Analisador Léxico

Trabalho de Tradutores

Amanda Oliveira Alves - 15/0116276

Profª. Cláudia Nalon

Fevereiro de 2021

1 Objetivos

A proposta da disciplina de Tradutores é abordar os componentes e a implementação de um tradutor. O resultado final esperado do trabalho proposto é a construção de um tradutor para uma linguagem de programação simplificada baseada em C e feita para um propósito específico. O projeto será dividido em quatro etapas: Analisador Léxico, Analisador Sintático, Analisador Semântico e Gerador de Código Intermediário. Este relatório é referente a primeira etapa que consiste na construção do Analisador Léxico.

1.1 Motivação e proposta

Hoje vemos um grande avanço das linguagens de programação de alto nível, uma das mais populares é a linguagem Python, amplamente utilizada na área de inteligência artificial e outras áreas relacionadas. Esta linguagem possui nativamente a estrutura de dados Set[1] e as operações necessárias para a sua manipulação. Sets em Python podem parecer restritos, mas são estruturas de dados úteis e muito boas para armazenar elementos distintos. Essa facilidade na manipulação de conjuntos é necessária para realizar operações matemáticas complexas. Nas linguagens de baixo nível como C esse tipo de operação não existe de forma nativa, caso necessário o desenvolvedor precisa criar funções específicas ou utilizar bibliotecas. A proposta do trabalho é criar um tradutor de um subconjunto da linguagem C com as funcionalidades mencionadas.

2 Desenvolvimento

Primeiro, através da descrição da linguagem é extraída a sua gramática. Depois, as palavras-chave definidas na gramática serão utilizadas para gerar as expressões regulares. Então, acontece a análise dos *tokens* auxiliares (chaves, parênteses, pontuação, etc). Os erros léxicos encontrados são tratados, exibindo na mensagem a causa do erro e a sua localização (linha e coluna).

[1] https://algoritmosempython.com.br/cursos/programacao-python/conjuntos/

2.1 Funções adicionadas

Além de suas regras e expressões regulares, o analisador léxico precisou de funcionalidades adicionais para atingir os seus objetivos.

Para tratamento de símbolos inválidos, foram utilizadas as variáveis "numline"e "numcolumn", estas armazenam respectivamente a linha e coluna correspondente ao erro encontrado. O erro é impresso para o usuário exibindo o *token*, a linha e a coluna onde ele se localiza no arquivo de entrada.

2.2 Dificuldades

Foram encontrados desafios inerentes ao desenvolvimento na plataforma Flex que era desconhecida pela autora. Outra dificuldade foi a construção da linguagem livre de contexto correspondente a descrição fornecida pela professora.

3 Funcionamento

A primeira fase de um compilador é a análise léxica, na qual um analisador léxico lê o fluxo de caracteres presente no código fonte e os agrupa em sequência de lexemas [1]. Um lexema é uma unidade básica de significado para uma linguagem [2]. Assim, o analisador se utiliza da gramática fornecida para identificar os lexemas da linguagem. Caso identifique algo que não esteja especificado na linguagem, o analisador deve informar a localização da ocorrência.

3.1 Tratamento de Erros

Caso seja identificado um lexema que não pertença a linguagem, o analisador deve imprimir o lexema lido e a sua localização (linha e coluna), dessa forma facilitando a correção deste. Depois o analisador continua lendo o fluxo de caracteres.

4 Arquivos de teste

Para completar, juntamente com o analisador léxico existem arquivos de teste para o analisador. Existem quatro arquivos somente com entradas válidas e dois arquivos com caracteres inválidos. No arquivo README são encontradas as instruções para compilação e execução do código.

Referências

[1] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[2] David Crystal. *The Cambridge encyclopedia of the English language*. Ernst Klett Sprachen, 2004.

Anexo I - Gramática[1]

<translation-unit> → {<external-declaration>}*

<external-declaration> → <function-definition>
| <declaration>

<function-definition> → <type-specifier> <declarator> <compound-statement>

<type-specifier> → int
| float
| elem
| set

<declarator> → <identifier> ({<parameter-list>}*)

<parameter-list> → <parameter-declaration>
| <parameter-list> , <parameter-list>

<parameter-declaration> → <type-specifier> <identifier>

<conditional-expression> → <logical-or-expression>

<logical-or-expression> → <logical-and-expression>
| <logical-or-expression> || <logical-and-expression>

<logical-and-expression> → <equality-expression>
| <logical-and-expression> && <equality-expression>

<equality-expression> → <relational-expression>
| <equality-expression> → <relational-expression>
| <equality-expression> → <relational-expression>

<relational-expression> → <additive-expression>
| <relational-expression> < <additive-expression>
| <relational-expression> > <additive-expression>
| <relational-expression> <= <additive-expression>
| <relational-expression> >= <additive-expression>

<additive-expression> → <multiplicative-expression>
| <additive-expression> + <multiplicative-expression>
| <additive-expression> - <multiplicative-expression>

[1]https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

<multiplicative-expression> → <unary-expression>
| <multiplicative-expression> * <unary-expression>
| <multiplicative-expression> / <unary-expression>

<unary-expression> → <postfix-expression>
| <unary-operator> <unary-expression>

<postfix-expression> → <primary-expression>
| <postfix-expression> ({<assignment-expression>}*)

<primary-expression> → <identifier>
| <constant>
| <string>
| (<expression>)

<constant> → <integer-constant>
| <character-constant>
| <floating-constant>
| <empty-constant>

<expression> → <assignment-expression>
| <expression> , <assignment-expression>

<assignment-expression> → <conditional-expression>
| <unary-expression> = <assignment-expression>

<unary-operator> → +
| -
| !

<compound-statement> → { {<declaration>}* {<statement>}* }

<declaration> → <type-qualifier> <identifier> ;

<statement> → <expression-statement>
| <compound-statement>
| <selection-statement>
| <iteration-statement>
| <set-statement>
| <jump-statement>

<expression-statement> → {<expression>}? ;

<selection-statement> → if (<expression>) <statement>
| if (<expression>) <statement> else <statement>

<iteration-statement> → for ({<expression>}? ; {<expression>}? ; {<expression>}?)
<statement>

<set-statement> → <membership-expression>
| <existence-statement>
| <type-check-statement>
|<inclusion-statement>
| <removal-statement>
| <set-iteration-statement>
|<is-set-statement>

<membership-expression> → <expression> in <expression>

<existence-statement> → exists (<membership-expression>)
| exists (<existence-statement>)

<inclusion-statement> → add (<membership-expression>) ;
| add (<expression> in <inclusion-statement>)

<removal-statement> → remove (<membership-expression>) ;
| remove (<removal-statement>)

<set-iteration-statement> → forall (<membership-expression>) <statement>

<is-set-statement> → type-check-statement (<membership-expression>) <statement>

<io-statement> → <membership-expression>
| <read-statement>
| <write-statement>
|<writein-statement>

<read-statement> → <identifier>

<write-statement> → <constant>

<writein-statement> → <constant> <white-space>

<jump-statement> → return {<expression>}? ;

<identifier> → letter (letter | digit)∗

<string> → (\\.|[”#{}\\])∗

<letter> → a | … | z | A | … | Z

<digit> → 0 | … | 9

<white-space> → \n | | \t