

# Tradutores: Gerador de Código Intermediário

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 implementação de um tradutor. Em seguida é utilizado o programa Bison para a análise semântica e sintática. A linguagem proposta é uma sublinguagem C para lidar com listas. No primeiro estágio foi apresentada a geração de *tokens* sendo feita uma gramática utilizada para a implementação do tradutor. No segundo estágio é construído o analisador sintático que lida com erros gramaticais. No terceiro estágio é adicionado a análise semântica que lida com o tratamento de tipos, verificação de escopo e verificação de main. Por último, no quarto estágio, é adicionado a geração de código intermediário este que possui comandos interpretáveis pelo interpretador TAC.

**Keywords:** Tradutor · Flex · Analisador léxico · Analisador Sintático · Bison · Analisador Semântico · *tokens* · Gerador de Código Intermediário · TAC

## 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 lexemas. Esses lexemas são unidades básicas de significado para uma linguagem. Com uma gramática, o analisador léxico consegue identificar se esses *lexemas* 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 duas variáveis chamadas *word\_position* e *column* são incrementadas 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 onde o programa encontrou o erro, além de também escrever qual foi o caractere ou lexemas que não pertencem à gramática. Facilitando a correção caso necessária.

## 3 Analisador Sintático

Utilizando o Bison, ferramenta de código aberto, 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.

O analisador sintático foi implementado 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 desta árvore. 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]. Temos na estrutura de controle de símbolo a variável **UT\_hash\_handler** *hh* utilizada para iterar pelos símbolos.

## 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, parâmetros de função inválidos e checagem de tipos. Cada função *for*, *if*, *else* incrementa uma variável *scope* se aberto uma chave e decrementa se a chave é fechada. Cada função fechada desempilha o escopo. Quando um **ID** é lido, ele verifica na pilha se existe algum símbolo igual em algum escopo. Assim é possível verificar erros de declaração. Existem variáveis de controle para erros e número de parâmetros de uma função. Existe a checagem de retorno de função. Na árvore existem os tipos inteiro, float, list int, list float, null(nil), undefined.

## 5 Descrição do Gerador de Código Intermediário

Para a geração de código intermediário, foi gerada um arquivo .tac para ser utilizado com o programa TAC [9]. Para que o arquivo .tac seja gerado, não pode haver erro léxico ou sintático no código passado para o compilador. Foram implementadas funções para a criação dos comandos. A escrita do arquivo .tac será feita da seguinte maneira:

- Um arquivo novo será criado a partir do nome do arquivo com o código passado para o compilador.
- Durante a análise, uma ou mais linhas de código tac são acrescentadas em uma lista de acordo com a regra que foi lida.
- Ao fim da análise, é escrito primeiramente a linha ".table", seguida de ".code", e por fim, cada linha que estava presente na lista gerada. Para a geração desta fase será utilizada a biblioteca utlist e utstring, por facilitar algumas leituras e concatenação de strings. Além disso, as estruturas utilizadas são as mesmas das seções anteriores, tabela de símbolos, árvore. A tabela de símbolos conterá um novo campo para a inserção de uma variável de controle para registrador. A árvore também será modificada para adição de uma variável de controle para criação de expressões lógicas e aritméticas. Booleanos serão representados apenas com 0 para falso e 1 para verdadeiro.

## 6 Arquivos de teste

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 é esperado um ponto-vírgula ou parênteses e ele possui uma vírgula. Existe o erro semântico de função não declarada e não possui main. 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. Existe o erro semântico de variável não declarada.

## 7 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. Para executar abra a pasta principal do trabalho no terminal e digite *make*. Os testes executados 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 sob o Valgrind, 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. Vern Paxson - Manual Flex, <https://westes.github.io/flex/manual/>. - The Regents of the University of California, Acessado em 05 de agosto 2021
5. Karl Abrahamson - 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. Leonidas Fegaras - Abstract Syntax Tree, <https://lambda.uta.edu/cse5317/notes/node26.html>. Acessado em 10 de setembro de 2021.
8. Troy D. Hanson - Manual Uthash, <https://troydhanson.github.io/uthash/>. Acessado em 11 de setembro de 2021.
9. Claudia Nalon, Luciano Santos: the Three Address Code interpreter, Interpretador TAC <https://github.com/lhsantos/tac>. Acessado em 22 de Outubro 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.  $SimpleVarDeclaration \rightarrow \mathbf{TYPE\ ID} \mid \mathbf{TYPE\ LISTTYPE\ ID}$
7.  $SimpleFuncDeclaration \rightarrow \mathbf{TYPE\ ID} \mid \mathbf{TYPE\ LISTTYPE\ ID}$
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.  $condStmt \rightarrow ifStmt\ primitiveStmt \mid ifStmt\ primitiveStmt\ elseStmt \mid ifStmt \{ \} \mid ifStmt \{ \} \elseStmt$
15.  $ifStmt \rightarrow \mathbf{IF} ( simpleExp )$
16.  $elseStmt \rightarrow \mathbf{ELSE}\ primitiveStmt$
17.  $iterStmt \rightarrow \mathbf{for} ( assignExp; simpleExp; assignExp )\ primitiveStmt \mid \mathbf{for} ( simpleExp; simpleExp; assignExp )\ primitiveStmt$
18.  $returnStmt \rightarrow \mathbf{return}\ expression ;$
19.  $expression \rightarrow assignExp \mid simpleExp$
20.  $assignExp \rightarrow \mathbf{ID\ ASSIGN\_OP}\ expression$
21.  $simpleExp \rightarrow binLogicalExp \mid binListExp$
22.  $inOP \rightarrow \mathbf{READ (ID) ;} \mid \mathbf{READ ( ) ;}$
23.  $outOP \rightarrow \mathbf{write (outConst) ;} \mid \mathbf{writeln (outConst) ;}$
24.  $outConst \rightarrow \mathbf{string} \mid \mathbf{simpleExp}$
25.  $binListExp \rightarrow factor : factor \mid factor >> factor \mid factor << factor$
26.  $unaryListExp \rightarrow ? factor \mid ! factor$
27.  $binLogicalExp \rightarrow binLogicalExp \parallel relationalOp \mid binLogicalExp \&\& relationalOp \mid relationalExp$
28.  $relationalExp \rightarrow relationalExp > sumExp \mid relationalExp < sumExp \mid relationalExp <= sumExp \mid relationalExp >= sumExp \mid relationalExp == sumExp \mid relationalExp != sumExp \mid sumExp$
29.  $sumExp \rightarrow sumExp + mulExp \mid sumExp - mulExp \mid mulExp$
30.  $mulExp \rightarrow mulExp * factor \mid mulExp / factor \mid factor \mid + factor \mid - factor \mid unaryListExp$
31.  $factor \rightarrow \mathbf{ID} \mid fCall \mid ( simpleExp ) \mid \mathbf{INTEGER} \mid \mathbf{DECIMAL} \mid \mathbf{NIL}$
32.  $fCall \rightarrow \mathbf{ID} ( callParams ) \mid \mathbf{ID} ( )$
33.  $callParams \rightarrow callParams, simpleExp \mid simpleExp$

Palavras reservadas: **read**, **write**, **writeln**, **int**, **float**, **string**, **char**, **if**, **else**, **for**, **return**, **append**, **headlist**, **taillist**, **destroyhead**, **map**, **filter**, **nil**

Símbolos reservados: **,|;| ( | ) | { | } | " | ' | + | - | \* | / | < | > | <= | >= | == | != | ? | ! << >>**

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

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: =
- if: *if*
- else: *else*
- for: *for*
- return: *return*
- read: IO *read*
- write: IO *write*
- writeln: IO *writeln*
- int: tipo *int*
- float: tipo *float*

- append: :
- headlist: ?
- taillist: !
- destroyhead: %
- map: >>
- filter: <<
- string: usadas tão somente para impressão
- (
- )
- {
- }
- ,
- .
- ;