Tradutores - Analisador Semântico

Pedro Lucas Pinto Andrade - 160038316

Ciência da Computação, Universidade de Brasília, Brasília DF, Brasil
pedrolpandrade@gmail.com

1 Motivação

O funcionamento de um tradutor é dividido em diversas etapas, sendo uma delas a etapa de análise. A análise também é dividida em quatro tipos principais: análise léxica, análise sintática, análise semântica e geração de código intermediário, cada uma possuindo um papel específico na execução do tradutor [ALSU07]. As fases têm como objetivo: ler um arquivo de entrada e separar o código em *tokens* seguindo as regras especificadas pelas expressões regulares (léxico); analisar se os *tokens* estão organizados conforme o conjunto de regras da gramática da linguagem (sintático); verificar se a semântica do código está condizente com o definido para a linguagem (semântico); e gerar o código intermediário que é utilizado para, posteriormente, gerar o código de máquina (geração de código intermediário).

Nesse trabalho, a linguagem escolhida é uma versão simplificada de C chamada C-IPL [Nal21], com a adição de um tipo novo de dados para listas (*list*), além de primitivas para manipular os dados nessas listas.

Com essas operações para manipulação dos elementos, uma lista se torna uma estrutura de dados versátil, que permite implementar diversas outras estruturas de dados por meio de abstrações, como pilhas ou filas. A adição da lista ao subconjunto da linguagem que foi selecionado para o trabalho permite armazenar e manipular dados de formas variadas.

2 Descrição

Para realizar a análise léxica, foram criadas definições (Tabela 1) com o uso de expressões regulares para separar o código de entrada em tokens, com o auxílio do gerador Flex. As expressões regulares determinam padrões para: identificadores; tipos de dados; operadores, como aritméticos, binários, lógicos e os operadores utilizados para manipular as listas; delimitadores, como parênteses, chaves, aspas e delimitadores de comentários; e marcadores utilizados para a formatação, como espaços e quebras de linha. Sempre que um lexema é lido e reconhecido por uma das expressões regulares, uma função correspondente dessa expressão é executada, informando qual o tipo do *token* e utilizando seu tamanho para calcular a sua posição no código de entrada.

Para esse posicionamento em relação à linha e coluna, são utilizadas variáveis globais para armazenar em qual posição está o *token* que foi lido por último,

calculadas com o auxílio da variável *yylen*, incluída pelo Flex, que guarda o tamanho do lexema lido. O posicionamento também é utilizado para a impressão de erros léxicos quando um lexema não segue as regras do conjunto de definições regulares.

Após ler e reconhecer cada lexema, o analisador léxico monta e envia os *tokens* para o analisador sintático, que cuida da próxima etapa da tradução.

O analisador sintático utiliza os *tokens* recebidos do léxico para verificar se o código está seguindo a sintaxe definida pelo conjunto de regras da gramática da linguagem C-IPL. Para isso, é construído um analisador, com o auxílio do Bison [DS21], que utiliza tabelas LR(1) canônicas para interpretar o código conforme as regras da gramática.

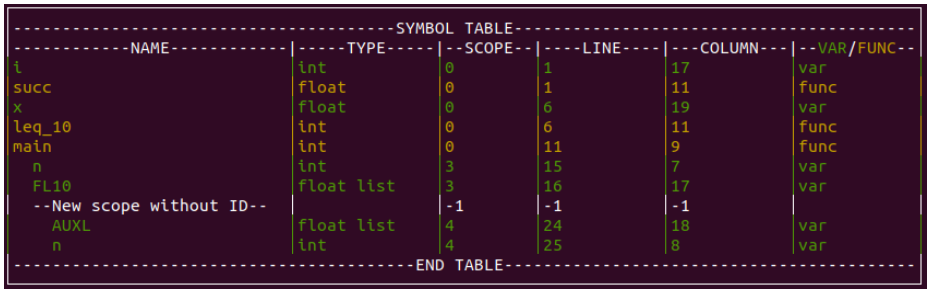
O analisador sintático é responsável por armazenar todas as unidades léxicas relevantes na tabela de símbolos, como declarações de variáveis e de funções, assim como sua posição no código e seu escopo. O analisador léxico também pode ser responsável por armazenar alguns dos *tokens* na tabela, mas, para simplificar o funcionamento do tradutor nesse trabalho, todas as inserções na tabela foram mantidas na fase sintática.

Os registros na tabela de símbolos serão úteis durante a fase semântica da análise, onde o programa poderá utilizar as informações armazenadas para determinar se as operações do código de entrada estão seguindo as regras semânticas definidas para a linguagem C-IPL, como verificar se uma variável está sendo utilizada dentro de seu escopo, se uma função está recebendo todos os parâmetros necessários, entre outras.

A tabela de símbolos foi implementada utilizando uma lista de símbolos [Aab04]. Um símbolo é representado por uma estrutura que contém os campos relevantes para a tabela (nome, posicionamento em linha e coluna, escopo, escopo pai e se é uma variável ou função) e campos relevantes para o funcionamento da lista (um ponteiro para o próximo elemento). Dois ponteiros globais são utilizados para auxiliar na implementação da lista, um para o começo e outro para o último elemento inserido.

Uma pilha é utilizada para determinar o escopo das declarações que são armazenadas na tabela de símbolos. Ao encontrar uma abertura de escopo, um novo valor é empilhado, para que todas as declarações dentro desse bloco de código recebam esse valor na tabela. Ao encontrar o fechamento do escopo, o valor é desempilhado. Um contador global é utilizado para garantir que o valor de cada escopo é único. O valor de escopo pai é uma forma de determinar qual símbolo, se existir, foi responsável por abrir um escopo onde uma determinada variável foi declarada. Isso auxilia na impressão da tabela de símbolos, deixando mais claro onde cada declaração ocorreu. Isso pode ser visualizado no exemplo da Figura 1.

A árvore sintática é construída pelo analisador sintático para exibir quais foram as derivações da gramática realizadas para obter o código C-IPL de entrada, mostrando assim como o código está dentro da sintaxe definida para a linguagem. Entretanto, a árvore construída não é a de derivação completa, mas sim a



SYMBOL TABLE

NAME	TYPE	SCOPE	LINE	COLUMN	VAR/FUNC
i	int	0	1	17	var
succ	float	0	1	11	func
x	float	0	6	19	var
leq_10	int	0	6	11	func
main	int	0	11	9	func
n	int	3	15	7	var
FL10	float list	3	16	17	var
--New scope without ID--		-1	-1	-1	
AUXL	float list	4	24	18	var
n	int	4	25	8	var

END TABLE

Figura 1. Exemplo de tabela de símbolos criada pelo tradutor

abstrata. Portanto, algumas regras são omitidas de forma a facilitar a leitura da sintaxe.

Cada nó da árvore é uma estrutura (Lista 1.1) com seu nome e ponteiros para seus nós filhos, para que seja possível encadear e exibir todos em formato de árvore ao final da execução. Além disso, cada um também possui uma flag para determinar se o mesmo é um nó vazio. Os nós vazios são criados nas derivações vazias e não são necessários para a impressão, mas sim para o funcionamento correto da árvore. Algumas regras da gramática possuem o intuito de gerar uma lista de outras regras com um número indeterminado de elementos, como a regra de declarações locais, por exemplo. Com a montagem da árvore, isso acaba gerando uma sequência de nós repetidos que dificulta a leitura. Para resolver isso, os nós repetidos são tratados e todos os filhos são impressos como filhos de um único nó. [Zan11]

```
typedef struct Node{
    char* name;
    int empty;
    struct Node** child;
} t_node;
```

Lista 1.1. Estrutura do nó

O analisador semântico utiliza a árvore sintática e a tabela de símbolos para verificar se a semântica do código C-IPL está correta. Para isso, algumas regras semânticas são verificadas:

- Existência de uma definição correta da *main*;
- Parâmetros corretos (número e tipo) em chamadas de funções;
- Existência de um identificador utilizado em um escopo válido;
- Tipos válidos para operadores;

Para verificar as regras semânticas uma estratégia de 2 passagens foi adotada, pois algumas regras são mais facilmente verificadas durante a análise sintática, enquanto outras são mais facilmente verificadas utilizando a árvore sintática e a tabela de símbolos completas.

Durante a análise sintática são verificados os números e os tipos de parâmetros em uma chamada de função e se os identificadores utilizados foram declarados em um escopo válido.

Para verificar os parâmetros, ao encontrar uma declaração de função a análise sintática registra na tabela de símbolos quais são os parâmetros necessário para chamar a função. Ao encontrar uma chamada de função, a análise semântica armazena os tipos dos parâmetros passados, busca na tabela de símbolos onde está a declaração da função e compara os tipos do dos parâmetros passados com os parâmetros armazenados. Caso o número ou os tipos sejam discrepantes, um erro é exibido.

Para verificar se um identificador foi declarado em um escopo válido, ao encontrar um identificador sendo utilizado na análise sintática, a análise semântica busca na tabela de símbolos se existem declarações desse identificador e, caso sejam encontrados, percorre a pilha procurando quais escopo ainda estão válidos naquele ponto do código. Caso a declaração encontrada faça parte de um dos escopos válidos, o identificador pode ser usado. Caso nenhuma declaração satisfaça isso, um erro é exibido.

Após o término da análise sintática e da construção completa da árvore sintática e da tabela de símbolos, é analisado se existe uma definição correta da *main*. Para isso, a tabela de símbolos é percorrida em busca de declarações de uma função com o nome *main*. Caso não exista ou exista mais de uma, um erro é exibido.

A verificação de tipos válidos para operadores ainda não foi implementada, mas seguirá uma ideia similar à de verificação de tipos de parâmetros em uma chamada de função.

3 Arquivos de teste

Estão disponíveis quatro arquivos de teste do analisador dentro da pasta */tests/*. Os arquivos são divididos em arquivos sem erros sintáticos:

- *test_ correct1.c*;
- *test_ correct2.c*;

E arquivos com erros semânticos:

- *test_ wrong1.c* - Erros de tipo de parâmetro incorreto (linha 84 coluna 33) e redefinição da função main (linha 93 coluna 9);
- *test_ wrong2.c* - Erros de número de parâmetros incorreto (linha 28 coluna 9) e redefinição da função main (linha 35 coluna 9);

4 Instruções para compilação e execução

Um arquivo Make foi incluído para facilitar a compilação do tradutor. Basta utilizar o comando

```
$ make semantico
```

na pasta principal.

Caso ocorra algum problema com o arquivo Make, é possível compilar executando os comandos contidos no arquivo *README.txt* na pasta principal.

Após compilar, basta executar os seguintes comandos para rodar os arquivos de teste:

```
$ make run_correct1
$ make run_correct2
$ make run_wrong1
$ make run_wrong2
```

Para executar o analisador em outros arquivos, basta utilizar o seguinte comando:

```
$ ./tradutor teste.c
```

trocando a palavra *teste* pelo nome de um arquivo na linguagem C-IPL.

As versões dos sistemas utilizados para compilação foram: *Flex* (flex 2.6.4), *Bison* (GNU Bison 3.7.5 compilado do código fonte), *Make* (GNU Make 4.2.1), *GCC* (11.2.0 compilado do código fonte), *OS* (Linux 5.8.0-63-generic 20.04.1-Ubuntu).

Referências

[Aab04] A. A. Aaby. Compiler construction using flex and bison. https://www.admb-project.org/tools/flex/compiler.pdf. Acessado em 14 Set 2021, Fevereiro 2004.

[ALSU07] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Addison Wesley, 2nd edition, 2007.

[DS21] C. Donnelly and R. Stallman. Bison - The Yacc-compatible Parser Generator. https://www.gnu.org/software/bison/manual/bison.pdf. Acessado em 14 Set 2021, Setembro 2021.

[Hec21] R. Heckendorn. A grammar for the C- programming language (version s21). http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf. Acessado em 14 Set 2021, Março 2021.

[Nal21] C. Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Acessado em 14 Set 2021, Agosto 2021.

[Zan11] B. V. Zanden. Building Abstract Syntax Trees. http://web.eecs.utk.edu/ bvanderz/cs461/notes/parse_tree/. Acessado em 15 Set 2021, Fevereiro 2011.

A Tabelas léxicas

A Tabela 1 mostra os *tokens* criados para o analisador léxico, além das expressões regulares utilizadas para definir cada um. A Tabela 2 mostra os lexemas de cada *token* e seu valor de atributo.

Tabela 1. Definições dos tokens no código Flex.

Token	Expressão regular	Exemplo de lexema
DIGIT	[0-9]	0, 6
INT	{DIGIT}+	1955
FLOAT	{DIGIT}+[.]{DIGIT}+	11.05
ID	[a-zA-Z_][a-z0-9A-Z]*	main
TYPE	int|float	int
LIST_TYPE	list	list
STRING	\"(\\.|[^"\\])* \"	"texto"
NULL_CONST	nil	nil
PLUS_OP	+	+
MINUS_OP	-	-
DIV_OP	/	/
MUL_OP	*	*
LOGIC_OP	|||&&	||, &&
BINARY_OP	<|<=|>|>=|==|!=	==, >
ASSIGN_OP	[=]	=
EXCLA_OP	[!]	!
IF_KEY	if	if
ELSE_KEY	else	else
FOR_KEY	for	for
RETURN_KEY	return	return
INPUT_KEY	read	read
OUTPUT_KEY	write	write
OUTPUTLN_KEY	writeln	writeln
ASSIGN_LISTOP	:	:
HEADER_LISTOP	?	?
TAILDES_LISTOP	%	%
MAP_LISTOP	»	»
FILTER_LISTOP	«	«
DELIM_PARENT_L	(	(
DELIM_PARENT_R	)	)
DELIM_BRACKET_L	[	[
DELIM_BRACKET_R	]	]
DELIM_CUR_BRACKET_L	{	{
DELIM_CUR_BRACKET_R	}	}
DELIM_COMMA	[,]	,
DELIM_SEMICOLLON	[;]	;
DELIM_SQUOTE	[']	'
DELIM_DQUOTE	["]	"
SINGLE_COMMENT	//[^\n]*	// texto
MULTI_COMMENT	\/*[^(*\/)]**\/	/* texto */
FORMAT_BLANKSPACE	[ ]	*espaço em branco*
FORMAT_NEWLINE	\n	*quebra de linha*
FORMAT_TAB	\t	*tab*

Tabela 2. Tokens e seus lexemas e valores de atributo correspondentes

Lexemas	Token	Valor de atributo
Inteiro	**INT**	Ponteiro para a tabela de símbolos
Decimal	**FLOAT**	Ponteiro para a tabela de símbolos
Id	**ID**	Ponteiro para a tabela de símbolos
int	**TYPE**	INT
float	**TYPE**	FLO
list	**LIST_TYPE**	LIST
Cadeia de caracteres	**STRING**	Ponteiro para a tabela de símbolos
nil	**NULL_CONST**	-
+	**PLUS_OP**	PLUS
-	**MINUS_OP**	MINUS
/	**DIV_OP**	DIV
*	**MUL_OP**	MUL
&&	**LOGIC_OP**	AND
||	**LOGIC_OP**	OR
<	**BINARY_OP**	LT
<=	**BINARY_OP**	LE
>	**BINARY_OP**	GT
>=	**BINARY_OP**	GE
==	**BINARY_OP**	EQ
!=	**BINARY_OP**	DIF
=	**ASSIGN_OP**	-
!	**EXCLA_OP**	-
if	**IF_KEY**	IF
else	**ELSE_KEY**	ELS
for	**FOR_KEY**	FOR
return	**RETURN_KEY**	RET
read	**INPUT_KEY**	-
write	**OUTPUT_KEY**	BASE
writeln	**OUTPUTLN_KEY**	LN
«	**LIST_OP**	FILT
»	**LIST_OP**	MAP
?	**LIST_OP**	HEAD
%	**LIST_OP**	TAIL
:	**LIST_OP**	CONS
(	**DELIM_PARENT_L**	LEFT
)	**DELIM_PARENT_R**	RIGHT
[	**DELIM_BRACKET_L**	LEFT
]	**DELIM_BRACKET_R**	RIGHT
{	**DELIM_CUR_BRACKET_L**	LEFT
}	**DELIM_CUR_BRACKET_R**	RIGHT
,	**DELIM_COMMA**	-
;	**DELIM_SEMICOLLON**	-
'	**DELIM_SQUOTE**	-
"	**DELIM_DQUOTE**	-
//texto...	**SINGLE_COMMENT**	-
*/*texto...*/*	**DOUBLE_COMMENT**	-
Espaço em branco	**FORMAT_BLANKSPACE**	-
\n	**FORMAT_NEWLINE**	-
\t	**FORMAT_TAB**	-

B Gramática da linguagem

A gramática foi criada com base na gramática para a linguagem C- [Hec21] e nas regras padrões de C das operações que fazem parte da linguagem reduzida definida para o trabalho.

1. *program* → *declarationList*
2. *delarationList* → *declarationList declaration* | *declaration* | *declarationList statement* | *statement*
3. *declaration* → *varDeclaration* | *funcDeclaration*
4. *varDeclaration* → **TYPE ID DELIM_SEMICOLLON** | **TYPE LIST_TYPE ID DELIM_SEMICOLLON**
5. *funcDeclaration* → **TYPE ID DELIM_PARENT_L** *parameters* **DELIM_PARENT_R** *bodyStatement* | **TYPE LIST_TYPE ID DELIM_PARENT_L** *parameters* **DELIM_PARENT_R** *bodyStatement*
6. *parameters* → *parameterList* | ϵ
7. *parameterList* → *parameterList* **DELIM_COMMA** *parameterSimple* | *parameterSimple*
8. *parameterSimple* → **TYPE ID** | **TYPE LIST_TYPE ID**
9. *statement* → *bodyStatement* | *ifStatement* | *loopStatement* | *returnStatement* | *listStatement* **DELIM_SEMICOLLON** | *writeOp* **DELIM_SEMICOLLON** | *readOp* **DELIM_SEMICOLLON** | *expressionStatement*
10. *bodyStatement* → **DELIM_CUR_BRACKET_L** *statementList* **DELIM_CUR_BRACKET_R**
11. *localDeclaration* → *localDeclaration varDeclaration* | ϵ
12. *statementList* → *statementList localDeclaration statement* | ϵ
13. *ifStatement* → **IF_KEY DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R** *statement* | **IF_KEY DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R** *statement* **ELSE_KEY** *statement*
14. *loopStatement* → **FOR_KEY DELIM_PARENT_L** *expression* **DELIM_SEMICOLLON** *simpleExpression* **DELIM_SEMICOLLON** *expression* **DELIM_PARENT_R** *statement*
15. *returnStatement* → **RETURN_KEY** *expression* **DELIM_SEMICOLLON**
16. *expression* → **ID ASSIGN_OP** *expression* | *simpleExpression*
17. *simpleExpression* → *logicBinExpression*
18. *logicBinExpression* → *logicBinExpression* **LOGIC_OP** *logicUnExpression* | *logicUnExpression*
19. *logicUnExpression* → **EXCLA_OP** *logicUnExpression* | *binExpression*
20. *binExpression* → *binExpression* **BINARY_OP** *sumExpression* | *sumExpression*
21. *sumExpression* → *sumExpression sumOP mulExpression* | *mulExpression*
22. *mulExpression* → *mulExpression mulOP factor* | *factor*

23. *sumOP* → **PLUS_OP** | **MINUS_OP**
24. *mulOP* → **MUL_OP** | **DIV_OP**
25. *factor* → **ID** | *functionCall* | **DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R** | *listExpression* | *constant*
26. *constant* → **INT** | **MINUS_OP INT** | **FLOAT** | **MINUS_OP FLOAT** | **NULL_CONST**
27. *functionCall* → **ID DELIM_PARENT_L** *parametersPass* **DELIM_PARENT_R**
28. *parametersPass* → *parametersPass* **DELIM_COMMA** *simpleExpression* | *simpleExpression* | ϵ
29. *writeOp* → *write* | *writeln*
30. *write* → **OUTPUT_KEY DELIM_PARENT_L STRING DELIM_PARENT_R** | **OUTPUT_KEY DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R**
31. *writeln* → **OUTPUTLN_KEY DELIM_PARENT_L STRING DELIM_PARENT_R** | **OUTPUTLN_KEY DELIM_PARENT_L** *simpleExpression* **DELIM_PARENT_R**
32. *readOp* → **INPUT_KEY DELIM_PARENT_L ID DELIM_PARENT_R**
33. *expressionStatement* → *expression* **DELIM_SEMICOLLON**
34. *listStatement* → *listAssign* | *listMap* | *listFilter*
35. *listExpression* → *listHeader* | *listTailDestructor*
36. *listAssign* → $\mathbf{ID_1}$ **ASSIGN_OP** $\mathbf{ID_2}$ **ASSIGN_LISTOP** $\mathbf{ID_1}$
37. *listHeader* → **HEADER_LISTOP ID**
38. *listTailDestructor* → **TAILDES_LISTOP ID**
39. *listMap* → **ID ASSIGN_OP** $\mathbf{ID_1}$ **MAP_LISTOP ID**
40. *listFilter* → **ID ASSIGN_OP** $\mathbf{ID_1}$ **FILTER_LISTOP ID**