

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 dividido em diversas etapas, sendo uma delas a etapa de análise. A análise também é dividida em três tipos principais: análise léxica, análise sintática e análise semântica, cada uma possuindo um papel específico na execução do tradutor. 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, 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, com linha e coluna, 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.

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`;
- `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;
- `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 as seguintes:

- 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

1. Heckendorn, R.: A Grammar for the C- Programming Language (Version S21). <http://marvin.cs.uidaho.edu/Teaching/CS445/c-Grammar.pdf> (2021). Acessado em 08 Ago 2021
2. Aho, A., Lam, M., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. 2nd edn. Pearson Addison Wesley, (2007)
3. Nalon, C., Trabalho Prático - Descrição da Linguagem <https://aprender3.unb.br/mod/page/view.php?id=464034>. Acessado em 09 Ago 2021

Tabela 1. Definições dos tokens no código Flex.

Lexema	Expressão regular
DIGIT	[0-9]
INT	{DIGIT}+
FLOAT	{DIGIT}+[.]{DIGIT}+
ID	[a-zA-Z_][a-zA-Z0-9_]*
TYPE	int float list
STRING	{DELIM_DQUOTE}(\. \^[{DELIM_DQUOTE}\\\\])*{DELIM_DQUOTE}
NULL	nil
ARITHMETIC_OP	+ - / *
LOGIC_OP	&&
BINARY_OP	< <= > > = = !=
ASSIGN_OP	[=]
EXCLA_OP	[!]
FLOW_KEY	if else for return
INPUT_KEY	read
OUTPUT_KEY	write writeln
LIST_OP	« » ? % :
DELIM_PARENT	[()]
DELIM_BRACKET	[[]]
DELIM_CUR_BRACKET	[{}]
DELIM_COMMA	[,]
DELI_SEMICOLLON	[;]
DELIM_SQUOTE	[']
DELIM_DQUOTE	["]
SINGLE_COMMENT	//[^\\n]*
MULTI_COMMENT	\\\[^(*\\)]**\\
FORMAT_BLANKSPACE	[]
FORMAT_NEWLINE	\\n
FORMAT_TAB	\\t

A Tabela léxica

A tabela 1 mostra os tokens criados para o analisador léxico, além das expressões regulares utilizadas para definir cada um.

B Gramática da linguagem

A gramática foi criada com base na gramática para a linguagem C- [1] e nas regras padrões de C das operações que fazem parte da linguagem reduzida definida para o trabalho.

1. $program \rightarrow declarationList$
2. $declarationList \rightarrow declarationList\ declaration \mid declaration$
3. $declaration \rightarrow varDeclaration \mid funcDeclaration$
4. $varDeclaration \rightarrow \mathbf{TYPE}\ varDeclList;$
5. $varDeclList \rightarrow varDeclList\ varDeclId \mid varDeclId$
6. $varDeclId \rightarrow \mathbf{ID}$
7. $funcDeclaration \rightarrow \mathbf{TYPE}\ \mathbf{ID}\ (parameters)\ statement$
8. $parameters \rightarrow parameterList \mid \epsilon$
9. $parameterList \rightarrow parameterList, \mathbf{TYPE}\ \mathbf{ID} \mid \mathbf{TYPE}\ \mathbf{ID}$
10. $statement \rightarrow bodyStatement \mid ifStatement \mid loopStatement \mid listStatement$
11. $bodyStatement \rightarrow \{ localDeclaration\ statementList \}$
12. $localDeclaration \rightarrow localDeclaration\ varDeclaration \mid \epsilon$
13. $statementList \rightarrow statementList\ statementList \mid \epsilon$
14. $ifStatement \rightarrow \mathbf{if}\ (simpleExpression)\ bodyStatement \mid \mathbf{if}\ (simpleExpression)\ bodyStatement\ \mathbf{else}\ bodyStatement$
15. $loopStatement \rightarrow \mathbf{for}\ (expression\ ;\ simpleExpression\ ;\ expression)\ bodyStatement$
16. $returnStatement \rightarrow \mathbf{return}\ expression;$
17. $expression \rightarrow \mathbf{ID}\ \mathbf{ASSIGN_OP}\ expression \mid simpleExpression$
18. $simpleExpression \rightarrow binExpression \mid logicExpression$
19. $logicExpression \rightarrow simpleExpression\ \mathbf{LOGIC_OP}\ simpleExpression \mid \mathbf{EX-CLA_OP}\ simpleExpression$
20. $binExpression \rightarrow sumExpression\ \mathbf{BINARY_OP}\ sumExpression \mid sumExpression$
21. $sumExpression \rightarrow sumExpression\ sumOP\ mulExpression \mid mulExpression$
22. $mulExpression \rightarrow mulExpression\ mulOP\ factor \mid factor$
23. $sumOP \rightarrow + \mid -$
24. $mulOP \rightarrow * \mid /$
25. $factor \rightarrow \mathbf{ID} \mid functionCall \mid (simpleExpression)$
26. $functionCall \rightarrow \mathbf{ID}\ (parameters)$
27. $writeOp \rightarrow \mathbf{write} \mid \mathbf{writeln}$
28. $\mathbf{write} \rightarrow \mathbf{write}(\mathbf{STRING}) \mid \mathbf{write}(simpleExpression)$
29. $\mathbf{writeln} \rightarrow \mathbf{writeln}(\mathbf{STRING}) \mid \mathbf{writeln}(simpleExpression)$
30. $\mathbf{read} \rightarrow \mathbf{read}(\mathbf{ID})$

- 31. $listStatement \rightarrow listAssign \mid listHeader \mid listTail \mid listTailDestructor \mid listMap$
 $\mid listFilter$
- 32. $listAssign \rightarrow ID_1 \text{ ASSIGN_OP } ID_2 : ID_1$
- 33. $listHeader \rightarrow ?ID$
- 34. $listTail \rightarrow !ID$
- 35. $listTailDestructor \rightarrow \%ID$
- 36. $listMap \rightarrow ID \text{ ASSIGN_OP } ID_1 \gg ID$
- 37. $listFilter \rightarrow ID \text{ ASSIGN_OP } ID_1 \ll ID$