Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Disciplina: Estrutura de Dados II – BCC 203
Aluno: Thiago Oliveira de Santana
Matrícula: 15.1.4313

ANALISADOR LÉXICO

Sabemos que o processo da análise léxica consiste em ler os caracteres de entrada e agrupa-los em conjuntos que são chamados de tokens, sendo estes, os identificadores e símbolos especiais de uma linguagem de computação. O compilador tem no seu analisador léxico o primeiro componente a entrar em contato com o código fonte, onde o seu papel basicamente é reconhecer os tokens existentes no código e passar essa informação para o parser.

Tendo isso em mente, foi implementado um analisador léxico para a linguagem de programação apresentada por Torben em seu livro de compiladores, no capítulo 4 (gramática 4.1). Para a implementação utilizou-se um gerador de analisador léxico escrito em Java, conhecido como **JFlex**.

1.0 - Linguagem

As palavras pertencentes à linguagem descritas pela gramática são: literais, identificadores, sinais de pontuação, operadores e palavras chaves. A linguagem possui também características como comentários em linha e bloco(podendo ser aninhado) bem como delimitadores de espaço.

1.1 - Comentários

Os comentários de linha, bloco e delimitadores de espaço ao serem examinados pelo analisador léxico são desconsiderados pelo fato destes, assim como nas linguagens de programação conhecidas, não serem importantes para o processo de decodificação realizada pelo compilador. O comentário de linha pode ser realizado utilizando **%** e o comentário de bloco utilizando **{% %}.** Já os delimitadores de espaço podem ser efetuados com **\t\r\n.** A seguir, veremos exemplos dessa categoria.

Exemplos: 1 - “ Print\n”;
2 - “ % Comentário em linha “;
3 - “ {% Comentário em bloco %}”.

1.2 - Literais

É sabido que os literais podem ser classificados como sequência de caracteres que representa uma constante, como um número inteiro, um número em ponto flutuante, um caracter, uma string, um valor verdade (verdadeiro ou falso), etc. Desta forma, os literais da linguagem em questão pertencem ao tipo inteiro**,** podendo ser classificadas

pelo analisador léxico como um token **LITERAL_INT.** Em seguida, contemplaremos alguns exemplos desse grupo.

Exemplos: 1 - “26342 ”;
2 - “0”;
3 - “100000”.

1.3 - Literais

Podemos compreender que os identificadores são palavras utilizadas para nomear entidades do programa, como funções, métodos, classes, módulos,etc. Desta maneira, o analisador léxico implementado classifica todas as palavras que pertencem ao conjunto **[a-zA-Z][a-zA-Z0-9_]*** como um token **ID.** È possível notar que as palavras pertencentes a esse conjunto são todas aquelas que iniciam com qualquer letra do alfabeto, podendo ser maiúsculas ou minúsculas, seguidas de qualquer letra ou número. A seguir, contemplaremos alguns exemplos dessa categoria.

Exemplos: 1 - “Nome”;
2 - “João09”;
3 - “T_Oliveira”.

1.4 – Sinais de Pontuação

É notório que os sinais de pontuação são sequências de caracteres que auxiliam na construção das estruturas do programa, como por exemplo , servindo de separador de expressões em uma lista de expressões.Sendo assim, os sinais de pontuação pertencentes a linguagem são “ **(** “, “ **)** ” e “ **,** “, na qual são classificados pelo analisador como sendo tokens **T_PARENTESEL**, **T_PARENTESER**, **T_VIRGULA**. Em seguida, veremos alguns exemplos dessa classe.

Exemplos: 1 - “ if(num == 0) ”;
2 - “ num1, num2”;
3 – “ (x, y, z)”.

1.5 - Operadores

Os operadores que fazem parte da linguagem são da forma “**+**” e “**=**” e que ao serem analisados, serão classificadas como sendo tokens **T_MAIS** e **T_IGUAL**, respectivamente. Em seguida, podemos ver alguns exemplos desse grupo.

Exemplos: 1 - “ num1 = num2 ”;
2 - “ soma = num1+num2”;
3 - “ +=”.

1.6 – Palavras-chaves

Sabemos que as palavras-chaves são usadas para expressar estruturas da linguagem, como comandos condicionais, comandos de repetição, etc. Geralmente são reservadas, não podendo ser utilizadas como identificadores. Portanto, as palavras reservadas desta linguagem são: “**bool”**, “**int**”, “**if**”, “**then**”, “**else**”, “**let**” e “**in**”. Ao serem examinadas pelo analisador léxico, respectivamente, serão gerados tokens **BOOL**, **INT** , **IF**, **THEN**, **ELSE**, **LET** e **IN**. Em seguida, contemplaremos alguns exemplos dessa categoria.

Exemplos:

1 - “let id = Exp in Exp”; .

2 - “if Exp then Exp else Exp”;

2.0 – Testes

Para cada categoria descrita anteriormente, foram realizados testes automatizados a fim de averiguar a efetividade do analisador léxico. A seguir, contemplaremos alguns exemplos de testes.

```
// Comentário de Linha
trun("% a line comment\n", "2:1-2:1 EOF");

// Comentário de Bloco
trun("{% a block comment %}", "1:22-1:22 EOF");
trun("{% outer {% inner %} outer %}", "1:30-1:30 EOF");

// Pontuação
trun("=", "1:1-1:2 T_IGUAL", "1:2-1:2 EOF");
trun(",", "1:1-1:2 T_VIRGULA", "1:2-1:2 EOF");

// Operador
trun("+", "1:1-1:2 T_MAIS", "1:2-1:2 EOF");
// Literais inteiros
trun("26342", "1:1-1:6 LITERAL_INT(26342)", "1:6-1:6 EOF");
trun("0", "1:1-1:2 LITERAL_INT(0)", "1:2-1:2 EOF");
// Palavras Reservadas
trun("bool", "1:1-1:5 BOOL", "1:5-1:5 EOF");
trun("int", "1:1-1:4 INT", "1:4-1:4 EOF");
trun("if", "1:1-1:3 IF", "1:3-1:3 EOF");
// Identificadores
trun("nome", "1:1-1:5 ID(nome)", "1:5-1:5 EOF");
trun("camelCase", "1:1-1:10 ID(camelCase)", "1:10-1:10 EOF"
```

4.0 Analisador sintático descendente recursivo

Para a construção do analisador sintático descendente recursivo, foram necessários alguns devidos tratamentos, pelo fato da gramática ser ambígua. Sendo assim, houve a remoção de ambiguidade, eliminação de recursão à esquerda e fatoração de algumas regras. A seguir será apresentada a gramática livre de contexto resultante, após esses processos citados anteriormente.

1. *S → Program $*

2. *Program → Funs* — **programa**

3. *Funs → Fun Funs´*
4. *Funs´ →* — **lista de funções**
5. *Funs´ → Funs*
6. *Fun → TypeId (TypeIds) = Exp* — **declaração de funções**

7. *TypeId → int id* — **tipo inteiro**
8. *TypeId → bool id* — **tipo booleano**
9. *TypeIds → TypeId TypeIds´*
10. *TypeIds´ →* — **lista de parâmetros**
11. *TypeIds´ → , TypeIds*

12. *Exp → let id = Exp in Exp* — **expressão de declaração**
13. *Exp → if Exp then Exp else Exp* — **expressão condicional**
14. *Exp → A Exp´*
15. *Exp´ → = A*
16. *Exp´ →*

17. *A → T A´*
18. *A´ → +T A´*
19. *A´ →*

20. *T → id T´*
21. *T´ → (Exps)*
22. *T´ →*
23. *T → num*

24. *Exps → Exp Exps´*
25. *Exps´ →*
26. *Exps´ → ,Exps*

Após o tratamento da linguagem, foram necessário calcularmos as tabelas NULLABLE, FIRST, FOLLOW e em seguida a tabela LL(1) para a implementação devida do analisador sintático. Sendo assim, a seguir temos as tabelas.

4.0.1 Tabelas *nullable, first* e *follow*

NT	Nullable	First	Follow
S	F	int bool	
Program	F	int bool	$
Funs	F	int bool	$
Funs'	V	int bool	$
Fun	F	int bool	int bool $
TypeId	F	int bool	(,)
TypeIds	F	int bool	)
TypeIds'	V	,	)
Exp	F	let if id num	in then else , int bool $)
Exp'	V	=	in then else , int bool $)
A	F	id num	= in then else , int bool $)
A'	V	+	= in then else , int bool $)
T	F	id num	+ = in then else , int bool $)
T'	V	(	+ = in then else , int bool $)
Exps	F	let if id num	)
Exps'	V	,	)

4.0.2 Tabela *LL(1)*

NT	let	id	num	if	int	bool	in	then	else	(	)	+	=	,	$
S					1	1									
Program					2	2									
Funs					3	3									
Funs'					5	5									4
Fun					6	6									
TypeId					7	8									
TypeIds					9	9									
TypeIds'											10			11	
Exp	12	14	14	13											
Exp'					16	16	16	16	16		16		15	16	16
A		17	17												
A'					19	19	19	19	19		19	18	19	19	19
T		20	23												
T'					22	22	22	22	22	21	22	22	22	22	22
Exps	24	24	24	24											
Exps'											25			26	