Compiladores

Ficha prática 2 - Lex

1. A ferramenta *lex*, tipicamente disponível em sistemas *Unix*, permite criar um analisador lexical extremamente poderoso. Para utilizar o *lex*, devemos criar um ficheiro que descreve as expressões regulares correspondentes aos *tokens* (ou padrões) pretendidos. A partir desta especificação, o *lex* gera então código "C", que pode ser utilizado como um programa independente (como verá em breve) ou associado a outras ferramentas (como o YACC).

O *lex* espera um ficheiro no seguinte formato:

```
...definições...
%%
...regras...
%%
...subrotinas...
```

Por agora, vamos concentrar-nos na segunda parte (regras), onde se pode descrever um conjunto de padrões, bem como definir acções associadas a cada padrão. Cada regra é dividida em duas partes distintas. A primeira parte, sempre do lado esquerdo, descreverá o padrão a ser identificado; a segunda parte, separada de pelo menos um espaço (ou "tab"), corresponde a um bloco de código em linguagem "C", que deverá ser executado sempre que encontrado o padrão especificado. Os padrões (ou tokens) são descritos em expressões regulares. Na seguinte tabela, mostramos a linguagem utilizada na descrição destas expressões regulares:

Expressão regular	**Significado**	**Exemplos**
x	Caracter "x"	a, aa, abc, 123, a1...
"x" ou \x	Caracter "x", mesmo que seja metacaracter	".", "[", \", "\n", \\...
[xy] ou x|y	Caracter x ou y	[ab], [a|b], [123], ...
[x-y]	Caracteres entre x e y	[a-z], [0-9], ...
[^x]	Todos os caracteres excepto x	[^a], [^a"."2], [^\n],...
.	Qualquer caracter excepto "\n"	
x?	Caracter x opcional	ab?c
( )	Associação de expressões regulares	(ab|cd), ([ab][123])
x*	X repetido 0 ou mais vezes	a*, [ab]*, (a|b)*,...
x+	X repetido 1 ou mais vezes	a+, ab+c, [ab]+, ...
x{n}	Repetição de x *n* vezes	a{3}b+, [0-9]{4}
x{n, m}	Repetição de x *n* a *m* vezes	a{3,6}, [0-9]{4-6}
^x	O caracter "x" deve aparecer no início da linha	
x$	O caracter "x" deve aparecer no fim da linha	

No ficheiro de especificações, deve então descrever todas as expressões regulares que o analisador lexical deve suportar. Caso haja algum padrão não considerado pelo analisador, este simplesmente copia para a saída o padrão encontrado.

Em baixo, é mostrada a especificação de um analisador que imprime a palavra "Inteiro" de cada vez que encontrar um número inteiro no input:

```
%%
[0-9]+              {printf("Inteiro");}
```

Se corressemos este analisador, verificariamos que substituiria simplesmente todos os inteiros pela palavra "Inteiro" (deixando tudo o resto igual). Elaborando agora um pouco mais, podemos acrescentar uma regra para os números reais:

```
[0-9]+"."[0-9]+          {printf("Real");}
```

Na zona de definições (antes das regras), podemos criar uma definição para "numero", que tornará mais legível o ficheiro (passaremos a usar *{numero}* em vez de *[0-9]+*). Na zona de "subrotinas", é também normal colocar as funções *int main()* e *yywrap()*. Assim, o nosso primeiro ficheiro de exemplo terá o seguinte:

```
numero  [0-9]+
%%
{numero}"."{numero}              {printf("Real");}
{numero}                        {printf("Inteiro");}
%%
int main()
{
yylex();   /* chama o entry point */
return 0;
}

int yywrap()
{
return 1;  /* Esta função será chamada quando o input
           chegar ao fim. Devolve 1 se não houver mais
           nada a processar.*/
}
```

Para testar este exemplo, comece por fazer o download do ficheiro *exemplo1.l* (dentro de "ficha1.zip" na secção "Material de Apoio" da página da cadeira). Em linux (ou unix), corra o seguinte comando:

```
lex exemplo1.l
```

De seguida, deverá compilar o código c gerado (*lex.yy.c*), com o seguinte comando:

```
cc –o analisador1 lex.yy.c –ll
```

Poderá agora experimentar o programa gerado (correndo o *analisador1* - no unix ./analisador1 -, que acabou de criar). Coloque números inteiros e reais, experimente com outro tipo de tokens.

a) Altere o seu programa para distinguir entre números positivos e negativos. Note que os símbolos – e + são operadores das expressões regulares, pelo que deverá usar aspas (“”). Não se esqueça de testar.

b) Altere o seu programa para emitir a mensagem “token!” quando for detectado um dos seguintes padrões:

i) `abc`

ii) `abbb...c` (começa com a, tem pelo menos um b e termina em c)

iii) `bc, abcbc, abcbcbc...` (pode começar ou não com a, tem pelo menos uma ocorrência de bc)

c) Acrescente regras gramaticais que reconheçam números tal como definidos na linguagem “C” (colocamos aqui em itálico a parte indicadora do tipo): 123, 123*l*, 123*u*, 123*ul*, 12.3, 123*e-1*, 12.3*e+1*, 123*f*. Indique, para cada um, qual o tipo correspondente (*integer, long, unsigned int, unsigned long, float, floating point, floating point, float*, respectivamente). Permita também a utilização da vírgula (“,”) para além vez do ponto (“.”) nos números com parte fraccionária.

e) Teste o seu programa com o ficheiro “test1.txt”, disponível também em “ficha1.zip”.

2. Escreva regras gramaticais no *lex* que reconheçam as strings tal como definidas em Java. Repare que são rodeadas de aspas (“”). Não necessita considerar a inclusão de aspas dentro da expressão (com \”). Caso pretenda, associe acções às regras.

Mais dicas sobre o lex:

1 – Em caso de haver um padrão que seja reconhecido por mais de uma expressão regular, o lex resolve assim a ambiguidade (por ordem de preferência):

- O analisador tenta escolher a regra que identifica um padrão maior
- O analisador escolhe a regra que aparece primeiro

2 – Quando não queremos que o analisador simplesmente reproduza tudo o que não é reconhecido, colocamos uma última regra semelhante à seguinte:

```
.                    {;}
```

Esta regra diz que “para todo o tipo de caracteres, exceptuando o \n, não se deve fazer nada”. Logicamente, tem que ser a ultima regra, para evitar prejudicar as regras anteriores.

Bibliografia recomendada:
. Anexo A de *Processadores de Linguagens*. Rui Gustavo Crespo. IST Press. 1998
. *A Compact Guide to Lex & Yacc*. T. Niemann.
http://epaperpress.com/lexandyacc/epaperpress .
. Manual do lex/flex em Unix (comando “man lex” na shell)