UFOP - Universidade Federal de Ouro Preto

ICEB - Instituto de Ciências Exatas e Biológicas

DECOM - Departamento de Computação

BCC328 - Construção de Compiladores I

ATIVIDADE PRÁTICA 01

Analisador Léxico para a linguagem de programação "Torben"

Aluno:

Gilberto Correa Mota - 12.1.4994

Prof . Dr. José Romildo Malaquias

Setembro de 2018

1 - Introdução

1.1 - Objetivo

O objetivo deste trabalho é programar um analisador léxico para a linguagem de programação apresentada pelo autor Torben [1] em seu livro de compiladores, no capítulo 4 - Interpretação (gramática 4.1)

1.2 - O papel do analisador

A função do analisador léxico é ler os caracteres do programa-fonte, agrupá-los em *lexemes* e produzir como saída uma sequencia de *tokens* para cada lexema no programa fonte. De forma resumida, a análise léxica identifica símbolos que compõem o programa [2].

A etapa seguinte é enviar o fluxo de *tokens* para o analisador sintático que registrará na tabela de símbolos as informações sobre os identificadores encontrados. Para esse trabalho nos interessa apenas o estudo da análise léxica.

A figura 1 explica o fluxo de funcionamento.

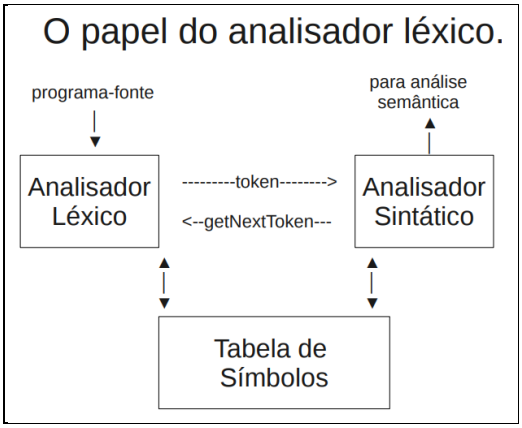


Figura1 - Fluxo de funcionamento do analisador léxico
Fonte: http://www.comp.uems.br/~chastel/compiladores [3]

O *Token* é um par contendo um nome e um valor de atributo (opcional). O nome do *token* é um símbolo que representa o tipo de unidade léxica esperado como entrada pelo analisador sintático (Figura 2). O padrão é uma descrição da forma com a qual os lexemas de um *token* podem assumir definidos por uma expressão regular.

Já o lexema, é uma sequência de caracteres do programa-fonte que "casa" com o padrão para um *token* e é identificado pelo analisador léxico como uma instância desse *token*.

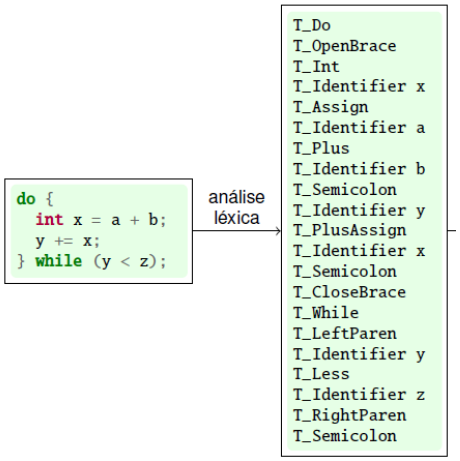
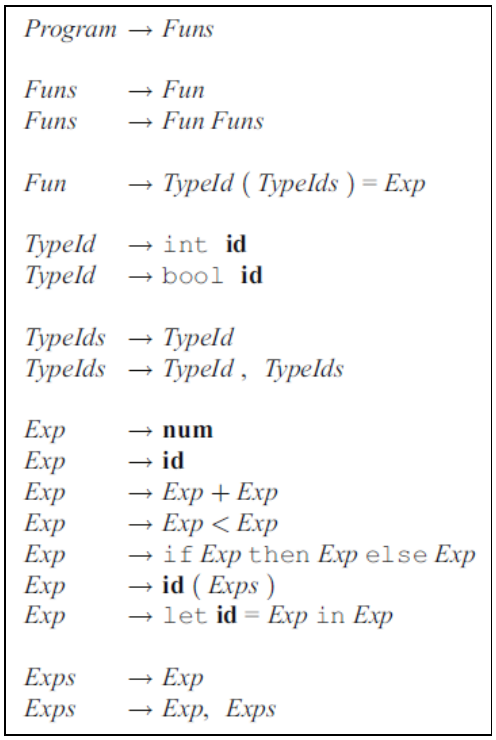


Figura 2 - Representação de uma análise léxica (adaptação)
Fonte: Slides do Prof. Romildo[2]

A figura 3 exibe a linguagem "torben" como exemplo para interpretação.



Program → *Funs*

Funs → *Fun*
Funs → *Fun Funs*

Fun → *TypeId* (*TypeIds*) = *Exp*

TypeId → `int` **id**
TypeId → `bool` **id**

TypeIds → *TypeId*
TypeIds → *TypeId* , *TypeIds*

Exp → **num**
Exp → **id**
Exp → *Exp* + *Exp*
Exp → *Exp* < *Exp*
Exp → `if` *Exp* `then` *Exp* `else` *Exp*
Exp → **id** (*Exps*)
Exp → `let` **id** = *Exp* `in` *Exp*

Exps → *Exp*
Exps → *Exp*, *Exps*

Figura 3 - Exemplo de linguagem
Fonte: Livro Torben [1] página 105

2 - Desenvolvimento

Parte do projeto foi desenvolvida em sala de aula em conjunto com o professor da disciplina e parte ficou para ser desenvolvida pelos alunos. Assim, como forma de otimizar o trabalho colaborativo entres os desenvolvedores, o professor recomendou o uso das ferramentas Git[1] e GitHub[2].

Outra ferramenta importante para esse trabalho foi o *JFlex*, que é um gerador de analisador léxico escrito em *Java* que gera código em *Java*.

O programa *JFlex* serve para se criar uma classe *Java* que faz a análise léxica de qualquer arquivo texto. Para que o *JFlex* crie esta classe, devemos indicar para ele um arquivo .j*flex*, que contém as regras da construção desta classe. No nosso caso usamos o lexer.jflex, que contém as especificações léxicas da linguagem.

Além do lexer.jFlex (que está dentro do Main/jflex), os outros principais arquivos desse projeto são LexerTest.java dentro de (Main/Test/java) e o parser.cup (Main/cup). Para realização dos testes, utilizou-se o arquivo LexerTest.java.

Explicaremos sobre as alterações dos arquivos a seguir e usaremos as imagens dos códigos para facilitar o entendimento.

Git[1] pronunciado [**git**] (ou pronunciado [dit] em inglês britânico) é um sistema de controle de versões distribuído, usado principalmente no desenvolvimento de software, mas pode ser usado para registrar o histórico de edições de qualquer tipo de arquivo. Fonte: https://pt.wikipedia.org/wiki/Git

GitHub[2]**:** é uma plataforma de hospedagem de código-fonte com controle de versão usando o Git. Ele permite que programadores, utilitários ou qualquer usuário cadastrado na plataforma contribuam em projetos privados e/ou Open Source de qualquer lugar do mundo.

2.1 - Arquivo LEXER.jflex

Os terminais explicitados no trabalho, tais como: atribuição, soma, divisão, foram tratados da seguinte forma:

```
":="          { return tok(ASSIGN); }
"="           { return tok(EQ); }
"("           { return tok(LPAREN); }
")"           { return tok(RPAREN); }
","           { return tok(COMMA); }
"+"           { return tok(PLUS); }
"-"           { return tok(MINUS); }
"*"           { return tok(TIMES); }
"/"           { return tok(DIV); }
"%"           { return tok(MOD); }
"<>"          { return tok(NE); }
"<"           { return tok(LT); }
"<="          { return tok(LE); }
">"           { return tok(GT); }
">="          { return tok(GE); }
"&&"          { return tok(AND); }
"||"          { return tok(OR); }


bool          {return tok(BOOL); }
int           {return tok(INT); }
string        { return tok(STRING); }
if            { return tok(IF); }
then          { return tok(THEN); }
else          { return tok(ELSE); }
while         { return tok(WHILE); }
do            { return tok(DO); }
let           { return tok(LET); }
in            { return tok(IN); }

{id}          { return tok(ID, yytext());}
```

```
id = [a-zA-Z][a-zA-Z0-9_]*
```

Note que uma operação que retorne o tipo e o nome do respectivo terminal é suficiente para tratá-los.

No caso dos terminais literais inteiros, *booleanos* e *strings*, necessitou-se também de mostrar o formato em que eles ocorreram como *print* abaixo.

```
[0-9]+        { return tok(LITINT, Integer.parseInt(yytext())); }
true|false    { return tok(LITBOOL, Boolean.parseBool(yytext())); }
```

```
<STR> \"          { yybegin(YYINITIAL); return tok(LITSTRING,
builder.toString(), locLeft(), locRight()); }
<STR> \\ b         { builder.append('\b'); }
<STR> \\ t         { builder.append('\t'); }
<STR> \\ n         { builder.append('\n'); }
<STR> \\ r         { builder.append('\r'); }
<STR> \\ f         { builder.append('\f'); }
<STR> \\ \\        { builder.append('\\'); }
<STR> \\ \"        { builder.append('"'); }
<STR> \\ [0-9]{3} {
builder.append((char)(Integer.parseInt(yytext().substring(1)))); }
<STR> \\ .         { error("invalid escape sequence in string
literal"); }
<STR> [^\"\n\\]+   { builder.append(yytext()); }
<STR> \n           { error("invalid newline in string literal"); }
```

```
<STR> <<EOF>>     { yybegin(YYINITIAL); error("unclosed string
literal"); }
```

Para o caso dos literais strings (que são formados por uma sequência gráfica delimitada por aspas duplas), verificou-se se a sequência é encerrada de maneira correta, caso contrário, o sistema emite uma mensagem de erro. De maneira análoga, foi necessário verificar o caracter "\" , que inicia uma sequência de *escape* conforme mostrado acima.

No caso dos espaços em branco, foi preciso identificar um de acordo com a expressão abaixo e ignora-los.

```
LineTerminator = \r|\n|\r\n

/* white spaces */
white_space = {LineTerminator} | [ \t\f]
```

```
{white_space}     { /* ignore */ }
```

Já para tratarmos os comentários e os blocos de comentários, foi necessário criar uma variável auxiliar para ajudar a controlar a quantidade de linhas de um bloco (conforme explicado em sala de aula). A partir daí, caso um bloco não terminar com #, uma mensagem de erro é exibida informando que o comentário não foi finalizado.

```
private int commentLevel = 0;
```

```
<COMMENT> {
"{#"            { commentLevel --;
                  if (commentLevel == 0)
                      yybegin(YYINITIAL);
                }
"#}"            { commentLevel ++; }
<<EOF>>         { yybegin(YYINITIAL);
                  System.err.println("error: unclosed comment");
                }
[^]             { }
}
```

2.2 Arquivo parser.cup

O arquivo *parser.cup* é utilizado para especificar os nomes dos terminais e seus tipos.

```
terminal                ASSIGN;
terminal                PLUS, MINUS, TIMES, DIV, MOD;
terminal                EQ, NE, LT, LE, GT, GE, AND, OR;
terminal                LPAREN, RPAREN, COMMA;
terminal          Integer LITINT;
terminal          String ID;
terminal          String LITBOOL;
terminal                BOOL, INT, STRING, IF, THEN, ELSE, WHILE, DO,
LET, IN;
terminal          String LITSTRING;
```

Finalmente, para encerrarmos esse trabalho prático, foi necessário especificar a associatividade dos operadores como sendo à esquerda ou à direita.

```
precedence left PLUS, MINUS, TIMES, DIV, MOD, AND, OR;
precedence right ASSIGN, THEN, ELSE, DO, IN;
```

Referências Bibliograficas

[1] Torben Ægidius Mogensen. Introduction to Compiler Design. Inglês. 2a ed. Springer, 2017. isbn: 978-3-319-66965-6.

[2] Slides do Prof. José Romildo Malaquias - Semestre 2014.2 (página 17). Construção de Compiladores - Capítulo 1.

[3] Slides do Prof. André Chastel Lima - UEMS Universidade Estadual de Mato Grosso do Sul. Disponível em http://www.comp.uems.br/~chastel/compiladores. Acessado em setembro/2018.