Analisador Léxico para Linguagem C-

Caio César Freitas Lara

Departamento de Ciência da Computação – Universidade Federal de Lavras – Lavras, MG

Abstract. *This article describes the Development Process of a lexical analyzer for a fictitious programming language. The lexical analyzer is part of a compiler for this language.*

Resumo. *Este artigo descreve o processo de desenvolvimento de um analisador sintático para uma linguagem de programação fictícia O analisador sintático é parte de um compilador para esta linguagem.*

1. Indrodução

Este trabalho tem o objetivo de implementar um analisador sintático, também conhecido como parser, como parte de um compilador para compreender como funciona os compiladores que trabalham por trás das linguagens de programação.

2. Referêncial Teórico

Um Compilador pode ser denotado como um programa que recebe como entrada um programa fonte e o traduz para um programa equivalente em outra linguagem (LOUDEN, 2004). Usualmente compiladores são utilizados para traduzir linguegem de programação de alto nível como C, Java, Python para linguagens de máquina (Assembly). Este processo facilita o trabalho do desenvolvedor que para desenvolver suas soluções teria que programar em baixo nível A primeira fase da compilação é a análise léxica. O seu objetivo é separar a seqüência de caracteres do texto de um programa-fonte em itens léxicos ou lexemas, que são sequencias de caracteres com um significado coletivo. Os lexemas são classificadoscomo identificadores, palavras-chave, operadores, constantes, símbolos de pontuação,entre outros [Aho et al. 2008]. O analisador Lexico também realiza outras tarefas como remover espaços em branco, comentários e detectar erros.

Tabela de símbolos é uma estrutura utilizada para guardar identificadores, ou seja nomes de variáveis, funções, enfim novas palavras declaradas por usuários, que podem se repetir no código. A maneira como a tabela de simbolos é construída varia de acordo com o projeto do compilador. Neste trabalho a mesma armazena um código e o nome do identificador.

A segunda fase de um processo de compilação é a análise sintática é responsável por identificar elementos léxicos na sequencia de acordo com a construção da linguagem, para isso é definido a gramática da linguagem, ou seja um conjunto de regras com simbolos terminais e não terminais.

Para auxiliar na implementação desse trabalho, foi utilizada a linguagem de programação Java, com o kit de desenvolvimento JDK 1.8 64 bits e a ferramenta ANTLR em sua versão 4.5.2, um gerador de analisador poderoso para leitura, processamento, execução, ou traduzir texto estruturado ou arquivos binários. É amplamente utilizado para construir linguagens, ferramentas e frameworks. Como Interface de Desenvolvimento foi utilizado o IntelliJ em sua versão 2016.2.4 Community, instalada em um sistema Linux (distribuição Ubuntu 16.04)

Foi criado uma linguagem de programação fictícia, chamada de C- (C-menos) baseada na linguagem de programação C.

3. A linguagem c-

Foi fornecido pelo professor Rafael Durelli a descrição da linguagem de programação fictícia que será contruído o analisador léxico.

• Tipos de dados: inteiro, real, caractere, arranjo e registro.

• Funções: recursão, parâmetros passados por valor.

• Comandos: Atribuição, if/else, while, E/S simples (tratados como funções).

• Comentários: texto entre /* e */ (sem comentários aninhados).

• Palavras reservadas: int, float, struct, if, else, while, void, return (caixa baixa)

• Símbolo Inicial: <programa>

1. <programa> ::= <declaração-lista>

2. <declaração-lista> ::= <declaração> {<declaração>}

3. <declaração> ::= <var-declaração> | <fun-declaração>

4. <var-declaração> ::= <tipo-especificador> <ident> ; | <tipo-especificador> <ident> <abre-colchete>

<num-int> <fecha-colchete> {<abre-colchete> <num-int> <fecha-colchete>} ;

5. <tipo-especificador> ::= int | float | char | void | struct <ident> <abre-chave> <atributos-declaração>

<fecha-chave>

6. <atributos-declaração> ::= <var-declaração> {<var-declaração>}

7. <fun-declaração> ::= <tipo-especificador> <ident> (<params>) <composto-decl>

8. <params> ::= <param-lista> | void

9. <param-lista> ::= <param> {, <param>}

10. <param> ::= <tipo-especificador> <ident> | <tipo-especificador> <ident> <abre-colchete> <fecha-colchete>

11. <composto-decl> ::= <abre-chave> <local-declarações> <comando-lista> <fecha-chave>

12. <local-declarações> ::= {<var-declaração>}

13. <comando-lista> ::= { <comando> }

14. <comando> ::= <expressão-decl> | <composto-decl> | <seleção-decl> | <iteração-decl> | <retorno-decl>

15. <expressão-decl> ::= <expressão> ; | ;

16. <seleção-decl> ::= if (<expressão>) <comando> |

17.

if (<expressão>) <comando> else <comando>

18. <iteração-decl> ::= while (<expressão>) <comando>

19. <retorno-decl> ::= return ; | return <expressão> ;

20. <expressão> ::= <var> = <expressão> | <expressão-simples>

21. <var> ::= <ident> | <ident> <abre-colchete> <expressão> <fecha-colchete> {<abre-colchete> <expressão>

<fecha-colchete>}

22. <expressão-simples> ::= <expressão-soma> <relacional> <expressão-soma> |

23.

<expressão-soma>

24. <relacional> ::= <= | < | > | >= | == | !=

25. <expressão-soma> ::= <termo> {<soma> <termo>}

26. <soma> ::= + | -

27. <termo> ::= <fator> {<mult> <fator>}

28. <mult> ::= * | /

29. <fator> ::= (<expressão>) | <var> | <ativação> | <num> | <num-int>

30. <ativação> ::= <ident> (<args>)
31. <args> ::= [<arg-lista>]
32. <arg-lista> ::= <expressão> {, <expressão>}
33. <num> ::= [+ | -] <dígito> {<dígito>} [. <dígito> {<dígito>}] [E [+ | -] <dígito> {<dígito>}]
34. <num-int> ::= <dígito> {<dígito>}
35. <digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
36. <ident> ::= <letra> {<letra> | <dígito>}
37. <letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z
38. <abre-chave> ::= {
39. <fecha-chave> ::= }
40. <abre-colchete> ::= [
41. <fecha-colchete> ::=]

4. Especificação da linguagem no ANTLR

Foi criado o arquivo Cmenos.g4 do tipo ANTLR, onde foram definidos os lexemas aceitos pela linguagem. Neste arquivo também foram definidas as regras da gramática da linguagem com seus simbolos terminais e não terminais. Exemplo.

```
termo : fator (mult fator)*;
mult : OP_MULTIPLICACAO |   OP_DIVISAO;
```

Na primeira regra “termo”, existe tres elementos não-terminais e na segunda regra “mult”, existem dois elementos não terminais.

Os símbolos não terminais, podem ser lexemas definidos na parte léxica, como no trecho de código acima, os lexemas OP_MULTIPLICACAO define o token “*” e OP_DIVISAO define “/”.

Atravéz deste arquivo, foi possível gerar o analisador Léxico e o Parser da linguagem.

5. Estrutura da aplicação

A aplicação é dividida em 4 classes e 2 interface implementadas

5.1 Classe principal

Na classe principal é feita a leitura com o arquivo que contenha o código na linguagem C-, o fluxo de token é capturado, e guardado no objeto “tokens” do tipo CommonTokensStream, e então é instanciado o parser, que tem os tokens passados na sua construção e então o parser é ativado chamando a função com o nome da declaração da primeira regra gramatical.

5.2 Classe NovosTokens

O ANTLR já possui uma classe de Tokens porém foi definido uma classe novo para tokens para que eles pudesses ser manipulados mais facilmente de acodo com as necessidades. Esta classe tem os atributos: tipo do token e o nome, que poderá ser o prório lexema do token ou um ponteiro para tabela de simbolos.

5.3 Cmenos

Cmenoslex é uma classe gerada automaticamente pelo ANTLR e contem os métodos que permitem trabalhar com os tokens.

5.4 CmenosLexer

CmenosLexer é uma classe gerada automaticamente pelo ANTLR, para utlização da análise léxica

5.5 CmenosParser

CmenosParser é uma classe gerada automaticamente pelo ANTLR, que contêm os métodos e atributos do parser.

5.6 Interfaces CmenosListener e CmenosVisitor

As interface CmenosListener e CmenosVisitor são interfaces geradas automaticamente pelo ANTLR e são implementas nos algorítmos do parser.

6 Execução da aplicação.

Para executar a aplicação basta descompactar o arquivo “analisadorLexico.zip”, abrir o terminal do seu sistema operacional, navegar até a pasta onde o arquivo foi descompactado e digitar:

java – jar analisadorSintatico.jar [caminho arquivo de código]

Exemplo 1 (para terminal UNIX):

java – jar analisadorSintatico.jar testes/teste_fornecido.cmm

Exemplo 2 (para terminal Windows):

java – jar analisadorSintatico.jar testes\teste_fornecido.cmm

Foi atribuído a extenção .cmm para os arquivos, porém é possível qualquer qualquer arquivo de texto fornecido.

Como saída são exibidos os erros léxicos e sintáticos.

7 Testes

Foi elaborado dois arquivos de testes: um contendo erros léxicos e outro sem erro. Também foi utilizado um arquivo fornecido pelo professor com 10 erros sintáticos Os arquivos de testes foram colocados na pasta teste.

8 Resultados

Após os testes e realizadas algumas correções foi possível visualizar os erros léxicos e sintáticos do arquivo fornecido.

9. Conclusão

Por meio deste trabalho foi possível como funciona na prática, uma das etapas do processo de compilação de códigos escreitos em linguagem de progração: a análise sintática Concluindo-se que é possível criar compiladores para novas linguagens de programação a partir de linguagens já existentes como o Java, utilizado neste trabalho e que ferramentas como o ANTLR podem ajudar no processo.

Na análise léxica as informações são organizadas a fim de serem usadas nas etapas sequintes e os itens desncessários como quebras de linhas e comentários são descartados.

A análise sintática útliza como produto os tokens gerados pela análise léxica para determinar se as declarações feitas pelo programador estão de acordo com as definições da linguagem.

10 Referências bibliográficas

AHO, S. Compiladores: Princípios, técnicas e ferramentas, Ed. LTC. 1996.

LOUDEN, K.C Compiladores – Princícios e Práticas. Ed.Thomson Pioneira, 2004.

DURELLI, Rafael – Slides de aula: Compiladores1.2016.2.Aula02.AnaliseLexica. 2016

DURELLI, Rafael – Slides de aula: Compiladores1.2016.2.Aula03.AnaliseSintáticaIntrodução. 2016