Tradutor da Linguagem C-IPL

Pedro Vitor Valença Mizuno - 17/0043665

Universidade de Brasília, DF, Brasil

1 Motivação

Linguagens de programação são formas de se descrever, por meio de termos, uma computação de forma mais simples e clara para pessoas. No entanto, para que os comandos contidos nessa linguagem sejam compreendidos por uma máquina, é necessária a utilização de um tradutor, um programa que traduz as linhas de código para um formato capaz de ser executado pelo computador. [ALSU06]

Como projeto da disciplina Tradutores, foi proposta a criação de um tradutor para a linguagem C-IPL, uma linguagem de programação baseada em C. O C-IPL apresenta como seu maior diferencial a existência da primitiva *list*, que permite criar uma lista de inteiros ou números de ponto flutuante. A primitiva *list* é uma ferramenta poderosa devido ao fato de criar uma estrutura de dados em formato de lista que permite adicionar e remover elementos de forma simples. Além disso, as operações map e filter (>> e <<, respectivamente) geram uma forma efetiva de operar com funções sobre as listas utilizadas.

2 Descrição

A análise léxica consiste na primeira fase do tradutor, em que caracteres são casados com um padrão definido por expressões regulares, sendo classificado como um *token*. Como saída, o analisador léxico produz um *token*, o qual é composto por um nome e um valor, no formato (nome-token, linha, coluna).

Por meio do programa *FLEX* [EP01], são criados autômatos a partir das expressões regulares. Por meio desses autômatos, é possível reconhecer os lexemas e classificá-los a partir do *token* gerado. Na primeira fase do projeto, ao ocorrer uma correspondência entre lexema e expressão regular, era impresso o *token* correspondente, assim como a linha e coluna onde ela ocorreu, no entanto, durante a realização da segunda etapa do projeto, esse processo foi substituído pelo retorno do token gerado. Porém, ainda se mantém que caso ocorra um erro léxico, o lexema é anunciado a partir de uma mensagem que aponta a linha e coluna onde houve o erro. Além disso, foi mantido também o tratamento de dois erros em específico, o comentário em múltiplas linhas sem fechamento e a string em múltiplas linhas.

A segunda etapa do projeto da disciplina consiste em implementar a análise sintática. Essa fase do tradutor recebe os *tokens* obtidos pelo analisador léxico, que representam o programa fonte. Então, é verificado se as cadeias de *token* correspondem a alguma regra da gramática [ALSU06]. Adicionalmente, a análise

sintática tem como responsabilidade gerar a árvore sintática abstrata e, opcionalmente, a tabela de símbolos, visto que também pode ser montada pela análise léxica.

Durante a segunda fase do projeto, foi utilizado o programa *Bison* [Cor14] para ser feita a análise sintática do tradutor. Para tal, foi utilizado o padrão LR(1) canônico e foi adaptada a gramática apresentada no apêndice B. Como citado anteriormente, o formato dos *tokens* retornados pelo analisador léxico consiste em uma *struct* que contém nome-token, linha e coluna, que são o lexema que representa o *token*, a linha e coluna onde ele foi encontrado, respectivamente. A partir dos tokens retornados e as regras que fazem parte da gramática, são feitas reduções para identificar se a entrada é aceita ou não pela linguagem C-IPL. Caso ocorra seja encontrado alguma construção recusada pela gramática, é destacado a linha e coluna onde ela ocorreu.

Para a criação da árvore sintática abstrata, foi definido que o tipo das reduções da gramática é a *struct* que define os nós da árvore e, ao fim de uma produção, é criado um nó da árvore. O nó consiste em uma *struct* composta por um nome, que o identifica, e três ponteiros para nós filhos, devido ao fato de que a derivação do *if-else*, que mais possui não-terminais, apresenta três deles. A fim de simplificar a árvore sintática gerada, foi adotada a ideia de passar adiante o ponteiro do nó caso seja uma regra composta por apenas um não-terminal ou similar, como em: $expLog \rightarrow expArit$.

A tabela de símbolos consiste em uma estrutura de dados usada pelo tradutor para guardar informações acerca das variáveis e funções que venham a aparecer no código a ser traduzido. Para tal, foi criada uma lista encadeada em que cada elemento consiste em uma *struct* composta por: símbolo, que identifica o termo, varOuFunc, que define se é uma variável ou função, tipo, que contém o tipo do termo, valor, que armazenará o valor atrelado a variável, numArgs, que armazena o número de argumentos da função, escopo, que identifica o escopo do termo, linha e coluna, que identificam o local onde foi declarado, e o ponteiro para o próximo elemento da lista. Foi também necessário criar uma lista que auxilia o processo de definir o escopo presentes nos programas. Nas figuras 1 e 2 são exemplificadas a árvore sintática e a tabela de símbolos de um exemplo.

3 Arquivos Teste

A fim de testar as funcionalidades do analisador léxico, foram criados quatro arquivos de entrada em linguagem C-IPL, localizados no diretório *tests*, para serem avaliados pelo programa:

1. *teste1_correto.ci*: programa sem erros sintáticos que recebe uma lista e retorna se a soma dos elementos contidos nessa lista é positiva ou negativa;
2. *teste2_correto.ci*: programa sem erros sintáticos que monta em uma lista a sequência de *Fibonacci* até uma posição n;
3. *teste1_errado.ci*: programa com erros sintáticos que calcula e apresenta a classificação do IMC de acordo com os dados inseridos. Os erros presentes são a falta de ponto e vírgula na linha 6 e coluna 5, o tipo errado na linha 21

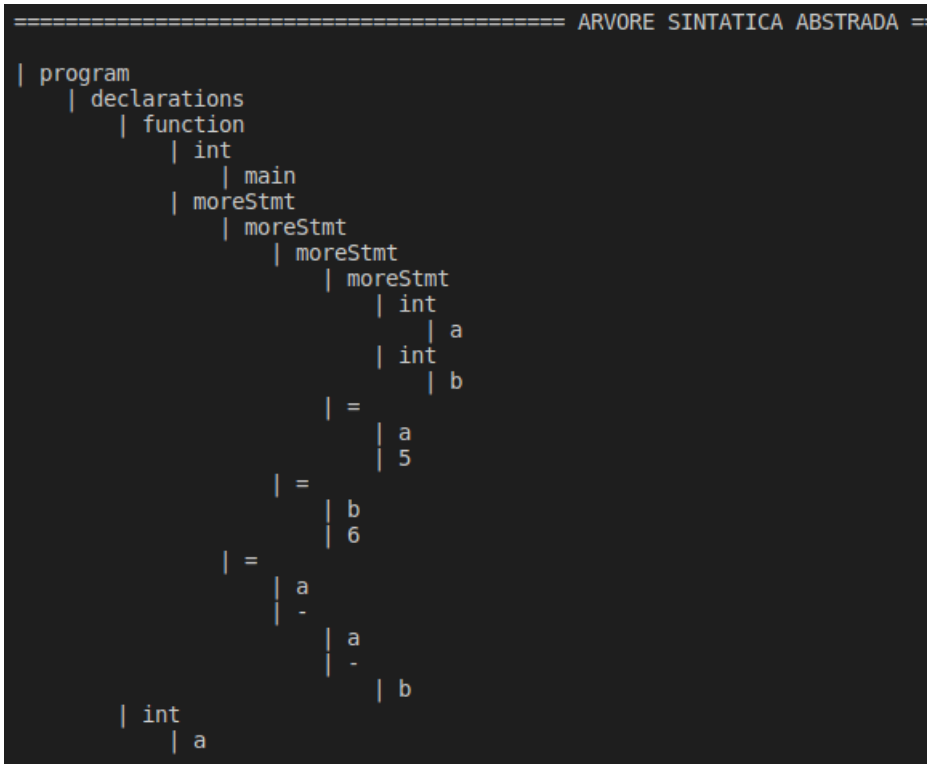

Figura 1. Árvore sintática abstrata gerada em um exemplo.

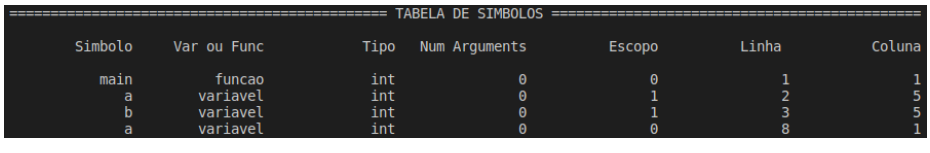

Figura 2. Tabela de símbolos gerada em um exemplo.

e coluna 14, e problemas de atribuição na linha 27 e coluna 32, assim como na linha 32 e coluna 20;

4. *teste2_errado.ci*: programa com erros sintáticos que encontra o menor número de ponto flutuante de uma lista. Os erros presentes são a função declarada com tipo errado na linha 1 e coluna 1, o número .9 após o número 9999.9 na linha 3 e coluna 18, o símbolo > na linha 17 e coluna 14 e o *for* com apenas uma expressão na linha 25 e coluna 18.

4 Instrução de Compilação

Versões das ferramentas utilizadas:

- Ubuntu 21.04
- FLEX 2.6.4
- GCC 11.1.0
- Make 4.3
- Kernel 5.11.0-25-generic
- Bison (GNU Bison) 3.7.5

Para compilar o programa no diretório do projeto, é necessário executar os seguintes comandos:
bison –defines=./src/sintaxe.tab.h –output=./src/sintaxe.tab.c -v ./src/sintaxe.y
flex ./src/lexico.l
gcc ./src/sintaxe.tab.c ./src/lex.yy.c ./src/arvore.c ./src/lista.c -Wall -o tradutor -lfl
Para a facilitar a compilação do analisador léxico criado, foi criado um *makefile* que sintetiza os passos de compilação em apenas um comando. Por meio de um terminal, no diretório do projeto (i.e. 17_0043665), onde está o arquivo *makefile*, execute o comando: **make**. Compilado o projeto, basta apenas executá-lo com o comando: **./tradutor tests/nome_do_arquivo.ci**.

Referências

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 75 Arlington Street Suite 300 Boston MA 02116 USA, 2 edition, 2006.

[Cor14] Robert Corbett. Gnu bison. https://www.gnu.org/software/bison/, 2014. [Accessed 02-September-2021].

[EP01] Will Estes and Vern Paxson. The fast lexical analyzer - scanner generator for lexing in C and C++. https://github.com/westes/flex, 2001. [Accessed 08-August-2021].

A Léxico

Tabela 1. Tabela que apresenta o léxico da linguagem C-IPL.

Lexema	Nome do Token	Valor do Atributo
Espaço	-	-
Comentário	-	-
if	***if***	-
else	***else***	-
for	***for***	-
return	***return***	-
read	***ENTRADA***	-
write	***SAIDA***	-
writeln	***SAIDA***	-
Identificador	***ID***	Apontador para a entrada da tabela
Constante Int	***CONST_INT***	Apontador para a entrada da tabela
Constante Float	***CONST_FLOAT***	Apontador para a entrada da tabela
String	***STRING***	Apontador para a entrada da tabela
NIL	***NIL***	NIL
int	***TIPO***	int
float	***TIPO***	float
list	***LIST***	list
+	***ARIT_OP_MAIS***	+
-	***ARIT_OP_MENOS***	-
*	***ARITOP_ALTA***	*
/	***ARITOP_ALTA***	/
&&	***LOG_OP_E***	&&
||	***LOG_OP_OU***	||
!	***LOG_OP_UN***	!
<=	***RELOP_ALTA***	<=
<	***RELOP_ALTA***	<
>=	***RELOP_ALTA***	>=
>	***RELOP_ALTA***	>
==	***RELOP_BAIXA***	==
! =	***RELOP_BAIXA***	! =

Tabela 2. Segunda parte da tabela que apresenta o léxico da linguagem C-IPL.

Lexema	Nome do Token	Valor do Atributo
?	***LIST_OP_UN***	?
!	***LIST_OP_UN***	!
%	***LIST_OP_UN***	%
:	***LIST_OP_BIN***	:
>>	***LIST_OP_BIN***	>>
<<	***LIST_OP_BIN***	<<
(	(	-
)	)	-
{	**{**	-
}	**}**	-
,	***VIRG***	-
;	***PV***	-

B Gramática

A gramática utilizada no tradutor da linguagem C-IPL é apresentada nas seguintes expressões:

1. *program* → *declarations* | ε
2. *declarations* → *declarations declaration* | *declaration*
3. *declaration* → *function* | *variable*;
4. *function* → *funcDecl* **(***parameters***)** {*moreStmt*} | *funcDecl* **()** {*moreStmt*}
5. *funcDecl* → **TIPO ID** | **TIPO LIST ID**
6. *parameters* → *parameters***,** *varDecl* | *varDecl*
7. *moreStmt* → *moreStmt stmt* | *stmt*
8. *stmt* → *oneLineStmt* | *multLineStmt*
9. *multLineStmt* → *conditional* | *iteration*
10. *conditional* → **if** (*attribuition*) *bracesStmt* | **if** (*attribuition*) *bracesStmt* **else** *bracesStmt*
11. *bracesStmt* → {*moreStmt*} | *oneLineStmt*
12. *iteration* → **for**(*iteArgs*) *bracesStmt*

13. *iteArgs* → *expIte*; *expIte*; *expIte*
14. *expIte* → *attribuition* | ε
15. *oneLineStmt* → *varDecl*; | *attribuition*; | *io*; | *ret*;
16. *io* → **READ**(**ID**) | **WRITE**(*attribuition*) | **WRITE**(**STRING**)
17. *varDecl* → **TIPO ID** | **TIPO LIST ID**
18. *attribuition* → **ID** = *expLogic* | *expLogic*
19. *expLogic* → *expLogic* || *andLogic* | *andLogic*
20. *andLogic* → *andLogic* && *expComp* | *expComp*
21. *expComp* → *expComp* == *expRel* | *expComp* != *expRel* | *expRel*
22. *expRel* → *expRel* < *expArit* | *expRel* > *expArit* | *expRel* <= *expArit* | *expRel* >= *expArit* | *expArit*
23. *expArit* → *expArit* + *expMul* | *expArit* - *expMul* | *expMul*
24. *expMul* → *expMul* * *negElement* | *expMul* / *negElement* | *negElement*
25. *negElement* → !*expList* | -*expList* | *expList*
26. *expList* → !*element* | ?*element* | %*element* | *expList* << *element* | *expList* >> *element* | *expList* : *element*
27. *element* → **ID** | (*attribuition*) | **ID**(*arguments*) | **ID**() | **CONST_INT** | **CONST_FLOAT** | **NIL**
28. *arguments* → *arguments*, *attribuition* | *attribuition*
29. *return* → **return** *attribuition*