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

Nesta seção serão descritas as etapas que foram realizadas para se criar o tradutor da linguagem C-IPL.

2.1 Analisador Léxico

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), que é, então, enviado para o analisador semântico.

Por meio do programa *FLEX* [EP01], são criados autômatos a partir das expressões regulares. Por meio desses autômatos, é possível reconhecer os lexemas e, então, classificar e construir os *tokens* por meio do analisador léxico. Na primeira fase do projeto, ao ocorrer uma correspondência entre lexema e expressão regular, era impresso o *token* correspondente, assim como a linha e a 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 a 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.

2.2 Analisador Sintático

A segunda etapa do projeto da disciplina consiste em implementar a análise sintática. Essa fase do tradutor recebe os *tokens* construídos 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 geração da 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 a coluna onde ele foi encontrado, respectivamente. A partir dos tokens retornados e o autômato de pilha gerado pelo *Bison*, são feitas reduções para identificar se a entrada é aceita ou não pela linguagem C-IPL. Caso seja encontrada alguma construção que não está inclusa na gramática, é destacada a linha e a 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 encontrar o fim de uma produção, é criado um nó da árvore. O nó consiste em uma *struct* composta por: um nome, que o identifica; Um inteiro que representa o escopo ao qual pertence, que é utilizado para demarcar o início de um escopo para a impressão da árvore; Um inteiro que representa o tipo do nó, utilizado durante a análise semântica para checagem de tipos; Três ponteiros para nós filhos; Um ponteiro para uma lista de nós, que é utilizado para simplificar a árvore criada; E um ponteiro para a tabela de símbolos, para caso o símbolo do nó esteja presente nela. A fim de simplificar a árvore sintática gerada, foram adotadas duas técnicas. A primeira consiste em passar adiante o ponteiro do nó caso seja uma regra composta por apenas um não-terminal ou similar, como em: $expLog \rightarrow expArit$. Para a segunda técnica utiliza-se o ponteiro que aponta para uma lista de nós. A lista de nós evita que hajam nós redundantes na árvore abstrata, como em $moreStmt \rightarrow moreStmt\ block$, regra essa que poderia gerar vários nós desnecessários $moreStmt$. No entanto, por meio da lista de nós, é possível que todos os *stmt* sejam encadeados em uma mesma estrutura, cada qual podendo gerar sua própria árvore.

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; função ou constante; 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 a coluna, que identificam o local onde foi declarado; e ponteiros para a lista dos tipos dos argumentos, caso seja uma função, e o próximo elemento

da lista de símbolos. Quanto a lista de tipos de argumentos, à medida que são identificados os argumentos de uma função, seus tipos são armazenados nesta lista.

Para auxiliar no processo de definir os escopos presentes nos programas, foi necessário criar uma lista. Nela os escopos são representados por inteiros, de forma que a lista seja iniciada com o escopo 0 como primeiro elemento. No momento em que o analisador léxico identifica o caractere "{", a variável *escopo_max*, que armazena o escopo de maior valor, é acrescido em 1 e é armazenado no fim desta lista. Ao encontrar o caractere "}", o último elemento da lista é retirado e o novo último escopo é copiado para a variável *escopo_atual*, que armazena o escopo deste momento da análise. Este mesmo processo de utilização da lista se repete durante a declaração de parâmetros de funções.

2.3 Analisador Semântico

A partir da árvore síntatica abstrata e a tabela de símbolos geradas pelo analisador sintático, o analisador semântico tem como objetivo verificar a consistência semântica do programa fonte em relação a linguagem C-IPL. Para esta fase, foi adotada a tradução em um passo, ou seja, foi necessário percorrer as estruturas intermediárias apenas uma vez durante a análise. Assim, a análise semântica foi feita a partir das ações presentes no fim das regras da gramática.

A conversão de tipos de variáveis e constantes foi feita da seguinte forma:

- **Inteiras e de ponto flutuante**: em operações unárias, o tipo é mantido e não é adicionado nada na árvore. No caso de ocorrer uma operação binária entre um *int* e um *float*, haverá uma coerção no inteiro, assim, adicionando um nó (*float*) que indica esta coerção na árvore e o tipo do nó da operação se torna *float*. Caso seja feita uma operação binária entre tipos iguais, não há coerção e o tipo do nó da operação é o mesmo dos operadores. Além disso, operações lógicas e relacionais sempre resultam em um tipo inteiro. Por fim, a utilização de inteiros e números de ponto flutuante em operações de listas resulta em erro léxico, exceto como valor à esquerda do construtor (:);
- **Listas**: em qualquer operação, binária ou unária, não há coerção com variáveis de lista. Variáveis do tipo lista são aceitas nas seguintes situações: em algumas operações relacionais (== e !=), quando comparadas com a constante NIL; Em operações de lista, ao depender de como a operação é formulada; E operações de atribuição.

Quanto a regra de escopo da declaração de variáveis e funções, foram utilizadas a lista de escopo e a tabela de símbolos introduzidas na análise sintática. Ao identificar uma variável, o analisador léxico busca por uma entrada na tabela de símbolos que tenha o mesmo identificador da variável e que esteja em um escopo contido na lista, priorizando o escopo mais próximo do atual. Assim, caso não encontre esta variável na tabela seguindo as condições, é anunciado o erro. De forma semelhante, ao declarar uma variável, é realizada uma busca na tabela para checar se esta variável já foi declarada no escopo atual. De forma

que, caso já tenha sido declarada, é anunciado um erro semântico. Caso não, a nova variável é adicionada na tabela. Todo este processo é feito de forma similar quanto a declaração de funções.

Para resolver erros relacionados a passagem de parâmetros, foi utilizada a tabela de símbolos. Durante a declaração de uma função, o número de argumentos e seus tipos são armazenados na entrada correspondente à função na tabela. Então, quando ocorre a chamada de uma função, é procurada a entrada dela na tabela. Caso não encontre, é anunciado um erro. Porém, caso encontre, é checado se o número de parâmetros e seus tipos correspondem aos presentes nesta entrada. De forma que, se houver alguma discrepância, é anunciado um erro. No caso específico das operações de lista *map* e *filter*, é checado se a função à esquerda do operador possui apenas um argumento.

Quanto ao retorno das funções, é checado se o tipo do valor retornado é igual ao da definição da função. Caso não, é informado um erro. Além disso, para checar se a função *main* foi definida, no fim do programa, antes de imprimir a árvore abstrata e a tabela de simbolos, é buscada na tabela uma entrada para a função *main*. De forma que, caso não exista esta entrada, é anunciado um erro semântico.

Durante a criação do analisador semântico, foi necessário alterar alguns detalhes das análises léxicas e sintáticas. A única alteração no analisador léxico foi a divisão as operações binárias de listas em dois grupos, representados pelos *tokens* LIST_OP_BIN e LIST_OP_CONSTRUTOR, visto que os tipos dos operandos de *map* e *filter* são bastante diferentes dos operandos do construtor (:). Quanto ao analisador sintático, foi necessário alterar a gramática para diminuir a precedência de operações binárias de lista, porém, nada foi modificado no tratamento de erros sintáticos. Como foi escolhida a análise em um passo, a adição de nós na árvore sintática abstrata foi modificada para permitir a adição de nós de coerção, de acordo com o tipo dos operadores.

Em resumo, foram criadas funções e seções de código capazes de detectarem os seguintes erros: (1) tipos errados, (2) número de parâmetros e tipos errados em uma chamada de função, (3) variáveis e funções redeclaradas, (4) variáveis e funções não declaradas e (5) falta da função *main*.

A Figura 1 apresenta um trecho de código, na linguagem C-IPL. Enquanto que nas Figuras 2 e 3 são exemplificadas a árvore sintática e a tabela de símbolos geradas a partir do trecho de código.

```
1  int sei_la;
2
3  int fibonacci(float num) {
4      sei_la = 5;
5      if(num == 0 || num == 1) {
6          return 1;
7      } else {
8          return fibonacci(num-1) + fibonacci(num-2);
9      }
10 }
```

Figura 1. Trecho de código C-IPL.

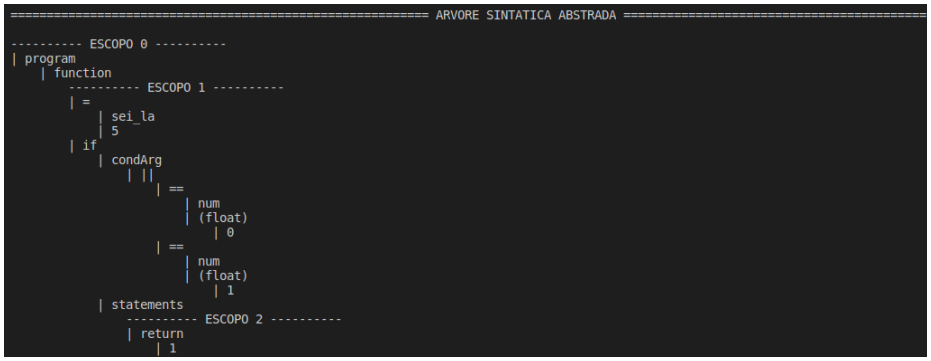

Figura 2. Árvore sintática abstrata gerada do exemplo.

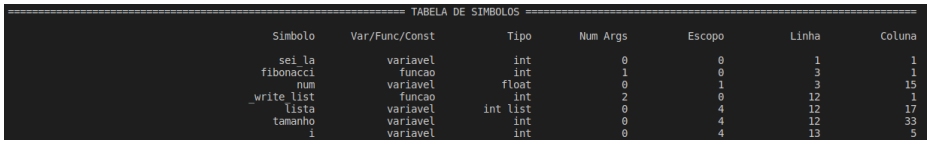

Figura 3. Tabela de símbolos gerada do exemplo.

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 e semânticos que calcula e apresenta a classificação do IMC de acordo com os dados inseridos;
2. *teste2_correto.ci*: programa sem erros sintáticos e semânticos que monta em uma lista a sequência de *Fibonacci* até uma posição n;
3. *teste1_errado.ci*: programa com erros sintáticos e semânticos que recebe uma lista e retorna se a soma dos elementos contidos nesta lista é positiva ou negativa. Os erros sintáticos presentes são a falta de um operador na linha 21 e coluna 16, a falta de ponto e vírgula na linha 58 e coluna 9, e o *writeln* sem argumentos na linha 80 e coluna 17. Os erros semânticos são a declaração implícita da função soma na linha 14 e coluna 15, redeclaração da função soma na linha 24 e coluna 1, variável não declarada na linha 47 e coluna 21, muitos argumentos para a chamada da função soma na linha 47 e coluna 22, redeclaração da variável lista na linha 56 e coluna 9, e declaração implícita da função soma2 na linha 88 e coluna 13;

4. *teste2_errado.ci*: programa com erros sintáticos e semânticos que encontra o menor número de ponto flutuante de uma lista. Os erros sintáticos presentes são o símbolo > na linha 18 e coluna 14, o *write* sem argumentos na linha 20 e coluna 15, e o *for* com apenas uma expressão na linha 26 e coluna 18. Os erros semânticos são a variável lista1 não declarada na linha 8 e coluna 24, a variável num não declarada na linha 15 e coluna 12, e o argumento de tipo errado na chamada de função na linha 32 e coluna 43.

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 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***	Constante
Constante Float	***CONST_FLOAT***	Constante
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_NEG***	!
<=	***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_HEADER***	?
!	***LIST_OP_UN***	!
%	***LIST_OP_UN***	%
:	***LIST_OP_CONSTRUTOR***	:
>>	***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* | *varDecl*;
4. *function* → *funcDecl* (*parameters*) {*moreStmt*} | *funcDecl* () {*moreStmt*}
5. *funcDecl* → **TIPO ID** | **TIPO LIST ID**
6. *parameters* → *parameters*, *varDecl* | *varDecl*
7. *moreStmt* → *moreStmt block* | *block*
8. *block* → *stmt* | *varDecl*; | {*moreStmt*}
9. *stmt* → *conditional* | *iteration* | *attribuition*; | *io*; | *ret*;
10. *conditional* → **IF** (*attribuition*) *bracesOrNot* | **IF** (*attribuition*) *bracesOrNot* **ELSE** *bracesOrNot*
11. *bracesOrNot* → {*moreStmt*} | *stmt*
12. *iteration* → **FOR** (*iteArgs*) *bracesOrNot*

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