5 }
```

Outro exemplo

- Considere o seguinte exemplo, no qual é contabilizado a quantidade de números e de linhas existentes no ficheiro.

- Neste exemplo recorre-se à utilização de uma macro para a definição de algarismo

Parte 1

```
%{
 int qtdNumeros=0, nLinhas=0;
 %}

 ALGARISMO [0−9]
```

Parte 2

```
%%

 / Se a ação for descrita numa só linha
 as chaves podem ser omitidas /

 \n                    nLinhas++;
 {ALGARISMO}+  { printf ( "d %s \n" , yytext ) ; qtdNumeros++;}
 .

%%
```

A variável yytext guarda o texto correspondente ao símbolo terminal (*token*) corrente

Parte 3

```
main ( )
{
 yylex ( ) ;
 p r i n t f ( "#_linhas = %d\n" , nLinhas ) ;
 p r i n t f ( "#_numeros =%d\n" , qtdNumeros ) ;
}
```

Padrões mais usados no FLEX
Expressões Regulares

[0-9] => Reconhece um dígito

[a-zA-Z] => Reconhece uma letra (comum = sem acentos)

[\ \t\n] => Reconhece um espaço em branco ou um tab ou uma nova linha

xxxxx => Reconhece a seqüência de caracteres “xxxxx”

- Símbolos especiais: Exemplo:

+	=> 1 ou mais ocorrências [0-9]+ => Um número
*	=> 0 (nenhuma) ou mais ocorrências [0-9][0-9]* => Um número
?	=> 0 (nenhuma) ou apenas 1 ocorrência -?[0-9]+ => Um número com/sem sinal
\n	=> Reconhece a marca de fim de linha / nova linha
.	=> Aceita um caracter qualquer de entrada
xxx$	=> Reconhece xxx se for seguido de um fim de linha
^xxx	=> Reconhece xxx se este estiver imediatamente após o início de uma linha
[^x]	=> Reconhece qualquer caracter menos “x”
[xyz]	=> Reconhece um dos caracteres “xyz” indicados
[a-z]	=> Reconhece um caracter pertencente ao intervalo de “a-z”
x{n}	=> Reconhece um número exato “n” de ocorrência de “x”
x{n,}	=> Reconhece a ocorrência de no mínimo “n” vezes de “x”
x{n,m}	=> Reconhece a ocorrência de “x” entre no mínimo “n” e no máximo “m” vezes
xx|yy	=> Reconhece a ocorrência de “xx” ou de “yy”
(x|y)	=> Agrupa (sub)expressões regulares
“x”	=> Reconhece exatamente o caracter “x” (usado com caracteres especiais). Ex.: “+”

Exemplos de expressões regulares simples:

- DIGITO [0-9]
- LETRA [a-zA-Z]
- ESPACO [\ \t\n]
- INTEIRO [0-9]+
- INTSIGNED -?[0-9]+
- DECIMAL [0-9]*\.[0-9]+ => aceita .33 / não aceita números sem casas decimais
- INTOUDEC ([0-9]+)|([0-9]*\.[0-9]+)
- IOUDSIGNED -?(([0-9]+)|([0-9]*\.[0-9]+))
- NOMEVAR [a-z][a-z0-9_]* => usando opção “case insensitive”...

Recursos Adicionais

- Acesso ao texto do token: yytext
- Acesso ao tamanho do token: yyleng
- Acesso ao arquivo/dispositivo de entrada:
- yyin => Arquivo de entrada (stdin = teclado)
- input() => Rotina usada na leitura da entrada (pode ser “sobreposta” e customizada)
- unput() => Rotina de apoio usada na leitura de entrada (pode ser adaptada)
- output() => Rotina usada para escrita de saída
- yywrap() => EOF: executa yywrap... 0 = mais dados de entrada / 1 = fim
- yymore() => hyper yymore();
- text printf(“Token: %s\n”,yytext);

A função atoi (array to integer) em C serve para transformar uma string em um integer

Outra opção

- http://jflex.de/download.html

- https://johnidm.gitbooks.io/compiladores-para-humanos/content/part2/building-the-first-lexical-analyzer-with-JFlex.html

Exercícios

1 - Implementar um analisador léxico para uma calculadora simples, capaz de realizar as quatro operações aritméticas básicas.

Entrada válida: número operador número.

Não é necessário tratamento de erros: descarte o que não for válido.

Objetivo do trabalho:

– Explorar as potencialidades do (f)lex e familiarização com o ambiente.

Referências

- Licenciatura em Engenharia Informática – DEI/ISEP - Linguagens de Programação 2006/07 - Ficha 1 - Introdução ao FLEX e expressões regulares

- Material prof. Osório – Unisinos – Compiladores