

# Analizador Semântico

Rômulo de Vasconcelos Feijão Filho - 140031260@aluno.unb.br  
Prof<sup>a</sup> Cláudia Nalon - nalon@unb.br

Universidade de Brasília, Brasília DF 70910-900, Brasil  
{pcr,cic}@unb.br

**Resumo** This is the third deliverable out of four for a Capstone project made for the Compilers course ministered by professor Cláudia Nalon at University of Brasilia. In this article we will present the motivation, description of the lexical analyser, description of the synthax analyser, description of the semanthic analyzer, description of test files and instructions for building and executing the project.

**Keywords:** Compilers · Lexical Analysis · Synthax Analysis · Semantic Analysis · FLEX · Bison.

## 1 Motivação

Para esse curso, é essencial essa implementação, tanto para alcançar o resultado final desejado do projeto da disciplina, quanto para o aprendizado e fixação dos conceitos ministrados, estes que também serão testados nas provas. A linguagem a ser desenvolvida foi projetada com o intuito de facilitar o uso de conjuntos (*set*) em programas escritos em C. Além do tipo *set*, também será implementado um tipo polimórfico, denominado *elem*. Essas funcionalidades são interessantes porque oferecem recursos a mais para o usuário, permitindo este trabalhar de forma mais ágil. Como a primeira fase de um compilador, a tarefa principal do analisador léxico é ler os caracteres da entrada do programa fonte, agrupá-los em lexemas e produzir como saída uma sequência de tokens para cada lexema no programa fonte[1]. A segunda etapa do processo de compilação, consiste da inserção dos tokens na gramática (saída do analisador léxico) no analisador sintático. Nessa etapa, a árvore sintática abstrata é gerada, assim como a tabela de símbolo contendo as declarações de variáveis e funções. A terceira etapa consiste da avaliação do código e seus valores semânticos, como por exemplo escopo de variáveis e funções, contenção da função main, verificação do número de parâmetros nas chamadas de funções, entre outros.

## 2 Descrição da Análise Léxica

A Análise Léxica foi feita com o auxílio da ferramenta de código aberto **FLEX**<sup>1</sup> (*Fast Lexical Analyzer Generator*)[2]. Foi criado um arquivo **newc.l**, contendo as

---

<sup>1</sup> <https://westes.github.io/flex/manual/>

definições e regras do analisador léxico da linguagem previamente definida e apresentada pela professora. Após a execução do comando **flex newc.l**, um arquivo **lex.yy.c** é gerado contendo o autômato finito determinístico que aceita a mesma linguagem das expressões regulares especificadas. Foi desenvolvido também, com o auxílio de variáveis locais e do FLEX, um mecanismo de detecção de caracteres não permitidos, indicando a linha e a coluna em que este se encontra. Os tokens gerados pelo analisador léxico serão passados para o analisador sintático, pois este irá gerar um tabela de símbolos e a árvore sintática abstrata.

### 3 Descrição da Análise Sintática

A Análise Sintática foi feita com o auxílio da ferramenta de código aberto **GNU Bison**[3]. A gramática foi implementada de forma que realize a análise de forma **LR(1) Canônica** (*Canonical Left-Right, Rightmost Derivation Parser*). Foi criado um arquivo **newcParser.y**, contendo uma union para que possamos receber valores de diferentes tipos vindos do analisador léxico (string, inteiro e float), as declarações dos tokens e dos não-terminais, as regras da gramática e a **main**, que agora foi passada do arquivo do analisador léxico para o sintático. Essa função recebe um arquivo como argumento e executa o analisador léxico e sintático sobre essa entrada. Ao executar o comando **bison -d -v newcParser.y**, são gerados pelo Bison dois arquivos, **newcParser.tab.h** e **newcParser.tab.c**, contendo os tokens declarados no arquivo **newcParser.y**. Com o auxílio da flag **--report=counterexamples** foi possível observar mais claramente situações de conflito na gramática, ficando assim mais fácil de depurá-la. Além do arquivo principal do analisador, foram criados mais dois arquivos para a etapa de análise sintática: **tree.h** e **symbol.h**. O arquivo **tree.h** consiste da implementação da árvore sintática abstrata e suas funções, cada nó dessa árvore possui um campo para cada tipo possível passado, até cinco nós filhos e um campo para determinar qual tipo de dado está sendo representado pelo nó (string, int, float, etc). Já o arquivo **symbol.h** é a implementação da tabela de símbolos. A tabela de símbolos guarda um nome, tipo do símbolo (variável ou função) e tipo da variável ou do retorno da função e esta foi montada a partir de uma estrutura hash com o auxílio da biblioteca uthash[4], essa tabela está sendo usada como uma variável global *extern* para facilitar o uso dessa tabela no arquivo **newcParser.y**.

### 4 Descrição da Análise Semântica

Para realizar a análise semântica do código, foi feito um analisador de passagem única e são utilizadas um conjunto de variáveis e estruturas de dados. A tabela de símbolos, implementada ainda na fase de análise sintática, agora possui dois novos campos, esses sendo o escopo da função, variável ou parâmetro encontrado, e, no caso de funções, quantos parâmetros são passados para essa função específica. Foi implementada também uma pilha com o auxílio da biblioteca **utstack**<sup>2</sup> para

<sup>2</sup> <https://troydhanson.github.io/uthash/utstack.html>

a validação dos escopos. Seu funcionamento é da seguinte forma, a cada função, *if*, *else*, *for* e *forall* aberto, uma variável **scope** é incrementada e adicionada na pilha, e a cada uma dessas funções fechadas, desempilhamos o primeiro elemento da pilha, removendo assim o escopo anterior. Quando um **ID** é lido, ele verifica para todos os elementos da pilha, se existe algum símbolo na tabela de símbolos com aquele escopo, dessa forma conseguimos avaliar cada escopo "pai" do escopo atual e se um deles contém a variável a ser buscada. Se algum ID já estiver presente na tabela de símbolos e estiver sendo declarado no mesmo escopo, um erro semântico será lançado. Existe também uma função de validação do número de parâmetros, aonde cada parâmetro acrescentado na declaração da função incrementamos seu atributo *parameters* na tabela de símbolos. No arquivo **newcParser.y**, algumas outras variáveis utilizadas são **errors**, aonde ficam guardados o número de erros encontrados, **semanticErrors**, que guarda a quantidade de erros semânticos, que no caso são interpretados como warning, não impedindo então a construção da árvore sintática, **args\_params**, que guarda o número de parâmetros passados como argumento para uma função e, por fim, **has\_main**, essa que guarda o número de ocorrências da função **main()**, se não houver exatamente uma ocorrência, um erro semântico será lançado. Existe também a checagem de retorno de funções, caso uma função exija um retorno e este não seja passado, ou vice-versa, o analisador semântico irá produzir uma mensagem de erro, isso também acontecerá caso seja passado um retorno de tipo inválido. Foi feita também uma análise de operações aritméticas, operações de set e de atribuições, caso o tipo esteja diferente, será feita uma coerção, se possível. As coerções podem ocorrer em operações aritméticas e atribuição. Em operações aritméticas, se os operadores forem do tipo "int" e "float", a coerção será feita para o tipo "float". Já em atribuições, o tipo será o mesmo da variável, a menos que um dos valores seja do tipo "set" ou "elem", o primeiro caso não será permitido, no segundo caso, a variável receberá o valor atribuído a ela. Os tipos na árvore anotada são 'i' para inteiro, 'f' para float, 's' para set, 'u' para valor indefinido e 'x' para vazio. O valor 'u' é utilizado caso tente ser feita operação de set usando algum valor que não seja um set e o valor 'x' foi usado principalmente para avaliação do retorno de funções.

## 5 Descrição dos Arquivos de Teste

Existem oito arquivos de teste implementados, todos estes se encontram na pasta **tests**. Os arquivos **test1.nc** e **test2.nc** devem passar pelo analisador léxico, sintático e semântico sem encontrar erros, pois estes contém apenas lexemas identificáveis, fazem parte da gramática definida e não contém erros semânticos. Os arquivos **test3.nc** e **test4.nc** devem apresentar apenas erros semânticos, indicando qual erro ocorreu e sua posição no código. O arquivo **test3.nc** deve apresentar erro de variável não declarada nas linhas 2 e 3, função não declarada na linha 3, função com retorno de tipo errado na linha 7, variável recebendo valor de tipo errado na linha 10, função sendo invocada com número de parâmetros inválidos na linha 12, operação de set com tipo errado nas linhas 14 e 15, função

sem retorno esperado na linha 34 e função `main` não encontrada. Já o arquivo de teste **test4.nc**, deverá apresentar variáveis não declaradas nas linhas 4, 11, 13 e 14, função `main` com retorno inesperado na linha 21 e mais de uma função `main` encontrada.

Os arquivos de teste **test5.nc** e **test6.nc** devem apresentar erros na fase de análise sintática, apresentando apenas a tabela de símbolos e indicando as linhas e colunas. No arquivo **test5.nc**, deverão ser apontados dois erros, um na linha 10 por comando `READ` inválido e outro na linha 14, porém o erro ocorre na linha 12, devido a falta de um ponto-vírgula. O arquivo **test6.nc** deverá também apresentar dois erros, ambos na linha 8, pois ambos o `"if"` e o `"for"` foram construídos de maneira errada.

Já os arquivos **test7.nc** e **test8.nc** devem apresentar erros, ainda na fase de análise léxica, indicando o que causou tal erro e sua localização no arquivo. O arquivo de teste **test7.nc** deve apresentar dois erros após a execução do analisador léxico: `"@ at line: 2, column: 7"` e `"@ at line: 4, column: 9"`, isso também gera um erro sintático na linha 2. O arquivo de teste **test8.nc** também deve apresentar dois erros após a execução do analisador léxico: `"| atline : 3, column : 9"`, `"\atline : 3, column : 11"e" & at line: 4, column: 7"`, consequentemente apresentando também erros sintáticos nas linhas 3 e 6.

## 6 Instruções de Compilação e Execução

**Antes de executar o programa, por favor verifique as versões do FLEX e do Bison instalados na máquina. As versão do flex utilizada durante o processo de implementação foi a 2.6.4, e para o bison, 3.7.4. Esse programa foi desenvolvido em ambiente ubuntu 20.10.**

Para a compilação do programa e execução dos arquivos de teste, foi criado um arquivo **makefile**, a fim de simplificar e acelerar esse processo. Para executar os arquivos de teste, basta entrar na pasta **src**, utilizando o comando **cd src** e, dentro dessa, executar o comando **make runtests**. Na pasta **results** serão criados os arquivos resultantes de cada caso de teste. Caso deseje executar cada comando de compilação separadamente, também dentro da pasta **src** execute, **flex newc.l**, seguido de **bison -d -v newcParser.y** e também **gcc lex.yy.c -g newcParser.tab.c -ll -g -Wall -o parser**, terminando então o processo de compilação; finalmente, para executar o programa: **./parser ../tests/<nome-do-arquivo-entrada>**, que o resultado aparecerá no terminal.

## Referências

1. Aho, Alfred V., Lam, Monica S., Sethi, Ravi, Ullman, Jeffrey D.: Compiladores: Princípios, Técnicas e Ferramentas. 2nd edn. Pearson, (1999)
2. Paxson, Vern: Lexical Analysis With Flex, for Flex 2.6.2, <https://westes.github.io/flex/manual/> (2001). Último acesso em: 15 de Fevereiro de 2021

3. Free Software Foundation: GNU Bison - The Yacc-compatible Parser Generator, <https://www.gnu.org/software/bison/manual/> (2020). Último acesso em: 17 de Março de 2021
4. D. Hanson, Troy: uthash User Guide, <https://www.cs.bu.edu/~jappavoo/Resources/psml/apps/hashty/uthash/doc/userguide.html> (2021). Último acesso em: 17 de Março de 2021

## A Gramática

A gramática da linguagem a ser implementada foi montada da seguinte maneira:

```

program → declarations_list
declarations_list → declarations_list declaration | declaration | ERROR
declaration → var_dec | func_dec
var_dec → TYPE ID ;
func_dec → TYPE ID ( params_list ) { statement_list } |
           TYPE MAIN ( params_list ) { statement_list }
params_list → params_list , parameter | parameter | ε | ERROR
parameter → TYPE ID
statement_list → statement_list statement | ε | ERROR
statement → expression_statement | ret_st | var_dec | io_ops | basic_ops
expression_statement → expression ;
basic_ops → if_ops |
           for_statement { statement_list } |
           forall_statement set_op ; |
           forall_statement { statement_list }
for_statement → FOR ( for_body )
forall_statement → FORALL ( in_set )
for_body → expression_statement expression_statement expression |
          ; expression_statement expression
if_ops → if_statement statement |
        if_statement statement else_statement
        if_statement { statement_list } |
        if_statement { statement_list } else_statement
if_statement → IF ( operation )
else_statement → ELSE statement
               ELSE { statement_list }
ret_st → RETURN expression ;
io_ops → READ ( ID ) ; |
        READ ( ) ; |
        WRITE ( expression ) ; |
        WRITELN ( expression ) ;
expression → set_op | func_call | assign_value
term → math_term |
      str_term

```

```

math_term → ID | INTEGER | DECIMAL | ( expression )
str_term  → CHAR | STRING | EMPTY | ERRORTOKEN
math_op   → math_op ADD math_op_muldiv |
            math_op SUB math_op_muldiv |
            math_op_muldiv
math_op_muldiv → math_op_muldiv DIV math_term |
                  math_op_muldiv MULT math_term |
                  math_term
set_op      → ADD.SET ( set_op ) |
              REMOVE ( set_op ) |
              EXISTS ( set_op ) |
              operation
operation    → math_op |
              in_set |
              IS_TYPE ( expression ) |
              term SMALLER expression |
              term GREATER expression |
              term SMALLEQ expression |
              term GREATEQ expression |
              term EQUALS expression |
              term DIFFERENT expression |
              term OR expression |
              term AND expression |
              NEG expression
func_call   → ID ( args_list ) |
              str_term
in_set      → term IN expression
args_list  → args_list , term |
              term |
              ε
assign_value → ID = expression

letter → [a-zA-Z]
digit  → [0-9]
ID     → letter ( letter | digit | _ | - ) *
INTEGER → { digit } +
DECIMAL → { digit } * . { digit } +
CHAR    → { letter }
STRING  → \ " ( \\ . | [ ^ " \\ ) * \ "
TYPE    → int | float | set | elem
DIV     → /
MULT    → *
ADD     → +
SUB     → -
EMPTY   → "EMPTY"

```

**MAIN** → main  
**FOR** → for  
**FORALL** → forall  
**IF** → if  
**ELSE** → else  
**RETURN** → return  
**READ** → read  
**WRITE** → write  
**WRITELN** → writeln  
**ADD.SET** → add  
**REMOVE** → remove  
**EXISTS** → exists  
**IS\_TYPE** → is\_set  
**IN** → in  
**SMALLER** → <  
**GREATER** → >  
**SMALLEQ** → <=  
**GREATEQ** → >=  
**EQUALS** → ==  
**DIFFERENT** → !=  
**OR** → ||  
**AND** → &&  
**NEG** → !