Ficha prática 1

Compiladores

2018/2019

Lex

1 Introdução

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.

1. A primeira parte, sempre do lado esquerdo, descreverá o padrão a ser identificado. Os padrões (ou tokens) são descritos em expressões regulares. Na tabela seguinte, 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], [1,2,3], ...
[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 facultativo	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 de n a m vezes	a{3,6}b+, [0-9]{4,6}, ...
^x	O caracter x deve aparecer no início da linha	
x$	O caracter x deve aparecer no fim da linha	

2. 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.

No ficheiro de uma especificação, 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 corrêssemos este analisador, verificaríamos 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 número, 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 conteúdo:

```
numero [0-9]+
%%
{numero}            {printf("Inteiro");}
{numero}"."{numero} {printf("Real");}
%%
int main()
{
    yylex();        /* chama o autómato que faz a análise lexical */
    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 `p-01-lex.zip`). Em linux (ou unix), corra o seguinte comando:

```
lex exemplo1.l
```

De seguida, deverá compilar o código C gerado (no ficheiro `lex.yy.c`), com o seguinte comando:

```
clang-3.8 -o analisador1 lex.yy.c
```

Poderá agora experimentar o programa gerado correndo o executável produzido, `analisador1`. Em linux (ou unix), corra o seguinte comando:

```
./analisador1
```

Coloque números inteiros e reais, e experimente com outro tipo de tokens.

2 Exercícios

Exercício 1 *Altere o programa a que se chegou na secção anterior 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.*

1. *Altere o seu programa para emitir a mensagem* `token!` *quando for detectado um dos seguintes padrões:*

   (a) `abc`

(b) `abbb...c` *(palavras que começam por* `a`*, têm um ou mais* `b` *e terminam em* `c`*)*

(c) `bc`, `abcbc`, `abcbcbc...` *(palavras que podem ou não começar por* `a`*, e têm uma ou mais ocorrências de* `bc`*)*

2. *Acrescente regras que reconheçam números tal como são 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 do ponto (*`.`*) nos números com parte fraccionária.*

3. *Teste o seu programa com o ficheiro* `test1.txt` *que acompanha esta ficha.*

□

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

3 Ainda sobre o lex

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):

1. O analisador tenta escolher a regra que identifica um padrão maior.
2. O analisador escolhe a regra que aparece primeiro.

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 todos os caracteres, exceptuando o `\n`, não se deve fazer nada”. Logicamente, tem que ser a última regra, para evitar prejudicar as regras anteriores.

O *lex* permite o acesso a um conjunto de variáveis e funções pré-definidas, de entre as quais destacamos:

Nome	Descrição
`int yylex(void)`	Chamada do analisador lexical
`char *yytext`	Ponteiro para a string detectada (o token)
`yyleng`	Comprimento da string detectada (do token)
`yylval`	Valor associado ao token
`int yywrap(void)`	Wrapup, devolve 1 se acabou input, 0 caso contrário
`int yylineno`	Número de linha do ficheiro de entrada
`FILE *yyin`	Ponteiro para ficheiro de entrada
`FILE *yyout`	Ponteiro para ficheiro de saida
`BEGIN`	Macro para condição inicial no switch
`ECHO`	Macro para repetir na saída a string detectada

Por agora, vamos utilizar o ponteiro para a string (`yytext`). Com esta variável, temos acesso directo ao token (ou padrão), e portanto são muito maiores as potencialidades do *lex*. Por exemplo, se, em vez de escrevermos simplesmente `Inteiro`, quiséssemos substituir o número pelo seu equivalente em hexadecimal, a regra seria:

```
%%
[0-9]+              printf("%X", atoi(yytext));
```

Lembramos que `"%X"` imprime um inteiro em hexadecimal (na função `printf`, de C) e que a função `atoi(char*)` devolve o valor inteiro contido numa string.

Outra possibilidade no *lex* é a declaração de variáveis na parte de definições (desde que entre `%{` e `%}`). No código seguinte, podemos ver um exemplo da sua aplicação:

```
%{
    int numints=0;     /* Contador de inteiros */
%}
%%
[0-9]+              { printf(\%X", atoi(yytext)); /* Passa a hex */
                      numints++;} /* Soma um */
[0-9]+"."[0-9]+     ; /* Real, não faz nada */
.                   ; /* Tudo o resto excepto new line */
\n                  ; /* new line */
%%
int main()
{
    yylex();
    printf("Numero de inteiros=%d", numints); /* Mostra resultado */
    return 0;
}
int yywrap()
```

```
{
    return 1;
}
```

4 Exercícios

Exercício 3 *Construa um analisador lexical que transforme, no texto de input, todas as minúsculas em maiiúsculas (e mantenha tudo o resto igual).* □

Exercício 4 *Construa um analisador lexical que extraia de um documento de texto todos os endereços de email nele contidos. Assuma que os caracteres não permitidos num endereço de email são* `\t`*,* `\n`*, espaço,* `;`*,* `,` *e* `"`*. Lembre-se também que não pode haver dois* `@` *nem dois* `.` *seguidos.*

Este analisador deve apenas produzir a lista de endereços, separados de virgula e espaço. Ou seja, prontos para serem copiados directamente para um cliente de email.

Por exemplo, para o texto seguinte:

```
O endereço do João Fonseca é jfonseca@gmail.com e o do Carlos Meireles
é meiras@hotmail.com. Para os outros, estão na lista abaixo:
- mariliaf@student.deeei.uq.tp; mfons@kkks.us
- gilaroi@yaaahooo.com; gil eanes@grlah.jh
```

o resultado seria

```
jfonseca@gmail.com, meiras@hotmail.com, mariliaf@student.deeei.uq.tp,
 mfons@kkks.us, gilaroi@yaaahooo.com, eanes@grlah.jh
```

□

Exercício 5 *Dado um ficheiro .srt (de legendas para Divx), construa um analisador lexical para adiantar/atrasar essas legendas de acordo com um valor de x segundos. Exemplo do formato de um ficheiro .srt:*

```
1
00:00:45,000 --> 00:00:50,400
Hello
2
00:00:50,300 --> 00:00:58,200
Goodbye
```

Neste exemplo serão geradas duas legendas. A primeira, será apresentada no filme aos 45 segundos durante cerca de 5 segundos. A segunda, será apresentada no filme aos 50 segundos durante cerca de 3 segundos.

□

Referências

[1] Anexo A de Processadores de Linguagens. Rui Gustavo Crespo. IST Press. 1998

[2] A Compact Guide to Lex & Yacc. T. Niemann.
`http://epaperpress.com/lexandyacc/epaperpress`

[3] Manual do lex/flex em Unix (comando `man lex` na shell)