Analisador Sintático

Lucas Mafra Chagas[12/0126443]

Prof.ª Dra. Cláudia Nalon

Resumo Esse projeto tem como intuito construir um analisador sintático para um subconjunto da linguagem C, mostrando sua nova primitiva de linguagem, sua descrição, instrução e gramática.

Keywords: Tradutores · Compiladores · Mini-C.

1 Objetivo

Tendo como base a metodologia de ensino baseada em projeto, determinou-se algumas diretrizes que o discente da Ciência da Computação necessita seguir para alcançar um melhor entendimento do conteúdo apresentado. Dessa forma, as diretrizes definidas consistem na implementação de um analisador de um subconjunto da linguagem C. Por fim, este projeto é constituído por quatro etapas: Implementação do analisador léxico; Implementação do analisador sintático; Implementação do analisador semântico; e Implementação do gerador de código intermediário.

2 Motivação

Um compilador é um programa que lê um programa escrito numa linguagem, conhecida como linguagem fonte, e o traduz num programa equivalente numa outra linguagem, conhecida como linguagem alvo. Essa ação é feita observando possíveis erros presentes no programa fonte[1].

Sendo assim, a linguagem projetada para o desenvolvimento do trabalho final segue as tratativas básicas da linguagem C e a adição de uma nova primitiva de dados, que é declarado como *set*. A declaração de uma variável desse tipo segue o molde de C, sem ter um tipo associado aos seus elementos, onde o *set* tem como papel ser composto por *elem*, onde elem é polimórfico, podendo assumir o papel de *int*, *float* ou de *set*[5].

Além disso, o novo tipo de dado da linguagem, o *set* possui operações específicas, como verificação de tipagem, *is_ set*, a inclusão de elemento em um conjunto, *add*, a remoção de elemento de um conjunto, *remove*, a seleção dentro desse conjunto, *exists*, e a iteração dentro desse conjunto, *forall*[5].

3 Descrição da Análise Léxica

A análise léxica é a primeira fase de um projeto de compilador. O analisador léxico é responsável por converter uma sequência de caracteres em uma sequência de lexemas, removendo qualquer comentário ou espaço em branco. O analisador léxico é responsável por varrer o código fonte do programa e identificar cada token, um por um[6].Durante essa fase, para uma melhor compreensão da linguagem desenvolvida, é necessário construir uma gramática livre de contexto que determina o seu funcionamento. Além da gramática, é necessário ter uma tabela de símbolos classificando os nomes das variáveis, as funções, as classes e as constantes[4].

No projeto desenvolvido, além das regras e expressões regulares, o analisador léxico necessitou da estruturação de algumas funcionalidades para a preparação do algoritmo. Essas novas estruturas servem de base para a construção da tabela de símbolos na próxima etapa. Para o tratamento de erros, foi construída uma função que destaca um token inválido ao ser encontrado, mostrando a linha e a coluna onde esse erro aconteceu. Foram adicionadas também sub-rotinas auxiliares para impressão colorida de texto e controle de linhas e colunas lidas.

4 Descrição da Análise Sintática

A análise sintática é a segunda fase de um projeto de compilador. O analisador sintático é responsável por apanhar os lexemas identificados pelo analisador léxico e descrevê-los de forma adequada aos seus programas[2]. Os tokens, que vem agrupados nos lexemas, possuem dois componentes: os seus nomes e seus valores de atributos. Os nomes dos tokens aparecem como símbolos terminais da gramática da linguagem, e seus atributos direcionam para a tabela de símbolos que possuem mais informações sobre os tokens[1].

Nessa segunda etapa, além da revisão do analisador léxico, foi necessário construir uma árvore sintática, que representa a estrutura sintática hierárquica do programa, e a estrutura da tabela de símbolos, que são estruturas de dados utilizados pelo compilador para obter informações sobre as construções do programa-fonte.

5 Descrição dos Arquivos de Teste

Os arquivos de teste buscam apresentar códigos sem erros e códigos que possuem algum erro para o analisador sintático. Os códigos inválidos buscam mostrar erros de um caractere ou string colocado incorretamente em um comando ou instrução que causa uma falha na execução, desrespeitando a sintaxe do programa.

No código incorreto1.mf, o código apresenta o erro de não ter uma chave para fechar o laço for, iniciado na linha 4.

No código incorreto2.mf, o código apresenta o erro de não ter um ponto e vírgula na chamada da função contador, na linha 10.

6 Instruções para Compilação

Para compilar, é necessário executar os seguintes comandos:

1. $ make
2. $ ./mafralang tests/corretoX.mf, onde X representa o número do exemplo, para rodar os códigos corretos;
3. $ ./mafralang tests/incorreto.mf, onde X representa o número do exemplo, para rodar os códigos incorretos.

Depois de compilar o código, ele gerará no próprio terminal a árvores sintática abstrata e a tabela de símbolos. Os erros sintáticos apareceram no terminal, informando a linha onde ocorreu.

Referências

1. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA (2006)
2. Grune, D., Jacobs, C.J.H.: Parsing Techniques: A Practical Guide. Ellis Horwood, USA (1990)
3. Kernigham, B.W., Ritchie, D.M.: The C programming language. Prentice hall of India (1973)
4. Louden, K.C.: Compiler construction. Cengage Learning (1997)
5. Nalon, C.: Trabalho prático - descrição da linguagem (2021), https://aprender3.unb.br/mod/page/view.php?id=294131, visitou em 26-02-2021
6. Ritchie, D.M., Kernighan, B.W., Lesk, M.E.: The C programming language. Prentice Hall Englewood Cliffs (1988)

A Gramática

A gramática foi feita a partir das adaptações da gramática apresentada por Brian W. Kernighan e Dennis M. Ritchie no livro K&R[3]. Os terminais são apresentados entre aspas simples e em caixa alta, enquanto as variáveis são apresentadas entre parênteses.

⟨*program*⟩ ::= ⟨*declarations*⟩

⟨*declarations*⟩ ::= ⟨*declarations*⟩ ⟨*declaration*⟩
| ⟨*declaration*⟩

⟨*declaration*⟩ ::= ⟨*var_ declaration*⟩
| ⟨*func_ declaration*⟩

⟨*var_ declaration*⟩ ::= 'TYPE' 'ID ';'

⟨*func_ declaration*⟩ ::= 'TYPE' 'ID '(' ⟨*paramaters*⟩ ')' ⟨*compound_ statement*⟩

⟨*paramaters*⟩ ::= ⟨*parameters*⟩ ',' ⟨*paramater*⟩
| ⟨*parameter*⟩
|

⟨*paramater*⟩ ::= 'TYPE' 'ID'

⟨*block_ struct*⟩ ::= ⟨*compound_ statement*⟩
| ⟨*block_ statement*⟩

⟨*coumpound_ statement*⟩ ::= '{' ⟨*statement_ list*⟩ '}'

⟨*statement_ list*⟩ ::= ⟨*statement_ list*⟩ ⟨*block_ statement*⟩
| ⟨*block_ statement*⟩

⟨*block_ statement*⟩ ::= ⟨*var_ declaration*⟩
| ⟨*set_ statement*⟩ ';'
| ⟨*statement*⟩
| ⟨*return_ statement*⟩
| ⟨*input_ statement*⟩ ';'
| ⟨*output_ statement*⟩ ';'
| ⟨*function_ call*⟩ ';'

⟨*function_ call*⟩ ::= 'ID' '(' ⟨*local_ paramaters*⟩ ')'

⟨*local_ parameters*⟩ ::= ⟨*local_ parameters*⟩ ',' ⟨*local_ parameter*⟩
| ⟨*local_ parameter*⟩

⟨*local_ paramenter*⟩ ::= 'ID'

⟨*input_ statement*⟩ ::= 'READ' '('⟨*expression*⟩ ')'

⟨*output_ statement*⟩ ::= 'WRITE' '(' ⟨*expression*⟩ ')'
| 'WRITELN' '(' ⟨*expression*⟩ ')'

⟨*return_ statement*⟩ ::= 'RETURN' '{' ⟨*expression*⟩ '}' ';'
| 'RETURN' ⟨*expression*⟩ ';'

⟨*statement*⟩ ::= ⟨*expression_ statement*⟩
| ⟨*conditional_ statement*⟩
| ⟨*iteration_ statement*⟩

⟨*expression_ statement*⟩ ::= ⟨*expression*⟩ ';'

⟨*conditional_ statement*⟩ ::= 'IF' '(' ⟨*expression*⟩ ')' ⟨*block_ struct*⟩
| 'IF' '(' ⟨*expression*⟩ ')' ⟨*block_ struct*⟩ 'else' ⟨*block_ struct*⟩

⟨*iteration_ statement*⟩ ::= 'FOR' '('⟨*expression*⟩ ';' ⟨*expression*⟩ ';' ⟨*expression*⟩')'
⟨*block_ struct*⟩
| 'FORALL '(' ⟨*set_ expression*⟩ ')' ⟨*block_ struct*⟩

⟨*set_ statement*⟩ ::= ⟨*is_ set_ statememt*⟩
| ⟨*remove_ statement*⟩
| ⟨*add_ statement*⟩
| ⟨*exists_ statement*⟩

⟨*is_ set_ statement*⟩ ::= 'IS_SET' '(' 'ID' ')'

⟨*remove_ statement*⟩ ::= 'REMOVE' '(' ⟨*set_ expression*⟩ ')'

⟨*add_ statement*⟩ ::= 'ADD' '(' ⟨*set_ expression*⟩ ')'

⟨*exists_ statement*⟩ ::= 'EXISTS' '(' ⟨*set_ expression*⟩ ')'

⟨*set_ expression*⟩ ::= ⟨*expression*⟩ 'IN' ⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*operation*⟩
| ⟨*expression*⟩ ',' ⟨*operation*⟩
| ⟨*function_ expression*⟩

⟨*function_ expression*⟩ ::= ⟨*set_ statement*⟩
| ⟨*function_ call*⟩

⟨*operation*⟩ ::= ⟨*arithmetic_ operation*⟩
| ⟨*variable*⟩ '=' ⟨*operation*⟩

⟨*arithmetic_ operation*⟩ ::= ⟨*logical_ operation*⟩
| ⟨*arithmetic_ operation*⟩ '+' ⟨*logical_ operation*⟩
| ⟨*arithmetic_ operation*⟩ '-' ⟨*logical_ operation*⟩
| ⟨*arithmetic_ operation*⟩ '*' ⟨*logical_ operation*⟩
| ⟨*arithmetic_ operation*⟩ '/' ⟨*logical_ operation*⟩

⟨*logic_ operation*⟩ ::= ⟨*relational_ operation*⟩
| '!' ⟨*logic_ operation*⟩
| ⟨*relational_ operation*⟩ '&&' ⟨*logic_ operation*⟩
| ⟨*relational_ operation*⟩ '||' ⟨*logic_ operation*⟩

⟨*relational_ operation*⟩ ::= ⟨*variable*⟩
| ⟨*variable*⟩ '==' ⟨*relational_ operation*⟩
| ⟨*variable*⟩ '<' ⟨*relational_ operation*⟩
| ⟨*variable*⟩ '≤' ⟨*relational_ operation*⟩
| ⟨*variable*⟩ '>' ⟨*relational_ operation*⟩
| ⟨*variable*⟩ '≥' ⟨*relational_ operation*⟩
| ⟨*variable*⟩ '!=' ⟨*relational_ operation*⟩

⟨*variable*⟩ ::= ⟨*constant*⟩
| ¨'¨ ⟨*string*⟩ ¨'¨
| 'ID'

⟨*constant*⟩ ::= 'INTEGER'
| 'FLOAT'
| 'EMPTY'