Analisador Sintático

Lucas Mafra Chagas

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, bem como de um sintetizador para 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*.[4]

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.[5]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.[3]

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 compilado. O analisador sintático é responsável por pegar os lexemas identificados pelo analisador léxico e descrever a forma adequada de seu programas. Isso é feito pela construção da árvore sintática abstrata e a tabela de símbolos. Para definir a sintaxe da linguagem, é utilizado a gramática livre-de-contexto. Ela tem a função de descrever a estrutura hierárquica da maioria das construções de linguagem de programação.

5 Descrição dos Arquivos de Teste

Os arquivos de teste apresentam códigos que não apresentam erros e que apresentam erros para o analisador sintático. Os códigos inválidos buscam mostrar erros de sequência de lexemas que acabam gerando erros sintáticos, como instruções sem ponto e vírgula, parenteses que não são fechados ou instruções mal formuladas.

6 Instruções para Compilação

Para compilar, é necessário executar os seguintes comandos:

1. $ make
2. $ ./mafralang validX.c, onde X representa o número do exemplo, para rodar os códigos corretos;
3. $ ./mafralang invalidX.c, onde X representa o número do exemplo, para rodar os códigos incorretos.

Depois de compilar o código, ele gerará no próprio terminal a análise sintática. Os erros sintáticos aparecem destacados, mostrando o erro, a linha e a coluna em que ocorrem. A tabela de símbolos aparecerá num arquivo mafralang.output.

Referências

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, principles, techniques. Addison wesley **7**(8), 9 (1986)
2. Kernigham, B.W., Ritchie, D.M.: The C programming language. Prentice hall of India (1973)
3. Louden, K.C.: Compiler construction. Cengage Learning (1997)
4. Nalon, C.: Trabalho prático - descrição da linguagem (2021), https://aprender3.unb.br/mod/page/view.php?id=294131, visitou em 26-02-2021
5. 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.[2] Os terminais são apresentados entre aspas simples, enquanto as variáveis são apresentadas entre parênteses.

⟨*program*⟩ ::= ⟨*declaration*⟩
| ⟨*function*⟩

⟨*function*⟩ ::= ⟨*type*⟩ 'id' '{'⟨*declaration*⟩'}'* ⟨*compound_ statement*⟩

⟨*type*⟩ ::= 'int'
| 'float'
| 'elem'
| 'set'

⟨*declaration*⟩ ::= '(' ⟨*declaration*⟩ ')'
| ⟨*declaration*⟩ ⟨*operation*⟩? ';'

⟨*operation*⟩ ::= ⟨*arithmetic_ operation*⟩
| ⟨*logic_ operation*⟩
| ⟨*relational_ operation*⟩
| ⟨*input_ operation*⟩
| ⟨*ouput_ operation*⟩

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

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

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

| ⟨*relational_ operation*⟩ '!=' ⟨*relational_ operation*⟩
| ⟨*expression*⟩

⟨*input_ operation*⟩ ::= 'read' '('⟨*expression*⟩ ')'

⟨*output_ operation*⟩ ::= 'write' '(' ⟨*expression*⟩ ')'
| 'writeln' '(' ⟨*expression*⟩ ')'

⟨*expression*⟩ ::= ⟨*operation*⟩
| ⟨*expression*⟩ ',' ⟨*operation*⟩
| ⟨*constant*⟩
| 'id'

⟨*constant*⟩ ::= ⟨*integer_ constant*⟩
| ⟨*float_ constant*⟩
| ⟨*char_ constant*⟩
| 'EMPTY'

⟨*compound_ statement*⟩ ::= '' ⟨*declaration*⟩* ⟨*statement*⟩* ''

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

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

⟨*iteration_ statement*⟩ ::= 'for' '('⟨*expression*⟩? ';' ⟨*expression*⟩? ';' ⟨*expression*⟩?')'
⟨*statement*⟩
| forall '('⟨*expression*⟩? ';' ⟨*expression*⟩? ';' ⟨*expression*⟩?')'
⟨*statement*⟩

⟨*conditional_ statement*⟩ ::= 'if' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
| 'if' '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ 'else' ⟨*statement*⟩

⟨*return_ statement*⟩ ::= 'return' ⟨*expression*⟩? ';'