

Tradutores: Analisador Sintático

Guilherme Andreúce Sobreira Monteiro - 14/0141961

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

Resumo 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 é 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* 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 encontrou o erro, além de também escrever qual foi o caractere ou lexemas que não pertence à gramática. Facilitando a correção caso necessária.

3 Analisador Sintático

Utilizando o Bison, ferramenta *opensource*, e a partir da gramática deste relatório, foi construído no arquivo *guillex.y* a gramática que será utilizada para construir o analisador, a árvore sintática abstrata e a tabela de símbolos.

A gramática foi implementada de forma LR(1) Canônica. A implementação utiliza uma *union* que recebe valores diferentes do analisador léxico para *String*, *Inteiro* e *Float* [5].

O novo programa também é responsável por declaração de *tokens*, não-terminais e regras da gramática. Para a representação da árvore sintática abstrata foram criadas funções dentro do *guillex.y* que contém a implementação da árvore sintática abstrata. Cada nó dessa árvore possui um campo para o valor representado possível, um campo para os tipos possíveis *Int*, *Nil*, *List*, *Float* e até 5 nós filhos. Para a tabela de símbolos, no mesmo arquivo foi construída uma função que guarda um identificador numérico, nome, se a entrada advém de uma função ou de uma variável e o tipo (se é *Int* ou *Float*) [7].

Para a construção foi utilizado uma estrutura de tabela hash com o auxílio da ferramenta *uthash* [8].

4 Analisador Semântico

Para a construção do analisador semântico está sendo utilizado a biblioteca *utstack* para poder construir e analisar o escopo. A partir da análise do escopo, será feita a análise para verificação de símbolos repetidos, verificação de main, utilização de variáveis e funções não declaradas, bem como todo o restante do analisador semântico. Também serão corrigidos erros apontados do trabalho anterior. No momento, o trabalho apenas está contando o escopo da variável e o escopo de origem. Já foi dado início as próximas etapas, porém, só serão apresentadas na versão final.

5 Arquivos de teste

Certifique-se de o ambiente é Linux Ubuntu 20.04 com o FLEX 2.6.4 e o Bison 3.7

O analisador sintático possui sete testes. Os cinco testes com 'correto' no nome são testes que o analisador lê corretamente os lexemas e apresenta corretamente a tabela de símbolos e a árvore sintática abstrata. Os dois testes com 'incorreto' no nome são testes que apresentam erros.

No caso dos arquivos que apresentam erro, temos no primeiro arquivo o primeiro erro em: Linha 1, Coluna 1 - um erro léxico '#'; segundo erro em: Linha 2, Coluna 6 - um erro sintático. No último caso esperado um ponto-vírgula ou parenteses e ele possui uma vírgula. No segundo arquivo temos o primeiro erro em: Linha 4, Coluna 5 - Um erro léxico '@'; segundo erro em: Linha 5, Coluna 10 - um erro sintático seguido. No último caso era esperado um ID ou tipoLista.

6 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 certifique-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), o make 4.2.1 e o Bison 3.7. A instalação da versão 3.7 do Bison é manual! Para executar abra a pasta principal do trabalho no terminal e digite *make*. Os testes rodados encontram-se na pasta *14_0141961/tests*. Os resultados obtidos encontram-se na pasta *14_0141961/results*. Para executar o Valgrind, no terminal digite *make valgrind*. Para alterar o teste que será executado, comente a primeira linha do make e descomente a que você gostaria de testar. Para limpar o trabalho de arquivos temporários ou criados a partir da compilação digite *make clean*.

Referências

1. Waldemar Celes, 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. Ashwani khemani, Bhupendra Rathore, Ajay Kumar, Nikhil Koyikkamannil, et al., Linked List Program in C <https://www.geeksforgeeks.org/linked-list-set-1-introduction/>. Acessado em 05 de agosto 2021
3. Python lists, oficial documentation, <https://docs.python.org/3/tutorial/introduction.html#lists>. Acessado em 14 de setembro 2021
4. Manual Flex, <https://westes.github.io/flex/manual/>. Acessado em 05 de agosto 2021
5. Canonical LR1 Parser, <http://www.cs.ecu.edu/karl/5220/spr16/Notes/Bottom-up/lr1.html>. Acessado em 28 de agosto de 2021.
6. Manual Bison, <https://www.gnu.org/software/bison/manual/>. Acessado em 10 de setembro de 2021.
7. Abstract Syntax Tree, <https://lambda.uta.edu/cse5317/notes/node26.html>. Acessado em 10 de setembro de 2021.
8. Manual Uthash, <https://troydhanson.github.io/uthash/>. Acessado em 11 de setembro de 2021.

A Linguagem da gramática

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

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	[&& ,]
RELATIONAL_OP	[<,>,<=,>=,==,!=]
ASSIGN_OP	[=]
COMMENT	" / " . *
TYPE	int float list
IN	read
OUT	write writeln
OUTCONST	string
INT	-?{digit}+
FLOAT	-?{digit}*[{.}{digit}]+
LIST	("list ")?{id}

Tabela 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: !=
- greater: >
- greateq: >=
- smaller: <
- smalleq: <=
- equal: ==

- 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
- (
-)
- {
- }
- ,
- .
- ;