Tradutores: Analisador Léxico

Guilherme Andreúce Sobreira Monteiro - 14/0141961

Universidade de Brasília - Darcy Ribeiro - CiC/Est, DF/Brasíl
140141961@aluno.unb.br

Abstract. Este trabalho consiste na utilização do programa Flex (gerador léxico) para gerar tokens que serão utilizados como referência para implementar futuramente um tradutor. A linguagem proposta será uma sublinguagem C para lidar com listas. Neste primeiro estágio será apresentado somente como foi feita a geração desses tokens e uma possível gramática que será utilizada para a implementação do tradutor que será construído nos próximos estágios.

Keywords: Tradutor · Flex · Analisador léxico · Tokens.

1 Motivação e Proposta

A linguagem C é uma linguagem muito versátil para construção de estruturas e manipulação de dados, no entanto, para que essa versatilidade ocorra, o programador precisa entender profundamente o que ele está fazendo. [1] Neste contexto, podemos observar que em C, diferentemente de Python, a construção das estruturas parte toda do programador; para se construir uma lista em python, basta declarar o tipo da variável, enquanto em C você tem que construir utilizando estruturas e ponteiros. [2] [3] Para facilitar o uso da linguagem C e, particularmente, suas estruturas, essa sublinguagem surge com essa intenção. Assim será possível utilizar nativamente as operações e funcionalidades necessárias para realizar certas operações com listas simplificadamente.

2 Analisador léxico

Análise léxica é a primeira fase de um compilador onde este recebe um fluxo de caracteres de um código e os agrupa em tokens. Esses Tokens, também conhecidos como lexemas, são unidades básicas de significado para uma linguagem. Com uma gramática, o analisador léxico consegue identificar se esses tokens fazem parte ou não da linguagem proposta, e se não fazem, onde o erro está localizado. Nesta primeira etapa, o analisador léxico, com o auxílio do programa flex, analisa um trecho de código e separa seus elementos.

2.1 Funções adicionadas

Para poder realizar a análise de onde existe algum erro léxico em cada leitura de lexema analisada uma variável chamada *word_position* é incrementada em 1.

2.2 Tratamento de Erros Léxicos

Ao identificar um possível erro léxico, é impresso no terminal o local exato, tanto em posição de caractere quanto a linha em que o programa parou, além de também escrever qual foi o caractere ou token que não pertence à gramática. Facilitando a correção caso necessária.

3 Analisador Sintático

Utilizando o Bison e apartir da gramática deste relatório, foi construído no arquivo guillex.y a gramática que será utilizada para construir o analisador. Ainda não há a implementação de uma árvore ou tabela de símbolos nesta versão. Existe um problema na hora de ler os arquivos.

4 Arquivos de teste

O analisador léxico possui 4 testes. Os dois primeiros são testes que o analisador lê corretamente os lexemas. Os dois últimos são testes que apresentam erros. Os arquivos são: test_1.c, test_2.c test_3.c test_4.c. No caso dos arquivos que apresentam erro, temos o primeiro erro em: Linha 2, Posição 9: Símbolo não permitido. Palavra @; No segundo erro temos na linha 3, posição 16: Símbolo não permitido. Palavra @.

5 Instruções para compilação

Certifique-se de estar utilizando o sistema Ubuntu 20.04.2 LTS com o comando *lsb_release -a*. Para os próximos passos certifiquem-se de que o gerenciador de pacotes (neste exemplo é utilizado o apt) esteja atualizado com as informações mais recentes dos pacotes utilizando *apt update* Para compilar, tenha instalado o flex 2.6.4(apt get install flex), o gcc 9.3.0(apt get install gcc) e o make 4.2.1 (apt get make) vá até à pasta *guillex_2021-1/src* e execute o comando *make* no terminal. Os testes rodados encontram-se na pasta *guillex_2021-1/tests*. Os resultados obtidos encontram-se na pasta *guillex_2021-1/results*

References

1. A importância e as vantagens de saber programar em linguagem C https://computerworld.com.br/plataformas/importancia-e-vantagens-de-saber-programar-em-linguagem-c/. Acessado em 05 de agosto 2021
2. Linked List Program in C https://www.geeksforgeeks.org/linked-list-set-1-introduction/. Acessado em 05 de agosto 20201
3. Python lists https://www.w3schools.com/python/python_lists.asp. Acessado em 05 de agosto 2021
4. Manual Flex https://westes.github.io/flex/manual/. Acessado em 05 de agosto 2021

A Linguagem da gramática

1. *program* → *declarationList*
2. *declarationList* → *declarationList declaration* | *declaration*
3. *declaration* → *varDeclaration* | *funcDeclaration*
4. *varDeclaration* → *simpleVDeclaration*
5. *funcDeclaration* → *simpleFDeclaration* **(** *params* **)** *compoundStmt* | *simpleFDeclaration* **( )** *compoundStmt*
6. *simpleVDeclaration* → **TYPE ID**
7. *simpleFDeclaration* → **TYPE ID**
8. *params* → *params, param* | *param*
9. *param* → *simpleVDeclaration*
10. *compoundStmt* → **{** *stmtList* **}**
11. *stmtList* → *stmtList primitiveStmt* | primitiveStmt
12. *primitiveStmt* → *exprStmt* | *compoundStmt* | *condStmt* | *iterStmt* | *returnStmt* | *listStmt* | *inOP* | *outOP* | *varDeclaration*
13. *exprStmt* → *expression* **;**
14. *condStmt* → **if (** *simpleExp* **)** *primitiveStmt* | **if (** *simpleExp* **)** *primitiveStmt* **else** *primitiveStmt*
15. iterStmt → **for (** *assignExp***;** *simpleExp***;** *assignExp***)** *primitiveStmt*
16. *returnStmt* → **return** *expression* **;**
17. listStmt → *appendOPS* | *returnlistOPS* | *destroyheadOPS* | *mapfilterOPS*
18. *appendOPS* → *simpleExp* **: ID;**
19. *returnlistOPS* → *returnlistOP* **ID**
20. *returnlistOP* → ? | !
21. *destroyheadOPS* → % **ID;**
22. *mapfilterOPS* → *fcall* >> **ID;**
23. *expression* → *assingExp* | *simpleExp*
24. *assisgnExp* → **ID ASSIGN_OP** *expression*
25. *simpleExp* → *binLogicalExp*
26. *constOP* → **INT** | **FLOAT** | **LIST** | **NIL**
27. *inOP* → **READ (ID)**
28. *outOP* → **write (outConst) ;** | **writeln (outConst);**
29. *outConst* → **string** | **simpleExp**
30. *binLogicalExp* → *binLogicalExp* **binLogicalOp** *unLogicalExp* | *unLogicalExp*
31. *binLogicalOp* → || | &&& | ~
32. *unLogicalExp* → **unLogicalOP** *unLogicalExp* | *relationalExp*
33. *unLogicalOp* → *!*
34. *relationalExp* → *relationalExp* **relational_Op** *sumExp* | *sumExp*
35. *relationalOp* → == | ! = | >= | <= | > | <
36. *sumExp* → *sumExp sumOP mulExp* | *mulExp*
37. *sumOP* → + −
38. *mulExp* → *mulExp mulOP factor* | *factor* | **sumOp** *factor*
39. *mulOP* → ∗ /
40. *factor* → **ID** | *fCall* | (*simpleExp*) | *constOP*
41. *fCall* → **ID** (*callParams*) | **ID ( )**

42. *callParams* → *callParams, simpleExp* | *simpleExp*

Palavras reservadas: **read, write, writeln, int, float, string, char, if, else, for, return, append, headlist, taillist, destroyhead, map, filter, nil**
Símbolos reservados: **,|;|** (|) | **{** | **}** | ” | ′ | + | **-** | ***** | / | < | > | <= | >= | == | != ?**list** !**list** << >>

Label	Regular Expressions (Flex RegEx)
digit	**[0-9]**
letter	**[a-zA-Z]**
MAIN	**main**
ID	**{letter}+({letter}|{digit}|_|-^)***
NIL	**nil**
KEYWORD	**if|else|for|return|append|headlist|taillist|desr**
ARITHMETIC_OP	[+ - * /]
BIN_LOGICAL_OP	[&&, ~ , | |]
UN_LOGICAL_OP	**!{id}**
RELATIONAL_OP	[<,>,<=,>=,==,!=]
ASSIGN_OP	[=]
COMMENT	”//”.*
TYPE	**int|float|list**
IN	**read**
OUT	**write|writeln**
OUTCONST	**string**
INT	**-?{digit}+**
FLOAT	**-?{digit}*[.]{digit}+**
LIST	(”list ”)?**{id}**

Table 1. Rótulos e expressões regulares para os lexemas de linguagem

B Lexemas utilizados

- id: variáveis e funções
- digit: números
- add: +
- sub: -
- mult: *
- div: /
- and: &&
- or: ||
- different: !=
- not: !
- negate: ~
- greater: >

- greateq: >=
- smaller: <
- smalleq: <=
- equal: ==
- diff: !=
- assign: =
- main: *main*
- if: *if*
- else: *else*
- for: *for*
- return: *return*
- read: IO *read*
- write: IO *write*
- writeln: IO *writeln*
- int: tipo *int*
- float: tipo *float*
- list: tipo *list*
- append: *expression* **:** *list*
- headlist: retorna o valor do primeiro elemento de uma lista sem alterar a lista - **?**
- taillist: retorna a cauda de uma lista sem alterar a lista - **!**
- destroyhead: retorna a cauda de uma lista e remove o primeiro elemento. - **%**
- map: retorna uma lista com a função aplicada aos elementos do segundo elemento - >>
- filter: retorna a lista dos elementos do segundo argumento para os quais a função dada como primeiro argumento retorna valor diferente de zero - <<
- string: usadas tão somente para impressão
- (
-)
- {
- }
- ,
- .
- ;