Análise Léxica da Linguagem C-IPL

Lucas Dalle Rocha
17/0016641

Universidade de Brasília - UnB
170016641@aluno.unb.br

1 Motivação

A disciplina propõe a implementação das etapas de análise léxica, sintática, semântica e de geração de código intermediário para um subconjunto da linguagem C, denominado *C-IPL*, com o intuito de ressaltar a importância o processo de tradução de uma linguagem em alto nível para linguagem de máquina, bem como o impacto do desenvolvimento do projeto no sistema como um todo. Assim, pela linguagem *C-IPL* busca-se facilitar a manipulação de listas para programas escritos em C, visto que são utilizadas frequentemente para armazenamento de dados em sequência.

2 Descrição

2.1 Análise léxica

A etapa de análise léxica visa reconhecer e classificar lexemas válidos para dada linguagem, além de estabelecer a tabela de símbolos para satisfazer a comunicação com o analisador sintático.

Dito isso, houve o uso de variáveis com a finalidade de exibir a linha e coluna em que o lexema analisado se encontra, bem como expressões regulares para buscar padrões que condizem com a descrição da linguagem *C-IPL*, de modo a atribuir *tokens* válidos para identificadores, dígitos, tipos de dado, palavras reservadas, números inteiros e reais, operadores aritméticos, relacionais, lógicos, e as operações sobre listas, implementadas pela linguagem *C-IPL*. Ademais, comentários não são tratados como lexemas, *strings* são sinalizadas como um único *token* e os símbolos de pontuação pareada também serão lexemas.

Dessa forma, quaisquer *lexemas* que não foram representados pelas expressões regulares são considerados erros léxico, assim como comentários multilinhas e *strings*, ambos não finalizados. Por fim, delimitadores como ponto e vírgula serão aprofundados na etapa de análise sintática, em conjunto com a tabela de símbolos, que será composta pelos atributos de identificadores (nome e tipo), além das suas respectivas posições em linha e coluna de código, definição das declarações e escopo.

2.2 Análise sintática

Já a etapa de análise sintática enfoca a sintaxe de uma gramática, isto é, a validez de uma sentença escrita na linguagem *C-IPL*. Para isso, foi-se utilizado o gerador de analisador sintático ascendente *Bison*, pelo padrão LR(1) canônico, de modo que o analisador léxico comunica-se com o analisador sintático através de *tokens* válidos da linguagem e, esse último, por sua vez, utiliza os *tokens* recebidos para esquematização de uma árvore sintática abstrata, que é a estrutura utilizada para validar a sintaxe em nossa gramática.

Desse modo, para cada regra válida da gramática, dado o conjunto de *tokens*, é criado um nó na estrutura da árvore, e para um terminal, é atribuído à estrutura da árvore esse conjunto de *tokens*, a fim de que possam ser impressos igualmente. Portanto, a estrutura utilizada para os nós da árvore possui o nome do próprio nó, ponteiros para seus nós filhos e, como supracitado, o conjunto de *tokens*.

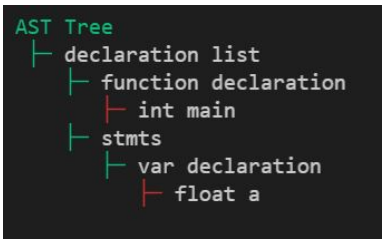


Figura 1. Árvore sintática abstrata de *int main ()*{*float a;*}, de acordo com o apêndice B.

Não obstante, a tabela de símbolos é uma estrutura implementada em conjunto das análises léxica e sintática, de modo que a primeira se ocupa em atribuir a posição do *token* em código (linha e coluna), além do escopo inicial, baseado nos *tokens* '{' e '}'. Já a análise sintática contribui para a construção da tabela de símbolos com a concatenação de *tokens* que especificam tipo e identificador, em nossa gramática, definição do tipo de declaração (variável ou função) e a devida atualização do escopo, haja vista símbolos já existentes na tabela. Assim, um símbolo possui os atributos: nome, tipo, tipo da declaração, escopo, linha e coluna de ocorrência, e são armazenados dinamicamente em uma lista encadeada.

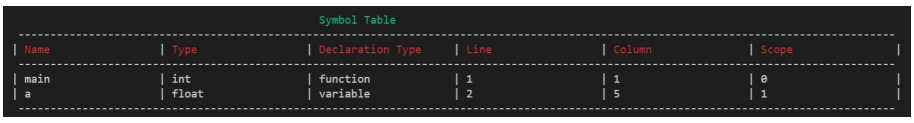
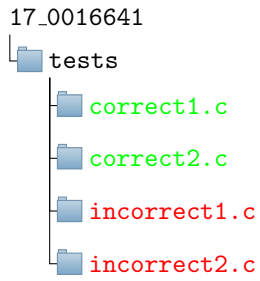


Figura 2. Tabela de símbolos de *int main ()*{*float a;*}, de acordo com o apêndice B.

3 Testes

Existem dois arquivos de teste que se encontram corretos (representados pela cor verde abaixo), de acordo com a análise sintática, uma vez que não apresentam sintaxes irreconhecíveis pela gramática da linguagem *C-IPL*. Em contrapartida, existem outros dois arquivos de teste que possuem erros (representados pela cor vermelha abaixo), que serão listados abaixo. Todos os testes encontram-se na pasta *tests*.



1. incorrect1.c apresenta erros de sintaxe não reconhecidas pela gramática da linguagem, como atribuição de valores dentro da função *if* (linha 2, coluna 6) - em que se espera uma expressão - e comando não finalizado com ponto e vírgula (linha 4, coluna 1), uma vez que se esperava o *token* ';', e não '}'.
2. incorrect2.c similarmente possui sintaxe não reconhecidas pela *C-IPL*, tais como função de escrita atribuída a função de lista *map* '>>' (linha 7, coluna 16) - em que se espera uma expressão matemática de soma, no mínimo - e atribuição de valores dentro da função *for* (linha 10, coluna 21), uma vez que se esperava uma expressão simples.

4 Instruções para compilação e execução

Para a compilação e execução, foi utilizado o sistema operacional Ubuntu 20.04.2 LTS, além do *gcc* versão 11.1.0, *flex* versão 2.6.4, *bison* versão 3.7.4 e GNU Make 4.2.1. O programa pode ser compilado pela utilização do *Makefile*, com o seguinte comando em seu terminal:

```
$ make
```

Ademais, pode-se compilar e executar o *Valgrind* para checagem de memória dos arquivos de teste, em conjunto com a utilização do *Makefile*:

```
correct1.c   -> $ make valgrind1
correct2.c   -> $ make valgrind2
incorrect1.c -> $ make valgrind3
incorrect2.c -> $ make valgrind4
```

Caso seja de seu interesse compilá-lo manualmente, execute a sequência de instruções:

```
$ bison -o src/syntax.tab.c -d src/syntax.y -Wcounterexamples
$ flex -o src/lex.yy.c src/lexical.l
$ gcc-11 -g -c src/structures.c -o obj/structures.o
$ gcc-11 -g src/syntax.tab.c src/lex.yy.c obj/structures.o
-I lib -o tradutor -Wall -Wpedantic
```

Por fim, para execução do analisador sintático nos arquivos teste:

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

Referências

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2nd edition, 2006.

[Hec21] Robert Heckendorn. A grammar for the C- programming language. *http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf*, 2021. [Online; Last accessed 19-August-2021].

[Lee85] Jeff Lee. ANSI C Yacc grammar. *https://www.lysator.liu.se/c/ANSI-C-grammar-y.html*, 1985. [Online; Last accessed 2-September-2021].

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., 2nd edition, 1992.

[PEM16] Vern Paxson, Will Estes, and John Millaway. Lexical analysis with flex, for flex 2.6.2. *https://westes.github.io/flex/manual/*, 2016. [Online; Last accessed 19-August-2021].

A Estruturação do *token*

Para estruturação do *token*, isto é, seus padrões, nomes e atributos, foi utilizado como referência a figura 3.12 do livro-texto [ALSU06], abordado na página 130, e atualizado para a análise sintática.

Lexema	**Nome do *token***	**Atributo**
Espaços em branco	–	–
Qualquer *id*	**ID**	Ponteiro p/ tabela de símbolos
Qualquer *integer_number*	**INTEGER**	Ponteiro p/ tabela de símbolos
Qualquer *real_number*	**REAL**	Ponteiro p/ tabela de símbolos
if	**IF**	–
else	**ELSE**	–
for	**FOR**	–
return	**RETURN**	–
read	**INPUT**	–
write, writeln	**OUTPUT**	–
NIL	**NIL_CONSTANT**	–
int, float	**SIMPLE_TYPE**	–
list	**COMPOUND_TYPE**	–
<, <=, >, >=, ==, ! =	**RELATIONAL**	–
+, -	**ADD**	–
*, /	**MUL**	–
||	**LOGICAL_OR**	–
&&	**LOGICAL_AND**	–
:	**LIST_CONSTR**	–
?	**LIST_OP**	–
%	**LIST_DESTR**	–
>>, <<	**LIST_FUNC**	–
!	**OP_OVERLOAD**	–

Tabela 1. Padrões de *tokens*, seus nomes e atributos.

B Gramática da linguagem

Para a montagem da gramática, foi utilizado como referência [Hec21], a fim de estimar a transição da análise léxica para a sintática. As adaptações foram feitas para que a gramática seja compatível com a linguagem *C-IPL* e correções para a análise sintática.

1. *initial* → *declaration_list*
2. *declaration_list* → *declaration_list decl* | *decl*
3. *decl* → *var_declaration* | *func_definition*
4. *var_declaration* → **SIMPLE_TYPE ID**; | **SIMPLE_TYPE COMPOUND_TYPE ID**;
5. *var_definition* → **SIMPLE_TYPE ID** | **SIMPLE_TYPE COMPOUND_TYPE ID**
6. *func_definition* → *func_declaration* (*params.opt*) *stmts*
7. *func_declaration* → **SIMPLE_TYPE ID** | **SIMPLE_TYPE COMPOUND_TYPE ID**
8. *params.opt* → *params* | ϵ
9. *params* → *param_list*
10. *param_list* → *param_list*, *param* | *param*
11. *param* → *var_definition*
12. *stmts* → {*stmt_list.opt*}
13. *stmt_list.opt* → *stmt_list* | ϵ
14. *stmt_list* → *stmt_list stmt.item* | *stmt.item*
15. *stmt.item* → *stmt* | *var_declaration*
16. *stmt* → *return_stmt* | *select_stmt* | *iter_stmt* | *io_stmt* | *exp_stmt* | *stmts*
17. *select_stmt* → **IF** (*simple_exp*) *stmt* | **IF** (*simple_exp*) *stmt* **ELSE** *stmt*
18. *iter_stmt* **FOR** (*assign_exp*; *simple_exp*; *exp*) *stmt*
19. *exp_stmt* → *assign_exp* | *simple_exp*
20. *return_stmt* → **RETURN** *exp*;
21. *io_stmt* → *in_stmt* | *out_stmt*
22. *in_stmt* → **INPUT** (**ID**);
23. *out_stmt* → **OUTPUT** (**STRING**); | **OUTPUT** (*exp*);
24. *assign_exp* → **ID** = *exp*
25. *simple_exp* → *simple_exp* **LOGICAL_OR** *and_exp* | *and_exp*
26. *and_exp* → *and_exp* **LOGICAL_AND** *rel_exp* | *rel_exp*
27. *rel_exp* → *rel_exp* **RELATIONAL** *list_exp* | *list_exp*
28. *list_exp* → *sum_exp* **LIST_CONSTR** *list_exp* | *sum_exp* **LIST_FUNC** *list_exp* | *sum_exp*
29. *sum_exp* → *sum_exp* **ADD** *mul_exp* | *mul_exp*
30. *mul_exp* → *mul_exp* **MUL** *unary_exp* | *unary_exp*
31. *unary_exp* → *factor* | **LIST_OP** *unary_exp* | **LIST_DESTR** *unary_exp* | **OP_OVERLOAD** *unary_exp* | **ADD** *unary_exp*
32. *factor* → **ID** | (*simple_exp*) | *func_call* | *constant*
33. *func_call* → **ID** (*func_params*)
34. *func_params* → *simple_exp* | *func_params*, *simple_exp* | ϵ
35. *constant* → **INTEGER** | **REAL** | **NIL_CONSTANT**

C Léxico da linguagem

Para a esquematização do léxico em expressões regulares, foi utilizado como referência o livro a respeito do analisador léxico [LMB92], bem como o manual do flex [PEM16], disponibilizado *online*.

Macro do token	**Definição *regex***
comment_line	"//".*
delim	[\t\r]
ws	[delim]+
newline	[\n]
brace_opening & brace_closing	"{" "}"
parenthese_opening & parenthese_closing	"(" ")"
letter	[A-Za-z_]
digit	[0-9]
keywords	"if"|"else"|"for"|"return"
data_type	"int list"|"int"|"float list"|"float"
input_command	"read"
output_command	"write"|"writeln"
nil_constant	"NIL"
id	{letter}({letter}|{digit})*
integer_number	-?{digit}+
real_number	-?{integer_number}(\.{digit}+)(E[+-]?{digit}+)?
arithmetic_operators	"+"|"-"|"*"|"/"
relational_operators	"<"|"<="|">"|">="|"=="|"! ="
logical_operators	"||"|"&&"
assignment	"="
separator	","
flow_control	";"
list_constructor	":"
list_operator	"?"
list_destructor	"%"
list_functions	">>"|"<<"
string_literal	\"([^"\\\n]|\\.)*\"
operator_overload	!

Tabela 2. Tabela de lexemas e suas regras.