Tradutores - Analisador Léxico

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 divido em diversas etapas, sendo uma delas a etapa de análise. A análise também é dividia em três tipos principais: análise léxica, análise sintática e análise semântica, cada uma possuindo um papel específico na execução do tradutor [ALSU07]. O analisador léxico, o foco desse trabalho, tem como objetivo ler um arquivo de entrada em uma linguagem definida e identificar quais são os tokens existentes nesse arquivo, seguindo as regras especificadas pela gramática. Nesse trabalho, a linguagem escolhida é uma versão simplificada de C [Nal21], com a adição de um tipo novo de dados: list.

O tipo list é um tipo de dado utilizado para armazenar informações no formato de uma lista. Para que seja mais útil, também são incluídos alguns operadores para manipular os dados armazenados.

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 subset 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 identificam 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 tags utilizadas para a formatação, como espaços. Sempre que um token 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 token lido. O posicionamento também é utilizado para a impressão de erros léxicos quando um token não segue as regras da gramática.

Futuramente, o analisador léxico irá armazenar os tokens lidos para uso do analisador sintático. Cada token será uma struct contendo o nome do token e seu valor de atributo.

Uma tabela de símbolos será utilizada para armazenar alguns tokens. Para isso, cada símbolo será um struct contendo pelo menos um nome e um tipo, podendo possivelmente conter também seu escopo e um valor de atributo caso necessário. O tradutor irá construir ela durante a análise léxica e sintática e a usará nas etapas de análise sintática e semântica, assim como em fases mais avançadas.

3 Arquivos de teste

Estão disponíveis 4 arquivos de teste do analisador dentro da pasta /tests/. Os arquivos são divididos em arquivos sem erros léxicos: *test_ correct1.c* e *test_ correct2.c*;

E arquivos com erros léxicos, onde os erros são de tokens não identificados pela gramática da linguagem: *test_ wrong1.c* (Erros de token indefinido na linha 2 coluna 9 e linha 12 coluna 17) e *test_ wrong2.c* (Erros de token indefinido na linha 4 coluna 5 e linha 7 coluna 9).

4 Instruções para compilação e execução

Um arquivo Make foi incluído para facilitar a compilação do tradutor. Basta utilizar o comando *make* na pasta principal. Caso ocorra algum problema com o arquivo Make, os comandos podem ser executados manualmente em um terminal:

```
flex -o ./src/lex.yy.c ./src/analyser.l
gcc ./src/lex.yy.c -o tradutor -Wall
```

Após compilar, basta executar o analisador com o seguinte comando:

```
./tradutor ./tests/teste.c
```

trocando a palavra *teste* pelo nome de um dos arquivos de teste citados anteriormente.

As versões dos sistemas utilizados para compilação foram: *Flex* (flex 2.6.4), *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

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

[Hec21] R. Heckendorn. A grammar for the c- programming language (version s21). http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf. Acessado em 08 Ago 2021, 2021.

[Nal21] C. Nalon. Trabalho prático - descrição da linguagem. https://aprender3.unb.br/mod/page/view.php?id=464034. Acessado em 09 Ago 2021, 2021.

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|list	int
STRING	{"}(\\.|[^{"}\\])*{"}	"texto"
NULL	nil	nil
ARITHMETIC_OP	+|-|/|*	+, -
LOGIC_OP	|||&&	||, &&
BINARY_OP	<|<=|>|>=|==|!=	==, >
ASSIGN_OP	[=]	=
EXCLA_OP	[!]	!
FLOW_KEY	if|else|for|return	return, if
INPUT_KEY	read	read
OUTPUT_KEY	write|writeln	write, writeln
LIST_OP	«|»|?|%|:	?, «
DELIM_PARENT	[()]	), (
DELIM_BRACKET	[[]]	], [
DELIM_CUR_BRACKET	[{}]	}, {
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*

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*

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	**TYPE**	LIST
Cadeia de caracteres	**STRING**	Ponteiro para a tabela de símbolos
nil	**NULL**	-
+	**ARITHMETIC_OP**	PLUS
-	**ARITHMETIC_OP**	MINUS
/	**ARITHMETIC_OP**	DIV
*	**ARITHMETIC_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	**FLOW_KEY**	IF
else	**FLOW_KEY**	ELS
for	**FLOW_KEY**	FOR
return	**FLOW_KEY**	RET
read	**INPUT_KEY**	-
write	**OUTPUT_KEY**	BASE
writeln	**OUTPUT_KEY**	LN
«	**LIST_OP**	FILT
»	**LIST_OP**	MAP
?	**LIST_OP**	HEAD
%	**LIST_OP**	TAIL
:	**LIST_OP**	CONS
(	**DELIM_PARENT**	LEFT
)	**DELIM_PARENT**	RIGHT
[	**DELIM_BRACKET**	LEFT
]	**DELIM_BRACKET**	RIGHT
{	**DELIM_CUR_BRACKET**	LEFT
}	**DELIM_CUR_BRACKET**	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**	-

3. *declaration* → *varDeclaration* | *funcDeclaration*
4. *varDeclaration* → **TYPE** *varDeclList*;
5. *varDeclList* → *varDeclList varDeclId* | *varDeclId*
6. *varDeclId* → **ID**
7. *funcDeclaration* → **TYPE ID** (*parameters*) *statement*
8. *parameters* → *parameterList* | ϵ
9. *parameterList* → *parameterList*, **TYPE ID** | **TYPE ID**
10. *statement* → *bodyStatement* | *ifStatement* | *loopStatement* | *listStatement*
11. *bodyStatement* → { *localDeclaration statementList* }
12. *localDeclaration* → *localDeclaration varDeclaration* | ϵ
13. *statementList* → *statementList statementList* | ϵ
14. *ifStatement* → **if** (*simpleExpression*) *bodyStatement* | **if** (*simpleExpression*) *bodyStatement* **else** *bodyStatement*
15. *loopStatement* → **for** (*expression* ; *simpleExpression* ; *expression*) *bodyStatement*
16. *returnStatement* → **return** *expression*;
17. *expression* → **ID ASSIGN_OP** *expression* | *simpleExpression*
18. *simpleExpression* → *binExpresion* | *logicExpression*
19. *logicExpression* → *simpleExpression* **LOGIC_OP** *simpleExpression* | **EXCLA_OP** *simpleExpression*
20. *binExpression* → *sumExpression* **BINARY_OP** *sumExpression* | *sumExpression*
21. *sumExpression* → *sumExpression sumOP mulExpression* | *mulExpression*
22. *mulExpression* → *mulExpression mulOP factor* | *factor*
23. *sumOP* → + | -
24. *mulOP* → * | /
25. *factor* → **ID** | *functionCall* | (*simpleExpression*)
26. *functionCall* → **ID** (*parameters*)
27. *writeOp* → *write* | *writeln*
28. *write* → **write**(**STRING**) | **write**(*simpleExpression*)
29. *writeln* → **writeln**(**STRING**) | **writeln**(*simpleExpression*)
30. *read* → **read**(**ID**)
31. *listStatement* → *listAssign* | *listHeader* | *listTail* | *listTailDestructor* | *listMap* | *listFilter*
32. *listAssign* → $\mathbf{ID_1}$ **ASSIGN_OP** $\mathbf{ID_2}$: $\mathbf{ID_1}$
33. *listHeader* → ?**ID**
34. *listTail* → !**ID**
35. *listTailDestructor* → %**ID**
36. *listMap* → **ID ASSIGN_OP** $\mathbf{ID_1}$ » **ID**
37. *listFilter* → **ID ASSIGN_OP** $\mathbf{ID_1}$ « **ID**