Compiladores

Análise Léxica (Parte 2)

Introdução

Este material contém a continuação das notas de aula sobre *Análise Léxica*, da disciplina *Compiladores*. O objetivo é aplicar os conhecimentos obtidos na aula anterior, cujo texto está disponível no documento Análise Léxica (Parte 1), para criar analisadores léxicos com a ferramenta `flex`. Recomenda-se a instalação do `flex` para possibilitar a reprodução dos exemplos e a realização dos exercícios.

Exercícios de Revisão

1) Como seriam as regras de produção de uma Gramática Livre de Contexto – GLC que gera números inteiros?
2) Como seriam as regras de produção de uma Gramática Regular – GR que gera números inteiros?
3) Na expressão `x = (y + tmp) * 30;`
    a) Quais são os lexemas e suas respectivas categorias?
    b) Quais são os tokens?
4) (POSCOMP QUESTÃO 68) A tarefa principal de um analisador léxico consiste em ler os caracteres da entrada do programa-fonte, agrupá-los em lexemas e gerar uma sequência de *tokens* que será enviada ao analisador sintático. Sobre o analisador léxico, analise as assertivas abaixo:

I. Além da identificação de lexemas, outras tarefas podem ser realizadas por esse analisador, tais como: remoção de comentários e espaços em branco e a associação de

mensagens de erros às linhas do programa-fonte.

II. *Token* é a unidade básica do texto-fonte. Pode ser representado por três informações: a classe do *token*, que representa o tipo do *token* reconhecido, o valor do *token*, que é o texto do lexema reconhecido e a posição que indica o local do texto-fonte (linha e coluna) onde ocorreu o *token*.

III. Expressões regulares e geradores de analisadores léxicos são notações utilizadas para especificar os padrões de lexemas.

IV. Na análise léxica, uma representação intermediária do tipo árvore é criada. Esta apresenta a estrutura gramatical da sequência de *tokens*.

Quais estão corretas?

a) Apenas I.
b) Apenas II.
c) Apenas IV.
d) Apenas I e II.
e) Apenas III e IV.

Respostas

1) S → ABC
   A → + | - | ∅
   B → dígito
   C → dígito C | ∅
   dígito → 0|1|2|3|4|5|6|7|8|9

2) número → [+-]? dígito+
   dígito → 0|1|2|3|4|5|6|7|8|9

3)

a)

Lexema	**Categoria**
x	identificador
=	operador
(	delimitador

y	identificador
+	operador
tmp	identificador
)	delimitador
*	operador
30	constante
;	delimitador

b) **Primeira versão:**

Lexema	**Token**
x	<identificador, 1>
=	<=, >
(	<(, >
y	<identificador, 2>
+	<+, >
tmp	<identificador, 3>
)	<), >
*	<*, >
30	<constante, 30>
;	<;, >

Segunda versão:

Lexema	**Token**
x	<identificador, 1>
=	<operador, =>
(	<separador, (>
y	<identificador, 2>
+	<operador, +>
tmp	<identificador, 3>
)	<separador,)>
*	<operador, *>
30	<constante, 30>
;	<separador, ;>

4) A.

I. correta.

II. incorreta, a unidade básica do código-fonte é o caractere. Lexemas são agrupamentos de caracteres. O *scanner* verifica os caracteres. A implementação envolve ler caracteres – `getchar()`.

III. analisadores léxicos são ferramentas e não notações.

IV. os *tokens* produzidos são enviados ao analisador sintático, não é tarefa do analisador léxico gerar a representação em árvore.

Flex

A Parte 1 introduziu a implementação de analisadores léxicos, citando as formas manual e via ferramentas. Nesta seção, apresenta-se a criação de analisadores léxicos com `lex`/`flex` (baseado principalmente em Ricarte e em Du Bois).

Fast LEXical analyzer generator, ou `flex`, é uma ferramenta geradora de *scanners*. Para usá-la, basta identificar o vocabulário de uma linguagem (ou seja, as palavras que devem ser reconhecidas), escrever uma especificação de padrões usando expressões regulares e o scanner será gerado. A Figura 1 resume a forma com que um scanner é gerado com a ferramenta `flex`.

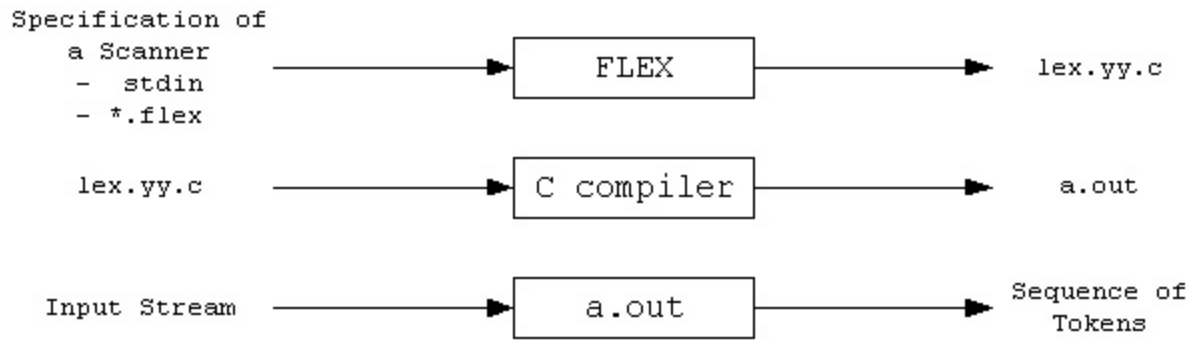

Figura 1. Forma geral de uso do `flex`. Fonte: GAO, s.d.

Como mostra a Figura 1, o primeiro passo é criar um arquivo com a especificação das expressões regulares que descrevem os lexemas que são aceitos. Este arquivo é composto por até três seções: **definições**, **regras** e **código do usuário**.

As seções são separadas pelos símbolos %%. Assim, o arquivo de especificação terá sempre o seguinte formato:

```
[DEFINIÇÕES]
%%
REGRAS
%%
CÓDIGOS++
```

A ordem das seções deve ser seguida rigorosamente. A seção de definições é opcional, pois, se desejável, é possível criar diretamente as regras sem as definições.

Em geral, costuma-se criar arquivos de especificação com a extensão `.l`. Exemplo: `linguagem.l`. Também pode-se utilizar a extensão `.lex`.

Pensemos em uma situação hipotética de criar um analisador léxico para verificar se todos os caracteres do programa fazem parte de um subconjunto do alfabeto de uma linguagem, devendo reconhecer somente três estruturas da linguagem: números inteiros, números reais e variáveis.

O primeiro passo deve ser a identificação dos símbolos permitidos, ou seja, os caracteres que podem fazer parte dos lexemas. Aqui podem se enquadrar:

- sinais
- dígitos
- letras maiúsculas
- letras minúsculas
- símbolos como underline, ponto, vírgula

Após a identificação dos símbolos, costuma-se categorizar os termos possíveis. Neste exemplo, isso já foi dado no enunciado. Os termos podem fazer parte de três categorias: números inteiros, números reais e variáveis. Em outras situações poderia haver uma categorização genérica para números, englobando inteiros e decimais.

O próximo passo pode ser a criação dos autômatos que devem reconhecer essas três situações. Como eles seriam?

Depois disso, basta criar o arquivo `.lex`. Neste caso, uma versão preliminar ficaria da seguinte forma:

```
DIGITO      [0-9]
LETRA       [a-zA-Z]
UNDERLINE   "_"
PONTO       "."

%%

{DIGITO}+({DIGITO}+)?                       { printf("número inteiro:  %s\n", yytext); }
({DIGITO})?+{PONTO}({DIGITO}+)              { printf("número real:  %s\n", yytext); }
{LETRA}+({UNDERLINE}*{DIGITO}*{LETRA}*)*    { printf("variável: %s\n", yytext); }

%%

int yywrap();

int main(void) {
  yylex();
}

int yywrap() {
 return 1;
}
```

O arquivo pode ser salvo como `primeiro-exemplo.l`.

Agora, a partir da especificação criada, é necessário gerar o analisador léxico com o `flex`. Ao gerar o analisador, o arquivo `lex.yy.c` será criado. Ele deverá ser compilado de forma específica, pois procederá a geração do analisador em um arquivo executável. Para isso, é necessário seguir os seguintes passos:

Criação do arquivo `lex.yy.c`:

> `flex reconhecedor-de-numeros-e-variaveis.l`

Caso ocorra algum erro, um aviso será exibido. Para exemplificar, pode-se mudar a expressão

UNDERLINE por UNDERLIN (remover o último “E”) na seção de definições e executar o comando acima.

Geração do arquivo executável do *scanner*: É necessário compilar o arquivo `lex.yy.c` com a diretiva `-lfl`:

> `gcc lex.yy.c -lfl`

O arquivo executável (`a.out`) foi gerado. Basta executá-lo (`./a.out`) e o *scanner* estará em funcionamento. Alguns valores que podem ser testados no *scanner* gerado:

11

11.11

.11

abc_

abcd_e_f

var_n0m3_estr4nh0

Note que a forma `{DIGITO}` é substituída no padrão pela expansão da definição `DIGITO`. Note também o uso de alguns caracteres como ?, -, + e *.

- ?: a expressão pode ocorrer zero ou uma vez(es).
- +: a expressão ocorre pelo menos uma vez.
- *: a expressão pode ocorrer zero ou muitas vezes.
- -: pode conter vários significados, mas no contexto do nosso exemplo ele delimita um intervalo.
- |: operador "ou" lógico.

No exemplo `primeiro-exemplo.l`, utilizamos a expressão "`{DIGITO}+({DIGITO}+)?`" apenas para demonstrar o uso do `?`. Nesse caso, a segunda parte da expressão é totalmente necessária. Utilizar "`{DIGITO}+`" é suficiente.

Como já dito, o arquivo (`f`)`lex` pode conter código C também na seção de definições. Nela é possível colocar definições C de objetos acessados por ações na seção de regras ou por rotinas com ligação externa. Por exemplo, ao usar `lex` com `yacc` (um gerador de analisador sintático), é necessário incluir o arquivo `y.tab.h`, que pode conter diretivas `#define` para

nomes de *token*. Por exemplo:

```
%{
#include "y.tab.h"
extern int valorDoToken;
int numerosDeLinha;
%}
```

Porém, para começarmos a elaborar construções mais complexas, tal como ocorre com as linguagens de programação que utilizamos, precisamos prever mais definições e regras. Por exemplo, se tentarmos entrar com o valor `x = 11`, teremos como retorno:

```
variável: x
 = número inteiro:  11
```

Vemos que os espaços e o sinal de igual não foram compreendidos, pois as expressões regulares definidas não contemplam regras com estes símbolos e eles também não estão definidos. Quando um símbolo não é definido, (`f`)`lex` apenas o imprime de volta. Assim, o operador de = é ignorado e só é repetido pelo `flex`.

Vamos alterar o código para que, apenas para fins didáticos, ele exiba a mensagem no formato: `Lexema encontrado: <termo>. É um(a) <categoria>.`

Além disso, vamos acrescentar outras regras, por exemplo, para os lexemas igualdade, ponto-e-vírgula ou outros operadores. Salve o novo arquivo como `reconhecedor-de-numeros-e-variaveis.l`.

O arquivo `reconhecedor-de-numeros-e-variaveis.l` traz um exemplo com as alterações mencionadas. Execute o *scanner* e teste-o! Agora, ao entrar com `x = 11`, o *scanner* retorna:

```
Lexema encontrado: x. É uma variável.
Lexema encontrado: =. É uma atribuição.
Lexema encontrado: 11. É um número inteiro.
```

Procure aprimorar o reconhecedor de números e variáveis! É possível, por exemplo, impedir e

ignorar expressões como `x = = 10`, `x == == 10`, `x === = 10` e outras.

Exemplo: `{IGUALDADE}+{ESPACO}+{IGUALDADE}+`

Ver também os seguintes códigos:

- reconhecedor-de-numeros-e-variaveis-v2.l.
- reconhecedor-de-numeros-e-variaveis-v3.l.
- reconhecedor-de-numeros-e-variaveis-v4.l.

Conclusão

A Análise Léxica é realizada pelo scanner, que pode ser criado manualmente ou por meio de ferramentas. `flex` é uma ferramenta de linha de comando que permite criar *scanners* com facilidade, a partir de uma especificação com expressões regulares. Exemplos práticos mostraram como gerar um arquivo de especificação e compilar o *scanner*. Por fim, abordou-se a possibilidade de aprimorar as regras de reconhecimento, incluindo símbolos e operadores adicionais para melhorar a precisão do analisador léxico.

Exercícios

1) Utilizando flex, elabore um analisador léxico para cada caso:
    a) que reconheça expressões no formato binário (iniciando com 0b) e no formato hexadecimal (iniciando com 0x).
    b) que reconheça strings que representem placas de veículos brasileiros. Para isso, considere os padrões antigo e novo. Faça com que o analisador reconheça as entradas com espaço na posição em que ele pode ser esperado. Por exemplo, as placas ABC1234 e ABC 1234 devem ser reconhecidas.
    c) que reconheça as palavras reservadas da linguagem Python.
    d) que reconheça as matrículas de 2020 dos alunos do IFC.

2) Os exemplos criados ainda são bastante simples. Além disso, os itens léxicos costumam ser divididos em categorias, o que é especialmente válido para linguagens de

programação. Tomando como base a classificação de itens léxicos apresentada em aulas anteriores, utilize o (f)lex para criar um *scanner* que reconheça itens léxicos válidos. A classificação apresentada continha palavras reservadas, identificadores, constantes, operadores, textos e delimitadores (ou separadores). Salve-as em um arquivo `lexico-seu-nome.l`.

Alguns dos testes que serão feitos:

```
> +.1 + (+.1)
> int x = 0; while (x++ > 10) { printf("Vou passar em Compiladores!\n"); }
> unsigned int minhaNota = 5; if (nota < 6) { printf("REPROVADO"); }
> int main() { printf("Hello, world!\n"); return 0; }
```

(nota extra para quem resolver no `flex` e em outra linguagem de programação -- C ou Python)

Observação 1: (f)lex reconhece `[[:space:]]` como todo tipo de espaços (envolve `\t`, `\n`) e isto pode ser utilizado em seus códigos.

Observação 2: a ordem das regras **faz** diferença! Ou seja, se uma palavra for igualmente satisfeita por duas regras, valerá a primeira. Por isso, a regra de palavras reservadas precisa vir antes da regra que reconhece identificadores.

Observação 3: por padrão, `flex` retorna exatamente o texto digitado se ele não for satisfeito por alguma regra. Isso pode ser modificado criando a regra "." (ver exemplo reconhecedor-de-numeros-e-variaveis-v4.l). Com ela podemos ignorar todas as entradas não satisfeitas, não obtendo retorno.

(Algumas) Respostas Válidas

1)

a)

```
%%
0b[01]+                         { printf("binário:  %s\n", yytext); }
0x[0-9A-Fa-f]+                  { printf("hexa:  %s\n", yytext); }
```

```
%%

int yywrap();
int main(void) {
  yylex();
}
int yywrap() {
 return 1;
}
```

b)

```
DIGITO        [0-9]
LETRA         [a-zA-Z]
LETRAS        {LETRA}{3}
DIGITOS       {DIGITO}{2}
DL            {DIGITO}{LETRA}
DLDD          {DL}{DIGITOS}

%%

({LETRAS}{DIGITO}{4})|({LETRAS}"            "{DIGITO}{4})|({LETRAS}"
"{DLDD})|({LETRAS}{DLDD})    {    printf("A    matrícula    %s    foi
reconhecida.\n", yytext); }

%%

int yywrap();
int main(void) {
  yylex();
}
int yywrap() {
 return 1;
```

```
}
```

c)

```
KEYWORD
"and"|"as"|"assert"|"async"|"await"|"break"|"class"|"continue"|"def"|
"del"|"elif"|"else"|"except"|"False"|"finally"|"for"|"from"|"global"|
"import"|"if"|"in"|"is"|"lambda"|"nonlocal"|"None"|"not"|"or"|"pass"|
"raise"|"return"|"True"|"try"|"while"|"with"|"yield"

%%
{KEYWORD} { printf("Palavra reservada em Python: %s.\n", yytext); }
{KEYWORD}+|[a-zA-Z0-9!@#$%^&*()_+\-=\[\]{};':"\\|,.<>\/?]+|" "|"\n"

%%
int yywrap();
int main(void) {
  yylex();
}
int yywrap() {
 return 1;
}
```

d)

```
ANO   2020
DIGITO  [0-9]

%%

{ANO}{DIGITO}{6} { printf("Matrícula válida: %s.\n", yytext); }
{DIGITO}*|{ANO}{DIGITO}{7,}|[[:space:]]

%%
```

```
int yywrap();
int main(void) {
  yylex();
}
int yywrap() {
 return 1;
}
```

Referências

DU BOIS, André Rauber. Notas de Aula sobre Compiladores. 18 ago. 2011.

RICARTE, Ivan Luiz Marques. 2003. Compiladores. Especificação das sentenças regulares. Disponível em: https://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node52.html. Acesso em: 22 mai. 2024.