Projeto de Compiladores

**Alexandre L. Fonseca, Artur K. Somenzi, Eduardo F. Valim,
Gabriel S. Ziviani, Thalita M. Lima**

1

Resumo. *Um Compilador é, basicamente, um programa que analisa a sequência de comandos de um código fonte escrito em uma determinada linguagem de programação e a traduz para a linguagem de máquina entendida pelo sistema computacional. Para executar a compilação, é necessário que o código fonte passe por diferentes etapas de análise. Entre elas estão a Análise Léxica, a Análise Sintática e a Análise Semântica. Para que, por fim, o Compilador possa realizar geração de código na linguagem de máquina. Este projeto teve como objetivo entender o funcionamento das fases de Análise Léxica e Sintática e implementá-las.*

1. Analisador Léxico

O Analisador Léxico, também chamado de scanner, é um programa de computador que tem como função ler um código-fonte e reconhecer símbolos válidos em uma determinada linguagem através da implementação de um autômato finito. Trata-se da primeira fase da execução de um Compilador. O objetivo dessa primeira parte do Projeto de Compiladores foi elaborar um Analisador Léxico. Para essa primeira etapa do Projeto, o grupo decidiu que seria usado um gerador automático chamado "Lex". Nesse caso, primeiro foi necessário identificar os símbolos que deveriam ser reconhecidos e as funções de cada uma deles para adicionar ao código do Lex. Depois de identificados, os símbolos foram agrupados de acordo com as seguintes classificações:

- Símbolos simples
- Operadores de adição
- Operadores de multiplicação
- Operadores relacionais
- Constantes
- Identificadores

Além desses grupos, também foi necessário identificar as palavras reservadas que seriam utilizadas na linguagem e separá-las dos identificadores. Para isso, foi criado um vetor contendo as palavras reservadas da linguagem e implementada uma lista de strings para formar a primeira versão da tabela de símbolos necessária para os identificadores. As palavras reservadas identificadas na linguagem são:

- INTEGER: identificador do tipo inteiro, pré-definido da linguagem;
- BOOLEAN: identificador do tipo lógico, pré-definido da linguagem;
- TRUE: identificador da constante lógica VERDADEIRO (valor 1), pré-definido da linguagem;
- FALSE: identificador da constante lógica FALSO (valor 0), pré-definido da linguagem;

- READ: identificador da função de leitura, pré-definida da linguagem;
- WRITE: identificador da função de escrita, pré-definida da linguagem;
- VOID: identificador do tipo de função sem retorno;
- WHILE: identificador de estrutura de repetição;
- IF: identificador de condição;
- RETURN: identificador que indica o retorno de um valor proveniente de uma execução;
- GOTO: identificador para estrutura de salto de instruções;
- TYPE: identificador usado para definir o tipo de uma variável;
- FUNCTION: identificador de uma função;
- LABEL: identificador de rótulo;
- VAR: identificador de uma variável;
- ELSE: identificador de caminho alternativo para a condição IF.

2. Analisador Sintático

O processo de Análise Sintática é a segunda fase de uma compilação, nela se analisa a estrutura gramatical de uma determinada entrada de acordo com a gramática utilizada pela linguagem em questão. É nessa fase que verificamos se uma determinada cadeia de símbolos léxicos está corretamente utilizada e na ordem em que os símbolos devem aparecer. Nessa segunda parte do projeto, o grupo iniciou utilizando o gerador automático "Yacc" para a criação do Analisador Sintático. Porém, devido a algumas dificuldades em utilizar o sistema Yacc, optamos por fazer a implementação manual do Analisador utilizando a linguagem C.

Desta forma, foi necessário criar funções para cada uma das regras de produção da gramática. Essa parte está declarada em analisador_sintatico.h e implementada no arquivo analisador_sintatico.c, junto com a main do programa. Além disso, para essa parte do projeto, foi preciso implementar uma versão melhorada da tabela de símbolos inicial para que fosse utilizada no Analisador Sintático.

A Tabela de Símbolos é uma estrutura de dados utilizada pelo Compilador para armazenar os identificadores e informações sobre eles (como funções, variáveis e tipos de dados). Durante a fase de Análise Sintática, ela é consultada para efeitos de comparação, armazenamento e atualização dos dados conforme é feita a varredura do código.

Nesse projeto, pode-se observar sua implementação nos arquivos tabeladesimbolos.h e tabeladesimbolos.c. Em tabeladesimbolos.c, foram usadas structs para descrever cada um dos tokens principais da gramática unidas por uma union que deverá guardar esses dados conforme forem encontrados pelo analisador.

3. Produções gramaticais

program ⇒ *function*

function ⇒ *identifier identifier formal_parameters block*

function ⇒ **void** *identifier formal_parameters block*

block ⇒ *body*

block ⇒ **labels** *body*

block ⇒ **variables** *body*

block ⇒ **functions** *body*

block ⇒ **labels variables** *body*

block ⇒ **labels functions** *body*

block ⇒ **variables functions** *body*

block ⇒ **labels variables functions** *body*

labels ⇒ **labels** *identifier_list* ;

variables ⇒ **vars** *identifier_list* : *type* ; *variables_*

*variables_* ⇒ *identifier_list* : *type* ; *variables_*

*variables_* ⇒ ϵ

functions ⇒ **functions** *function functions_*

*functions_* ⇒ *function functions_*

*functions_* ⇒ ϵ

body ⇒ (*body_*)

*body_* ⇒ *statement body_*

*body_* ⇒ ϵ

type ⇒ *identifier*

formal_parameters ⇒ (*formal_parameter formal_parameters_*)

formal_parameters ⇒ ( )

*formal_parameters_* ⇒ ; *formal_parameter formal_parameters_*

*formal_parameters_* ⇒ ϵ

formal_parameter ⇒ *expression_parameter*

expression_parameter ⇒ **var** *identifier_list* : *identifier*

expression_parameter ⇒ *identifier_list* : *identifier*

statement ⇒ *compound*

statement ⇒ *unlabeled_statement*

statement ⇒ *identifier statement_*

*statement_* ⇒ : *unlabeled_statement*

*statement_* ⇒ *unlabeled_statement_*

unlabeled_statement ⇒ *identifier unlabeled_statement_*

unlabeled_statement ⇒ **goto**

unlabeled_statement ⇒ **return**

unlabeled_statement ⇒ *conditional*

unlabeled_statement ⇒ *repetitive*

unlabeled_statement ⇒ ;

*unlabeled_statement_* ⇒ *function_call* ;

*unlabeled_statement_* ⇒ *assignment*

assignment ⇒ **=** *expression* ;

goto ⇒ **goto** *identifier* ;

return ⇒ **return** *return_*

*return_* ⇒ ;

*return_* ⇒ *expression* ;

compound ⇒ (*unlabeled_statement compound_*)

*compound_* ⇒ *unlabeled_statement compound_*

*compound_* ⇒ ϵ

conditional ⇒ **if** (*expression*) *compound conditional_*

*conditional_* ⇒ **else** *compound*

*conditional_* ⇒ ϵ

repetitive ⇒ **while** (*expression*) *compound*

expression ⇒ *simple_expression expression_*

*expression_* ⇒ *relop simple_expression*

*expression_* ⇒ ϵ

simple_expression ⇒ *addop term simple_expression*

simple_expression ⇒ *term simple_expression*

*simple_expression_* ⇒ *addop term simple_expression_*

*simple_expression_* ⇒ || *term simple_expression_*

*simple_expression_* ⇒ ϵ

term ⇒ *factor term_*

*term_* ⇒ *mulop factor term_*

*term_* ⇒ ϵ

factor ⇒ *identifier factor_*

factor ⇒ *constant*

factor ⇒ (*expression*)

factor ⇒ ! *factor*

*factor_* ⇒ *function_call*

*factor_* ⇒ ϵ

function_call $\Rightarrow$ *identifier* (*expression_list*)

identifier_list $\Rightarrow$ *identifier identifier_list_*

*identifier_list_* $\Rightarrow$, *identifier identifier_list_*

*identifier_list_* $\Rightarrow \epsilon$

expression_list $\Rightarrow$ *expression expression_list_*

expression_list $\Rightarrow \epsilon$

*expression_list_* $\Rightarrow$, *expression expression_list_*

*expression_list_* $\Rightarrow \epsilon$

relop $\Rightarrow$ ==

relop $\Rightarrow$!=

relop $\Rightarrow <$

relop $\Rightarrow$ <=

relop $\Rightarrow >$

relop $\Rightarrow$ >=

addop $\Rightarrow$ +

addop $\Rightarrow$ -

mulop $\Rightarrow$ *

mulop $\Rightarrow$ /

mulop $\Rightarrow$ &&

identifier $\Rightarrow$ *letter identifier_*

*identifier_* $\Rightarrow$ *letter identifier_*

*identifier_* $\Rightarrow$ *digit identifier_*

*identifier_* $\Rightarrow \epsilon$

constant $\Rightarrow$ *digit constant_*

*constant_* $\Rightarrow$ *digit constant_*

*constant_* $\Rightarrow \epsilon$

letter $\Rightarrow$ a...z

digit $\Rightarrow$ 0...9

4. Conclusão

A partir da realização deste trabalho torna-se possível entender de perto o funcionamento de cada uma das fases de um Compilador. Além de criar estratégias durante a aplicação do código para que fosse possível implementar, utilizando linguagem C, o funcionamento dos analisadores necessários para o processo de compilação.

5. Referências

1. https://celsokitamura.com.br/o-que-e-compilador
2. https://blog.betrybe.com/tecnologia/compilacao
3. https://www.alura.com.br/artigos/o-que-e-compilacao
4. https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/syntax-analysis.html