Analisador Sintático

Heitor de Lima Belém

Universidade de Brasília cic@unb.com

1 Motivação

Este trabalho tem como objetivo a apresentação das análises léxica e sintática, responsáveis por ler caractere por caractere de um programa, agrupar e formar tokens reconhecidos por uma linguagem e também por avaliar se a sequência de tokens produzidos obedecem às regras de uma linguagem (gramática). Para fixação do conteúdo obtido através do livro base da disciplina [ALSU07], foi proposto, inicialmente, o desenvolvimento dos analisadores léxico e sintático para a linguagem *C-IPL* [Nal], baseada nos princípios da linguagem *C*. O objetivo dessa linguagem é acrescentar um tipo de dados não existente em *C*, a lista.

Listas implementam uma coleção organizada de valores, assim como os vetores, entretanto, elas possuem operações especiais de acesso, adição e remoção de itens, que podem variar de linguagem para linguagem. Para a *C-IPL* [Nal], foram apresentadas operações de acesso aos elementos através dos operadores '?' e '!', operações de remoção utilizando o operador '%' e de inserção por meio do ':'. Por fim, são descritas funções para operar sobre as listas, sendo elas: filter, representada pelo operador binário '<<' e map, representada por '>>'.

2 Descrição da análise léxica

Para implementar o analisador léxico da linguagem proposta, foi utilizado o programa *FLEX* (*Fast Lexical Analyzer*), uma ferramenta geradora de programas que reconhecem padrões léxicos em textos [Est]. A estrutura de um arquivo reconhecido pelo *FLEX*, que possui a extensão .l, é dividida em três partes:

1. Definições
   Definições de funções, constantes, variáveis globais e inclusão de bibliotecas. Para este trabalho, foram criadas três variáveis globais: *errors_count*, *line_idx* e *column_idx*, que representam, respectivamente, a quantidade de erros obtidos durante a análise e o número da linha e coluna atual.
2. Regras
   Aqui são escritas as expressões regulares que vão procurar padrões no arquivo juntamente com as ações a serem tomadas ao encontrar tais padrões.
3. Código
   Nesta seção, encontra-se o código da função principal do arquivo com extensão *.l*, é aqui que será colocado o código escrito pelo usuário, este código é transferido e incorporado para o programa que implementa o autômato.

É importante ressaltar que, na fase de análise léxica, além da geração dos tokens através do processo de escaneamento do texto, é feita também a determinação dos escopos existentes no programa. Para isso, foi utilizad um contador global responsável por identificar o escopo e incrementar a contagem à medida que um token de abertura de chaves, '{', for encontrado.

3 Descrição da análise sintática

Para o analisador sintático deste trabalho, foi utilizada a ferramenta *Bison* [CS21], que consiste em um programa que gera, a partir das regras de uma gramática livre do contexto, um analisador sintático LR(1) canônico.

A estrutura básica de um arquivo reconhecido pelo *Bison*, que tem a extensão *.y*, segue a mesma ideia do Flex, com as mesmas seções. Entretanto, na seção de regras do Bison é onde ficam as regras da gramática livre de contexto, que se assemelham à seguinte forma:

```
non-terminal
  : non-terminal terminal
  | terminal
;
```

3.1 Tabela de Símbolos

A tabela de símbolos é uma estrutura auxiliar, utilizada pelo compilador, para localizar variáveis ou funções (símbolos) utilizados durante a execução de um programa.

Para este trabalho, a implementação desse componente foi realizada através da utilização de uma lista de estruturas do tipo *T_Symbol*, que armazena informações importantes para a próxima etapa do processo de compilação: a análise semântica.

As informações guardadas para cada símbolo são: conteúdo do símbolo, linha, coluna, escopo do identificador e flag para indicar se o símbolo é variável ou função.

3.2 Árvore sintática

Para a criação da árvore sintática, foi utilizada uma estrutura de dados composta por nós não terminais e terminais. Cada nó da árvore possui informações sobre a regra atual da gramática, campo para indicar se o nó é terminal ou não, caso não seja terminal, existe também um campo para conectar aos nós filhos.

Com essa estrutura, é possível percorrer a árvore utilizando um algoritmo de busca em profundidade, apresentando dados relevantes sobre cada nó visitado e, posteriormente, buscando informações necessárias para a análise semântica.

4 Descrição dos arquivos de teste

Os arquivos utilizados para verificar o funcionamento do analisador sintático desenvolvido no trabalho estão no subdiretório */tests*. Arquivos com o prefixo *success_*, representam os casos em que a análise sintática não identifica nenhum erro durante o processo. Já os arquivos com o prefixo *wrong_* englobam os casos em que o analisador identifica erros sintáticos.

Os erros apresentados para cada arquivo estão identificados abaixo:

- *wrong_ex1.c*

```
[PARSER] Line: 2 | Column: 1 => ERROR syntax error,
unexpected SIMPLE_TYPE, expecting '(' or ';'
[PARSER] Line: 3 | Column: 16 => ERROR syntax error, unexpected '}',
expecting end of file or SIMPLE_TYPE
[PARSER] Line: 12 | Column: 16 => ERROR syntax error, unexpected IO_READ
[PARSER] Line: 14 | Column: 30 => ERROR syntax error, unexpected ')'
```

- *wrong_ex2.c*

```
[PARSER] Line: 3 | Column: 14 => ERROR syntax error, unexpected IDENTIFIER,
expecting SIMPLE_TYPE or ')'
[PARSER] Line: 30 | Column: 17 => ERROR syntax error,
unexpected BINARY_LIST_OP
[PARSER] Line: 56 | Column: 9 => ERROR syntax error, unexpected '=',
expecting ';'
```

5 Compilação e execução do programa.

Requisitos para compilação: os programas *FLEX* e *BISON* (versão 2.6.4 e 3.7.6, respectivamente), o compilador *GCC* (versão 11.1.0), GNU Make (versão 4.3). Com isso devidamente instalado, é possível prosseguir para os seguintes passos.

- No diretório raiz do projeto:
- Verificar se existe a pasta */src/obj/*, caso não exista, é necessário criar antes de executar os comandos abaixo.

```
$ make all
$ ./tradutor ./tests/<nome_do_arquivo>.c
```

- Se desejar executar o valgrind para verificar possíveis vazamentos de memória, basta executar os seguintes comandos:

```
$ make all
$ make valgrind ./tradutor ARGS="<caminho_arquivo_teste>"
```

– Os comandos executados na compilação, com suas respectivas flags, são os seguintes:

```
bison -d ./src/parser.y -Wcounterexamples -v
flex ./src/scanner.l
gcc -c -o src/obj/parser.tab.o src/parser.tab.c -ll -Wall -g -I ./lib
gcc -c -o src/obj/symbol_table.o src/symbol_table.c -ll -Wall -g -I ./lib
gcc -c -o src/obj/syntatic_tree.o src/syntatic_tree.c
-ll -Wall -g -I ./lib
gcc -c -o src/obj/lex.yy.o src/lex.yy.c -ll -Wall -g -I ./lib
gcc -o tradutor src/obj/parser.tab.o src/obj/symbol_table.o
src/obj/syntatic_tree.o src/obj/lex.yy.o -ll -Wall -g -I ./lib -lm
```

Referências

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, Tools*. Pearson/Addison Wesley, 2nd edition, 2007.

[CS21] R. Corbett and R. Stallman. Bison. https://www.gnu.org/software/bison/manual/bison.pdf, Online; acessado 08 de Agosto de 2021.

[Est] W. Estes. Flex: Fast lexical analyser generator. https://github.com/westes/flex. Online; acessado 08 de Agosto de 2021.

[Gup21] A. Gupta. The syntax of C in Backus-Naur form. https://tinyurl.com/max5eep, Online; acessado 08 de Agosto de 2021.

[Nal] C. Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Acessado pela última vez em 10/08/2021.

A Gramática

Gramática que descreve o compilador da linguagem C-IPL [Nal], corrigida do trabalho anterior, seguindo as observações da professora. A estrutura do [Gup21] foi utilizada como base.

⟨*program*⟩ ::= ⟨*declarations*⟩

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

⟨*declaration*⟩ ::= ⟨*function_declaration_statement*⟩
| ⟨*variable_declaration_statement*⟩

⟨*block*⟩ ::= '{' ⟨*statements*⟩ '}'

⟨*statements*⟩ ::= ⟨*statements*⟩ ',' ⟨*statement*⟩
| ⟨*statement*⟩

⟨*statement*⟩ ::= ⟨*expression_statement*⟩
| ⟨*io_statement*⟩
| ⟨*return_statement*⟩
| ⟨*variable_declaration_statement*⟩
| ⟨*for_statement*⟩
| ⟨*if_else_statement*⟩
| ⟨*block*⟩

⟨*function_declaration_statement*⟩ ::= ⟨*SIMPLE_TYPE*⟩ ⟨*IDENTIFIER*⟩ '(' ⟨*parameters_optative*⟩ ')' ⟨*statement*⟩
| ⟨*SIMPLE_TYPE*⟩ ⟨*LIST_TYPE*⟩ ⟨*IDENTIFIER*⟩ '(' ⟨*parameters_optative*⟩ ')' ⟨*statement*⟩

⟨*parameters_optative*⟩ ::= ⟨*empty*⟩
| ⟨*parameters*⟩

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

⟨*parameter*⟩ ::= ⟨*SIMPLE_TYPE*⟩ ⟨*IDENTIFIER*⟩
| ⟨*SIMPLE_TYPE*⟩ ⟨*LIST_TYPE*⟩ ⟨*IDENTIFIER*⟩

⟨*for_statement*⟩ ::= ⟨*RW_FOR*⟩ '(' ⟨*expression_optative*⟩ ';' ⟨*or_expression_optative*⟩ ';' ⟨*expression_optative*⟩ ')' ⟨*statement*⟩

⟨*if_else_statement*⟩ ::= ⟨*RW_IF*⟩ '(' ⟨*expression*⟩ ')' ⟨*statement*⟩
| ⟨*RW_IF*⟩ '(' ⟨*expression*⟩ ')' ⟨*statement*⟩ ⟨*RW_IF*⟩ '(' ⟨*expression*⟩ ')' ⟨*statement*⟩

⟨*expression_statement*⟩ ::= ⟨*expression*⟩ ';'

⟨*io_statement*⟩ ::= ⟨*input_statment*⟩
| ⟨*output_statement*⟩

⟨*input_statement*⟩ ::= ⟨*IO_READ*⟩ '(' ⟨*IDENTIFIER*⟩ ';'

⟨*output_statement*⟩ ::= ⟨*IO_WRITE*⟩ '(' ⟨*expression*⟩ ')' ';'
| ⟨*IO_WRITE*⟩ '(' ⟨*LIT_STRING*⟩ ')' ';'

⟨*return_statement*⟩ ::= 'return' ⟨*expression*⟩ ';'

⟨*expression*⟩ ::= ⟨*IDENTIFIER*⟩ '=' ⟨*expression*⟩
| ⟨*or_expression*⟩
| ⟨*function_call_expression*⟩

⟨*function_call_expression*⟩ ::= ⟨*IDENTIFIER*⟩ '(' ⟨*function_arguments_optative*⟩ ')'

⟨*function_arguments_optative*⟩ ::= ⟨*empty*⟩
| ⟨*function_arguments*⟩

⟨*function_arguments*⟩ ::= ⟨*function_arguments*⟩ ',' ⟨*function_argument*⟩
| ⟨*function_argument*⟩

⟨*function_argument*⟩ ::= ⟨*expression*⟩

⟨*expression_optative*⟩ ::= ⟨*empty*⟩
| ⟨*expression*⟩

⟨*or_expression_optative*⟩ ::= ⟨*empty*⟩
| ⟨*or_expression*⟩

⟨*or_expression*⟩ ::= ⟨*or_expression*⟩ ⟨*LOGICAL_OP_OR*⟩ ⟨*and_expression*⟩
| ⟨*and_expression*⟩

⟨*and_expression*⟩ ::= ⟨*and_expression*⟩ ⟨*LOGICAL_OP_AND*⟩ ⟨*equality_expression*⟩
| ⟨*equality_expression*⟩

⟨*equality_expression*⟩ ::= ⟨*equality_expression*⟩ ⟨*EQUALITY_OP*⟩ ⟨*relational_expression*⟩
| ⟨*relational_expression*⟩

⟨*relational_expression*⟩ ::= ⟨*relational_expression*⟩ ⟨*RELATIONAL_OP*⟩ ⟨*list_expression*⟩
| ⟨*list_expression*⟩

⟨*list_expression*⟩ ::= ⟨*list_expression*⟩ ⟨*BINARY_LIST_OP*⟩ ⟨*addition_expression*⟩
| ⟨*addition_expression*⟩

⟨*addition_expression*⟩ ::= ⟨*addition_expression*⟩ ⟨*ARITMETIC_OP_ADDITIVE*⟩ ⟨*multiplication_expression*⟩
| ⟨*multiplication_expression*⟩

⟨*multiplication_expression*⟩ ::= ⟨*multiplication_expression*⟩ ⟨*ARITMETIC_OP_MULTIPLICATIVE*⟩ ⟨*simple_value*⟩
| ⟨*simple_value*⟩

⟨*simple_value*⟩ ::= ⟨*constant*⟩
| ⟨*IDENTIFIER*⟩
| ⟨*ARITMETIC_OP_ADDITIVE*⟩ ⟨*simple_value*⟩
| '!' ⟨*simple_value*⟩
| ⟨*UNARY_LIST_OP*⟩ ⟨*simple_value*⟩
| '(' ⟨*expression*⟩ ')'

⟨*variable_declaration_statement*⟩ ::= ⟨*SIMPLE_TYPE*⟩ ⟨*IDENTIFIER*⟩ ';'
| ⟨*SIMPLE_TYPE*⟩ ⟨*LIST_TYPE*⟩ ⟨*IDENTIFIER*⟩ ';'

⟨*constant*⟩ ::= ⟨*C_INTEGER*⟩ | ⟨*C_FLOAT*⟩ | ⟨*C_NIL*⟩

B Definições Regulares

Padrões	**Expressões Regulares**
DIGIT	[0-9]
CHARACTER	[a-zA-Z]
C_INTEGER	{digit}+
C_FLOAT	{digit}*.{digit}+
C_NIL	'NIL'
T_SIMPLE	int | float
T_LIST	list
IDENTIFIER	{character}|[_]({character}|{digit}|[_])*
UNARY_LIST_OP	'?' | '%'
BINARY_LIST_OP	':' | '<<' | '>>'
EQUALITY_OP	'==' | '!='
RELATIONAL_OP	'>' | '<' | '>=' | '<='
ARITMETIC_OP_-ADDITIVE	'+' | '-'
ARITMETIC_OP_-MULTIPLICATIVE	'*' | '/'
ASSIGN_OP	'='
LOGICAL_OP_AND	'&&'
LOGICAL_OP_OR	'||'
RW_FOR	'for'
RW_IF	'if' | 'else'
RW_RETURN	'return'
IO_READ	'read'
IO_WRITE	'write' | 'writeln'
DELIMITERS	'(' | ')' | '{' | '}' | ';' | ','
LIT_STRING	" .* "
EXCLAMATION_OP	'!'