

Analizador Semântico

Heitor de Lima Belém

Universidade de Brasília 160123950@aluno.unb.br

1 Motivação

Este trabalho tem como objetivo a apresentação das análises léxica, sintática e semântica, responsáveis por ler caractere por caractere de um programa, agrupar, formar tokens reconhecidos por uma linguagem, avaliar se a sequência de tokens produzidos obedecem às regras de uma linguagem (gramática) e, além disso, verificar os aspectos relacionados às instruções do programa. Para fixação do conteúdo obtido através do livro base da disciplina [ALSU07], foi proposto, inicialmente, o desenvolvimento dos analisadores léxico, sintático e semântico 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 utilizado um contador global responsável por identificar o contagem de escopos atual.

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 da análise semântica

A análise semântica é responsável por verificar as instruções existentes em um código e determinar se, combinadas, elas fornecem um programa que possua sentido. Por exemplo, avaliar se uma variável utilizada foi declarada e está acessível em um determinado trecho de código, apontar problemas em chamadas de funções com número errado de parâmetros, entre outros erros descritos posteriormente nesta seção.

Para realizar a análise semântica, foi utilizada a seção de regras do Bison [CS21], em conjunto com a tabela de símbolos, árvore sintática gerada pela análise sintática e pilha de escopos.

4.1 Regras gerais

Conforme exposto na descrição do trabalho [Nal], é necessário que o programa contenha a definição de uma função com o identificador `main` como a principal do programa. Para realizar esta análise, basta apenas percorrer a tabela de símbolos em busca de uma entrada que tenha como tipo “função” e que o identificador seja “main”.

4.2 Declaração e utilização de variáveis e funções

Para esta análise, foi utilizada a pilha de escopos e a tabela de símbolos. As etapas para a verificação consistem em, para a declaração de variáveis e funções, identificar se a variável ou função já foi declarada no mesmo escopo, percorrendo a tabela de símbolos à procura de um símbolo com o mesmo identificador do atual, caso encontre uma entrada na tabela que corresponda a essa condição, é lançado um erro semântico.

Já para a utilização de variáveis e funções, foi utilizada a pilha de escopos e a entrada correspondente ao símbolo atual na tabela de símbolos, de modo que fosse possível verificar se o identificador do escopo em que o símbolo foi utilizado possui também uma entrada na pilha de escopos, o que indicaria que a variável ou função possui definição disponível naquele escopo.

4.3 Chamada de função

Para a análise semântica das chamadas de função foram considerados os seguintes aspectos:

- Número de parâmetros: é necessário verificar se a quantidade de argumentos passados para a função é o mesmo que o determinado na declaração dela e, caso não seja, é informado o erro semântico.
- Análise dos tipos dos parâmetros e retorno: para este aspecto, é avaliado o tipo de cada parâmetro passado e também do valor retornado, de forma que, se não for o mesmo, há uma tentativa de fazer a coerção de tipos e, caso não seja possível, um erro semântico é lançado para o usuário.

4.4 Operações com listas

Para esta análise, foi utilizada a descrição da linguagem [Nal] afim de verificar como as operações com listas devem ser feitas. A partir daí, é possível dividir em operações unárias e binárias. Para o primeiro tipo de operação, é apenas considerado que, ao encontrar um operador unário, é necessário que exista logo depois um operando do tipo lista, e isso é verificado através da tabela de símbolos. Já para as operações binárias, como “map” e “filter”, foi definido que o operando à esquerda do operador pode ser uma expressão que resulte em um tipo simples e que o operando à direita seja do tipo lista. Caso essas especificações não sejam seguidas, é lançado um erro semântico.

4.5 Conversão de tipos

Para realizar a conversão dos tipos, é preciso recuperar a informação do tipo da variável da árvore sintática, para então determinar para qual tipo ela deve ser convertida. É possível apenas fazer a coerção do tipo **int** para o tipo **float** e vice-versa.

Para o tipo lista, é preciso considerar que não há como fazer a conversão entre esse tipo e inteiro ou decimal e vice-versa. Além disso, um ponto relevante é que a constante **NIL** pode ser atribuída apenas para variáveis ou funções do tipo lista.

5 Descrição dos arquivos de teste

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

Os erros apresentados para cada arquivo estão identificados abaixo:

– *wrong_ex1.c*

```
[SEMANTIC ERROR] Line: 53 | Column: 9   Variable already declared
[SEMANTIC ERROR] Line: 68 | Column: 13  Variable unavailable
```

– *wrong_ex2.c*

```
[SEMANTIC ERROR] Line: 2 | Column: 12  Variable unavailable
[SEMANTIC ERROR] Line: 2 | Column: 15  Variable unavailable
[SEMANTIC ERROR] Line: 19 | Column: 9   Variable already declared
```

6 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 -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/syntactic_tree.o src/syntactic_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/syntactic_tree.o src/obj/lex.yy.o -ll -Wall -g -I ./lib
```

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.

$\langle \text{program} \rangle$	$::= \langle \text{declarations} \rangle$
$\langle \text{declarations} \rangle$	$::= \langle \text{declarations} \rangle \langle \text{declaration} \rangle$ $\quad \langle \text{declaration} \rangle$
$\langle \text{declaration} \rangle$	$::= \langle \text{function_declaration_statement} \rangle$ $\quad \langle \text{variable_declaration_statement} \rangle$
$\langle \text{block} \rangle$	$::= \{ \langle \text{statements} \rangle \}$
$\langle \text{statements} \rangle$	$::= \langle \text{statements} \rangle \text{ ',' } \langle \text{statement} \rangle$ $\quad \langle \text{statement} \rangle$
$\langle \text{statement} \rangle$	$::= \langle \text{expression_statement} \rangle$ $\quad \langle \text{io_statement} \rangle$ $\quad \langle \text{return_statement} \rangle$ $\quad \langle \text{variable_declaration_statement} \rangle$ $\quad \langle \text{for_statement} \rangle$ $\quad \langle \text{if_else_statement} \rangle$ $\quad \langle \text{block} \rangle$
$\langle \text{function_declaration_statement} \rangle$	$::= \langle \text{SIMPLE_TYPE} \rangle \langle \text{IDENTIFIER} \rangle \text{ ' (' } \langle \text{parameters_optative} \rangle$ $\quad \text{ ') ' } \langle \text{statement} \rangle$ $\quad \langle \text{SIMPLE_TYPE} \rangle \langle \text{LIST_TYPE} \rangle \langle \text{IDENTIFIER} \rangle$ $\quad \text{ ' (' } \langle \text{parameters_optative} \rangle \text{ ') ' } \langle \text{statement} \rangle$
$\langle \text{parameters_optative} \rangle$	$::= \langle \text{empty} \rangle$ $\quad \langle \text{parameters} \rangle$
$\langle \text{parameters} \rangle$	$::= \langle \text{parameters} \rangle \text{ ',' } \langle \text{parameter} \rangle$ $\quad \langle \text{parameter} \rangle$
$\langle \text{parameter} \rangle$	$::= \langle \text{SIMPLE_TYPE} \rangle \langle \text{IDENTIFIER} \rangle$ $\quad \langle \text{SIMPLE_TYPE} \rangle \langle \text{LIST_TYPE} \rangle \langle \text{IDENTIFIER} \rangle$
$\langle \text{for_statement} \rangle$	$::= \langle \text{RW_FOR} \rangle \text{ ' (' } \langle \text{expression_optative} \rangle \text{ ' ; ' } \langle \text{or_expression_optative} \rangle$ $\quad \text{ ' ; ' } \langle \text{expression_optative} \rangle \text{ ') ' } \langle \text{statement} \rangle$
$\langle \text{if_else_statement} \rangle$	$::= \langle \text{RW_IF} \rangle \text{ ' (' } \langle \text{expression} \rangle \text{ ') ' } \langle \text{statement} \rangle$ $\quad \langle \text{RW_IF} \rangle \text{ ' (' } \langle \text{expression} \rangle \text{ ') ' } \langle \text{statement} \rangle \langle \text{RW_ELSE} \rangle$ $\quad \text{ ' (' } \langle \text{expression} \rangle \text{ ') ' } \langle \text{statement} \rangle$
$\langle \text{expression_statement} \rangle$	$::= \langle \text{expression} \rangle \text{ ' ; ' }$
$\langle \text{io_statement} \rangle$	$::= \langle \text{input_statement} \rangle$ $\quad \langle \text{output_statement} \rangle$
$\langle \text{input_statement} \rangle$	$::= \langle \text{IO_READ} \rangle \text{ ' (' } \langle \text{IDENTIFIER} \rangle \text{ ' ; ' }$
$\langle \text{output_statement} \rangle$	$::= \langle \text{IO_WRITE} \rangle \text{ ' (' } \langle \text{expression} \rangle \text{ ') ' ' ; ' }$ $\quad \langle \text{IO_WRITE} \rangle \text{ ' (' } \langle \text{LIT_STRING} \rangle \text{ ') ' ' ; ' }$
$\langle \text{return_statement} \rangle$	$::= \text{ 'return' } \langle \text{expression} \rangle \text{ ' ; ' }$

$$\begin{aligned}
\langle \text{expression} \rangle &::= \langle \text{IDENTIFIER} \rangle \text{'='} \langle \text{expression} \rangle \\
&\quad | \langle \text{or_expression} \rangle \\
&\quad | \langle \text{function_call_expression} \rangle \\
\langle \text{function_call_expression} \rangle &::= \langle \text{IDENTIFIER} \rangle \text{'('} \langle \text{function_arguments_optative} \rangle \\
&\quad \text{'}, \\
\langle \text{function_arguments_optative} \rangle &::= \langle \text{empty} \rangle \\
&\quad | \langle \text{function_arguments} \rangle \\
\langle \text{function_arguments} \rangle &::= \langle \text{function_arguments} \rangle \text{' , ' } \langle \text{function_argument} \rangle \\
&\quad | \langle \text{function_argument} \rangle \\
\langle \text{function_argument} \rangle &::= \langle \text{expression} \rangle \\
\langle \text{expression_optative} \rangle &::= \langle \text{empty} \rangle \\
&\quad | \langle \text{expression} \rangle \\
\langle \text{or_expression_optative} \rangle &::= \langle \text{empty} \rangle \\
&\quad | \langle \text{or_expression} \rangle \\
\langle \text{or_expression} \rangle &::= \langle \text{or_expression} \rangle \langle \text{LOGICAL_OP_OR} \rangle \langle \text{and_expression} \rangle \\
&\quad | \langle \text{and_expression} \rangle \\
\langle \text{and_expression} \rangle &::= \langle \text{and_expression} \rangle \langle \text{LOGICAL_OP_AND} \rangle \langle \text{equality_expression} \rangle \\
&\quad | \langle \text{equality_expression} \rangle \\
\langle \text{equality_expression} \rangle &::= \langle \text{equality_expression} \rangle \langle \text{EQUALITY_OP} \rangle \langle \text{relational_expression} \rangle \\
&\quad | \langle \text{relational_expression} \rangle \\
\langle \text{relational_expression} \rangle &::= \langle \text{relational_expression} \rangle \langle \text{RELATIONAL_OP} \rangle \langle \text{list_expression} \rangle \\
&\quad | \langle \text{list_expression} \rangle \\
\langle \text{list_expression} \rangle &::= \langle \text{list_expression} \rangle \langle \text{BINARY_LIST_OP} \rangle \langle \text{addition_expression} \rangle \\
&\quad | \langle \text{addition_expression} \rangle \\
\langle \text{addition_expression} \rangle &::= \langle \text{addition_expression} \rangle \langle \text{ARITMETIC_OP_ADDITIVE} \rangle \\
&\quad \langle \text{multiplication_expression} \rangle \\
&\quad | \langle \text{multiplication_expression} \rangle \\
\langle \text{multiplication_expression} \rangle &::= \langle \text{multiplication_expression} \rangle \langle \text{ARITMETIC_OP_MULTIPLICATIVE} \rangle \\
&\quad \langle \text{simple_value} \rangle \\
&\quad | \langle \text{simple_value} \rangle \\
\langle \text{simple_value} \rangle &::= \langle \text{constant} \rangle \\
&\quad | \langle \text{IDENTIFIER} \rangle \\
&\quad | \langle \text{ARITMETIC_OP_ADDITIVE} \rangle \langle \text{simple_value} \rangle \\
&\quad | \text{'!' } \langle \text{simple_value} \rangle \\
&\quad | \langle \text{UNARY_LIST_OP} \rangle \langle \text{simple_value} \rangle \\
&\quad | \text{'(' } \langle \text{expression} \rangle \text{')' } \\
\langle \text{variable_declaration_statement} \rangle &::= \langle \text{SIMPLE_TYPE} \rangle \langle \text{IDENTIFIER} \rangle \text{' ; ' } \\
&\quad | \langle \text{SIMPLE_TYPE} \rangle \langle \text{LIST_TYPE} \rangle \langle \text{IDENTIFIER} \rangle \\
&\quad \text{' ; ' } \\
\langle \text{constant} \rangle &::= \langle \text{C_INTEGER} \rangle | \langle \text{C_FLOAT} \rangle | \langle \text{C_NIL} \rangle
\end{aligned}$$

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	'!'