Analisador Semântico

Gabriel Cunha Bessa Vieira - 16/0120811

Universidade de Brasília

1 Motivação

Este é o relatório inicial do trabalho que será conduzido ao longo do semestre, contando apenas com o analisador léxico. O objetivo final do trabalho será reconhecer uma linguagem baseada em C denominada *C-IPL* que foi descrita em [Nal]. A implementação dessa nova linguagem terá como impacto o aprendizado de todo o processo de tradução desde linguagem de alto nível até a geração da linguagem intermediária a nível de código. Saber como ocorrem tais processos é fundamental para um cientista entender como um código pode ser otimizado visando o processo de compilação, criando, assim, um programa mais performático.

A implementação de listas a partir da nova primitiva tem por objetivo principal facilitar a manipulação de dados por meio de operadores como '?', '≫', '≪', ':', '!', '%'. A lista dá a possibilidade de usar uma estrutura robusta que ainda não existe no C. Além de ser possível manipular dados dentro do código com listas, é possível manipular a memória e determinar como os arquivos de fato são armazenados dentro do sistema operacional por meio das mesmas.

2 Descrição da análise léxica

A análise léxica tem como objetivo atribuir lexemas [Est] para a nova primitiva e todas as palavras da linguagem a partir da *regex* criada com o intuito de identificar os possíveis *tokens* [ALSU07], os quais podem ser alfanuméricos[a-z0-9], *strings*, operadores aritméticos '+', '−', '∗', '/', operadores lógicos '&&', '||', identificadores, tipos(int—float), constantes, operadores de lista(mencionados na seção anterior) e delimitadores como ';', ',', '{', '}', '(', ')'. Quando ocorre um comportamento não esperado, é produzido uma linha de erro que aponta no formato '**(linha:coluna) lexema**' juntamente com a descrição do erro.

Para realizar a separação de identificadores e operadores foram utilizadas macros que têm o nome análogo à sua expressão regular. Tudo foi feito de acordo com a estrutura padrão do Flex [Est], que são definições, seguidas de suas regras e por fim o código que será exportado para o .c gerado a partir do arquivo Flex.

A tabela de símbolos está implementada utilizando listas encadeadas. Já os tokens descritos na estrutura dos tokens no Apêndice B serão armazenados por meio de uma estrutura.

3 Descrição da análise sintática

A análise sintática que foi baseada na gramática dada em [Gup21] tem como objetivo verificar se a sequência de tokens corresponde a uma ordem válida de unidades significativos da linguagem, que foram lidos na análise léxica, constituindo parte da gramática proposta no Apêndice A.

A análise sintática foi feita utilizando o Bison [Cor21], um analisador sintático que utiliza a gramática livre do contexto que foi fornecida, com o intuito de construir um autômato com pilha que reconhece a linguagem. Nessa parte do projeto foi recomendada a utilização da flag:

```
\%define lr.type canonical-lr
```

O nome do arquivo em questão é o **sintático.y**. No arquivo em questão é possível decorrer por toda gramática livre do contexto presente no Apêndice A.

A estrutura do **sintatico.y** está dividida três seções distintas.

- Cabeçalho
  - Aqui são declaradas as primitivas necessárias no decorrer do trabalho, juntamente com o código C inicial que será gerado pelo Bison no arquivo "sintatico.tab.c"e "sintatico.tab.h". Além disso têm a estrutura de tokens que é usada no analisador léxico para passar os argumentos que serão usados no sintático usando a variável "**yylval**", passando o número de linha, columa e o token lido.
- Gramática
  - Essa seção contém a gramática A, com suas respectivas derivações e regras usando os tokens que são obtidos a partir do analisador léxico.

3.1 Implementação da árvore sintática

Na implementação da árvore sintática como temos na figura 1, temos a criação de uma estrutura onde são implementados cinco nós filhos (o máximo que uma regra guarda de tokens e tipos na gramática proposta são cinco) e a respectiva regra que foi encontrada. Com as informações descritas pela estrutura, percorrer a árvore em profundidades distintas a partir do nó raiz se torna possível e todos os detalhes descritos pela estrutura são mostrados corretamente.

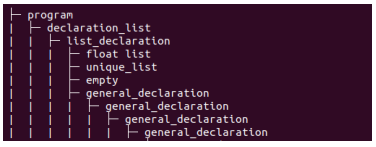

Figura 1. Arvore sintática

3.2 Implementação da Tabela de Símbolos

Na implementação da tabela de símbolos como visto na figura 2, são armazenados símbolos que são classificados como variável, nome de função, escopo. A medida que o analisador léxico varre o programa, são identificados os *tokens* que estão presentes na linguagem. Para o armazenamento é criada uma estrutura com linha, coluna, tipo de função, escopo, corpo com o que é lido pelo léxico e uma variável que verifica se o que foi lido é uma função ou não.

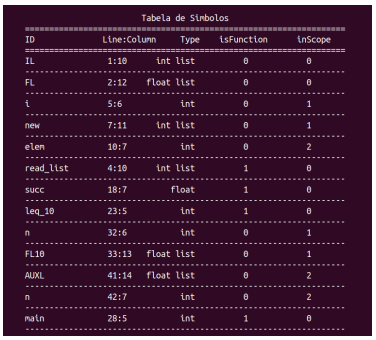

Tabela de Simbolos

ID	Line:Column	Type	isFunction	inScope
IL	1:10	int list	0	0
FL	2:12	float list	0	0
l	5:6	int	0	1
new	7:11	int list	0	1
elen	10:7	int	0	2
read_list	4:10	int list	1	0
succ	18:7	float	1	0
leq_10	23:5	int	1	0
n	32:6	int	0	1
FL10	33:13	float list	0	1
AUXL	41:14	float list	0	2
n	42:7	int	0	2
main	28:5	int	1	0

Figura 2. Tabela de símbolos

4 Descrição da análise semântica

Para a análise semântica foi utilizado o Bison [Cor21] e o Flex [Est] e todas as estruturas produzidas nas análises léxica e sintática, que foram a tabela de símbolos, árvore sintática e a pilha de escopos.

Nessa análise o intuito é verificar se um trecho de código é escrito de uma forma correta e tenha uma certa coerência. Nas próximas seções é descrito o que um analisador semântico necessita verificar.

4.1 Verificação de escopo

Durante a análise do código inúmeros escopos são declarados, seja em início de função, condicional ou em *loops* de repetição. Para estabelecer quais escopos podem ser acessados, temos a pilha de escopos. Com a pilha podemos verificar se uma variável foi declarada ou não, se temos variáveis duplicadas no mesmo escopo.

Na pilha de escopo há uma incrementação quando é encontrado um '{' e uma decrementação quando encontra-se um '}'.

4.2 Variáveis duplicadas e/ou não declaradas

Para fazer essa verificação há uma manipulação de nós da árvore, na tabela de símbolos e na pilha. Por parte dos nós da árvore, é feita uma busca em profundidade para achar determinados tipos de regras que são passíveis de atribuição como '*expression*'.

Feito isso são feitas verificações na pilha e na tabela de símbolos para ver se o operador sendo usado na atribuição ou na expressão aritmética já foi declarado no escopo atual ou em algum anterior que se comporte como um escopo pai.

Com isso é possível verificar quais variáveis do código foram redeclaradas mais de uma vez, e quais variáveis não foram inicializadas no escopo em questão.

4.3 Parâmetros e argumentos de funções

Para realizar a busca por parâmetros e argumentos dentro das funções foi realizada uma busca em profundidade nos nós da árvore com o intuito de obter o '*params*' de dentro da árvore e obter seus respectivos nós adjacentes que são as variáveis passadas como parâmetro dentro da função.

Em seguida é verificado na tabela de símbolos se tem algum argumento da função que não foi chamado, se tem algum argumento com tipo diferente ou até mesmo se a lista de parâmetros passados na função são maiores que o necessário.

4.4 Conversão de tipos

Para realizar a conversão em variáveis de tipos distintos é necessário verificar se a conversão é necessária naquele contexto. Caso se faça necessário, executar a conversão e depois conferir se os tipos das variáveis são diferentes.

Para fazer essa verificação, é necessário fazer uma busca em profundidade na árvore. Cada nó da árvore tem informações a respeito do seu tipo. Quando temos uma diferenciação de tipos é onde fazemos um casting, por exemplo em uma operação aritmética em ***float*** para uma variável que é declarada como ***int*** anteriormente. Abaixo segue uma lista de restrição para a conversão de tipos:

- Variáveis do tipo **int** podem ser convertidas para **float** e vice-versa.
- Operações lógicas e relacionais sempre retornam o tipo **int**;
- Listas podem ser do tipo **int** ou *float*; Uma vez que valores diferentes de seu tipo usual são colocados, é feita uma conversão de todos os elementos da lista;
- Não é possível utilizar operações aritméticas com listas inteiras, apenas com os seus elementos;
- A constante **NIL** só pode ser utilizada para inicialização de listas, ou seja, variáveis do tipo **int list** ou **float list**;

5 Variáveis de ambiente e versões

5.1 Compilação

Para compilar, é necessário estar na raiz e executar o seguinte comando:

```
$ make all
```

Flags usadas na compilação:

```
bison -d -Wother -o src/sintatico.tab.c src/sintatico.y
flex -o src/lex.yy.c src/lexico.l
gcc-11 -g -Wall -I lib -o tradutor src/sintatico.tab.c src/lex.yy.c
src/tabela.c src/arvore.c -ll
```

Para executar algum programa é necessário entrar o seguinte comando:

```
$ ./tradutor tests/<nome_do_programa>.c
```

Os arquivos disponibilizados serão os seguintes:

1. teste_correto1.c
2. teste_correto2.c
3. teste_errado1.c
4. teste_errado2.c

São fornecidos dois arquivos teste para cada caso que esteja correto ou errado. Com o **teste_errado1.c** contendo erro na linha 2, coluna 2 (2|2). E o **teste_errado2.c** contendo erro na linha 4, coluna 11 (4|11) retratados a seguir:

- *TesteErrado_1.c*

```
(2|2) Syntax error: syntax error, unexpected ID,
expecting end of file or SIMPLE_TYPE
(4:6) 'a' - Has already been declared on this scope -> [1]
(6:2) Unidentified character: '$'
(9:5) 'i' - Has already been declared on this scope -> [0]
(11|6) Syntax error: syntax error, unexpected BINARY_BASIC_OP2, expecting LIST_TYPE or I
(12|12) Syntax error: syntax error, unexpected BINARY_CONSTRUCTOR, expecting '(' or ';'
```

- *TesteErrado_2.c*

```
(3:11) 'i' - Has already been declared on this scope -> [1]
(4|11) Syntax error: syntax error, unexpected BINARY_BASIC_OP2,
expecting LIST_TYPE or ID
(6:11) 'b' - Has already been declared on this scope -> [1]
(8:5) Unidentified character: '#'
(10|8) Syntax error: syntax error, unexpected '=', expecting '(' or ';'
'Main' function not detected.
```

5.2 Software Utilizado

- Ubuntu LTS 20.04
- Flex 2.6.4
- Gcc 11.2.1
- GNU Make 4.3
- Bison 4.7.5
- Kernel 5.11.0-27-generic

Referências

[ALSU07] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, & Tools*. Pearson/Addison Wesley, 2nd edition, 2007. Online; acessado 10 de Agosto de 2021.

[Cor21] R Corbett. Gnu bison - the yacc-compatible parser generator. https://www.gnu.org/software/bison/manual/, Online; acessado 2 de Setembro de 2021.

[Est] W. Estes. Flex: Fast lexical analyser generator. https://github.com/westes/flex. Online; acessado 10 de Agosto de 2021.

[Gup21] A Gupta. The syntax of c in backus-naur form. https://tinyurl.com/max5eep, Online; acessado 10 de Agosto de 2021.

[Nal] C Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Online; acessado 10 de Agosto de 2021.

A Gramática

1. *program* → *declaration_list*
2. *declaration_list* → *declaration_list declaration*
   | *declaration*
3. *declaration* → *var_declaration*
   | *function_declaration*
   | *list_declaration*
4. *var_declaration* → *SIMPLE_TYPE ID ';'*
5. *function_declaration* → *SIMPLE_TYPE ID '(' params ')' '{' multiple_stmt '}'*
   | *SIMPLE_TYPE LIST_TYPE ID '(' params ')' '{' multiple_stmt '}'*
6. *list_declaration* → *SIMPLE_TYPE LIST_TYPE ID ';'*
7. *params* → *params ',' param*
   | *param*
   | ϵ
8. *param* → *SIMPLE_TYPE ID*
   | *SIMPLE_TYPE LIST_TYPE ID*
9. *if_stmt* → *IF '(' expression ')' '{' multiple_stmt '}'*
10. *if_else_stmt* → *IF '(' expression ')' '{' multiple_stmt '}' ELSE '{' multiple_stmt '}'*
    | *IF '(' expression ')' '{' multiple_stmt '}' ELSE stmt*
11. *for_stmt* → *FOR '(' for_variation_null_expressions ';' for_variation_null_expressions ';' for_variation_null_expressions ')' '{' multiple_stmt '}' ELSE stmt*
12. *return_stmt* → *RETURN ';'*
    | *RETURN expression ';'*
13. *general_declaration* → *general_declaration var_declaration*
    | *general_declaration list_declaration*
    | *general_declaration stmt*
    | ϵ
14. *multiple_stmt* → *general_declaration*
15. *expression_stmt* → *expression ';'*
16. *expression* → *ID '=' expression*
    | *simple_expression*
    | *binary_construct*
    | *list_operation*
17. *for_variation_null_expressions* → *expression*
    | ϵ
18. *stmt* → *expression_stmt*
    | *if_stmt*
    | *if_else_stmt*
    | *for_stmt*
    | *return_stmt*
    | *print*
    | *scan*

19. *simple_expresion* → *relational_expression*
    | *arithmetic_expression LOGIC_OP arithmetic_expression*
20. *relational_expression* → *relational_expression BINARY_COMP_OP arithmetic_expression*
    | *arithmetic_expression*
21. *arithmetic_expression* → *arithmetic_expression BINARY_BASIC_OP1 term*
    | *BINARY_BASIC_OP1 term*
    | *BINARY_COMP_OP term*
    | *TAIL term*
    | *term*
22. *term* → *term BINARY_BASIC_OP2 factor*
    | *factor*
23. *factor* → *'(' expression ')'*
    | *ID*
    | *INT*
    | *FLOAT*
    | *ID '(' expression ')'*
    | *LIST_CONSTANT*
    | *HEADER ID*
24. *print* → *OUTPUT '(' STRING ')' ';'*
    | *OUTPUT '(' expression ')' ';'*
25. *scan* → *INPUT '(' ID ')' ';'*
26. *binary_construct* → *binary_construct_recursive BINARY_CONSTRUCTOR ID*
27. *binary_construct_recursive* → *binary_construct_recursive BINARY_CONSTRUCTOR ID*
    | *ID*
28. *list_operation* → *recursive_list_operation MAP ID*
    | *recursive_list_operation FILTER ID*
29. *recursive_list_operation* → *recursive_list_operation MAP ID*
    | *recursive_list_operation FILTER ID*
    | *ID*

B Estrutura dos Tokens

Token	Lexema	Expressão Regular
<int, >	12	DIGIT+
<float, >	12.5	DIGIT+"."DIGIT+
<list_constant, >	NIL	NIL
<digit, >	2	[0-9]
<string, >	"blablabla"	(\"([^\"\']) * \")
<id, >	nome_generico	$[a-zA-Z\_][a-z0-9A-Z\_]*$
<binary_basic_op, >	+	[+*/-]
<logic_op, >	&&	|||&&
<binary_comp_op, >	$\leq$	
<input, >	read	-
<output, >	write|writeln	-
<binary_constructor, >	:	:
<header, >	?	?
<tail, >	%	"!"|"%"
<map, >	$\gg$	"$\gg$"
<filter, >	$\ll$	"$\ll$"