Universidade Federal de Itajubá - Campus Itabira

Instituto de Ciência e Tecnologia

Engenharia da Computação

Relatório de Compiladores

Desenvolvimento de um Compilador para a Linguagem MINI

Guilbert Wilkerson Marques Oliveira RA: 2019004349

Guilherme Henrique Dias Costa RA: 2019009908

Itabira - MG

2023

Introdução

Um compilador é um programa capaz de converter um código fonte, escrito em uma linguagem específica, para um programa equivalente em outra linguagem, denominada linguagem objeto. O processo de conversão realizado por um compilador pode ser subdividido em duas partes: a primeira, conhecida como fase de análise, é responsável por realizar a análise léxica, sintática e semântica, com o objetivo de verificar se o código escrito na linguagem fonte está em conformidade com as regras estabelecidas pela definição da linguagem. A segunda fase, denominada fase de síntese, é responsável por construir um novo programa escrito na linguagem objeto.

Neste trabalho, abordaremos a construção de um compilador para a linguagem MINI. Apresentaremos uma abordagem simplificada para cada uma das fases de um compilador, utilizando a linguagem Java, com o auxílio das ferramentas JFlex e Java Cup.

1) Especificação da Linguagem

A linguagem Mini é uma linguagem de programação fictícia baseada em pascal , sua estrutura e funcionamento é descrito pela seguinte gramática.

Backus naur form
program ::= **program** identifier body
body ::= [**declare** decl-list] **begin** stmt-list **end**
decl-list ::= decl {";" decl}*
decl ::= type ident-list
ident-list ::= identifier {"," identifier}*
type ::= **integer** | **decimal**
stmt-list ::= stmt {";" stmt}

stmt ::= assign-stmt | if-stmt | while-stmt

| read-stmt | write-stmt

assign-stmt ::= identifier ":=" simple_expr

if-stmt ::= **if** condition **then** stmt-list **end**

| **if** condition **then** stmt-list **else** stmt-list **end**

condition ::= expression

do-while-stmt ::= **do** stmt-list stmt-suffix

stmt-suffix ::= **while** condition

for-stmt ::= **for** assign-stmt **to** condition **do** stmt-list **end** //o for é opcional

while-stmt ::= **while** condition do stmt-list **end** // o while..do é opcional

read-stmt ::= **read** "(" identifier ")"

write-stmt ::= **write** "(" writable ")"

writable ::= simple-expr | literal

expression ::= simple-expr | simple-expr relop simple-expr

simple-expr ::= term | simple-expr addop term | "(" simaddopple-expr ")" ? simple-expr ":" simple-expr

term ::= factor-a | term mulop factor-a

fator-a ::= factor | **not** factor | "-" factor

factor ::= identifier | constant | "(" expression ")"

relop ::= "=" | ">" | ">=" | "<" | "<=" | "<>"

addop ::= "+" | "-" | **or**

mulop ::= "*" | "/" | **mod** | **and**

shiftop ::= "<<" | ">>" | "<<<" | ">>>"

constant ::= digit {digit}*

literal ::= " “" {caractere} "”"

identifier ::= letter {letter | digit}*

letter ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J"

| "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T"

| "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d"

| "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n"

| "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x"

| "y" | "z"

digit ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

caractere ::= um dos 256 caracteres do conjunto ASCII, exceto "

A) Características da Gramática

- Todas as palavras escritas em negritos são terminais da linguagem MINI e também são palavras chave.
- Todas as palavras entre aspas duplas são terminais da linguagem MINI
- A linguagem não reconhece o símbolo de aspas
- Comentários são iniciados com símbolo %

2) Analisador Léxico

A Análise Léxica é a primeira fase do processo de compilação. O principal objetivo do analisador léxico é percorrer os caracteres do programa fonte, agrupá-los em unidades chamadas lexemas, identificar os tokens associados a cada lexema e gerar uma sequência de tokens como saída. A sequência de tokens resultante da análise léxica é armazenada em uma estrutura denominada tabela de símbolos e o token serve como entrada para a análise sintática.

B) Metodologia

Para implementar o analisador léxico da linguagem MINI, utilizamos o auxílio do JFlex, na versão 1.7.0. aaO JFlex é um gerador de analisador léxico, também conhecido como gerador de scanner, desenvolvido em Java. Sua principal função é receber uma especificação contendo um conjunto de expressões regulares e suas ações correspondentes.

A instalação do JFlex no Linux pode ser realizada de maneira simples através do apt-get, com o seguinte comando:

```
sudo apt-get install jflex
```

Um arquivo jflex segue a seguinte especificação :

scanner. jlex

```
%%

%{
   // Código Java inserido aqui
%}

// Definições de expressões regulares


// Regras e ações associadas
%%

%%

// Código Java adicional, se necessário
```

C) Desenvolvimento

O primeiro estágio da análise léxica consiste na construção de uma classe capaz de armazenar as informações relevantes de um token na linguagem. Para isso, desenvolvemos a classe Yytoken, que contém os atributos de um token específico da linguagem.

Yytoken.java

```
import java_cup.runtime.Symbol;
import java.io.IOException;



public class Yytoken extends Symbol{
```

```
private String name; // Nome do grupo
private int index; // Id do Grupo de tokens (identificadores ,ifs, ....)
private String lexema; // lexema
private int line;
private int charBegin;
private int charEnd;
private Integer PosicaoTs;

public Yytoken(int index, String lexema, int line, int charBegin, int charEnd, String name) {
   super(index);
   this.index = index;
   this.lexema = lexema;
   this.line = line;
   this.charBegin = charBegin;
   this.charEnd = charEnd;
   this.name = name;
}
public Yytoken(String lexema, String name,int index){
   super(index);
   this.index = index;
   this.lexema = lexema;
   this.name = name;
}
public Yytoken(int index){
     super(index);
     this.index = index;
}

// Getters
public int getIndex() {
   return index;
}

public String getName() {
   return name;
}

public String getlexema() {
   return lexema;
}

public int getLine() {
   return line;
}

public int getCharBegin() {
```

```
    return charBegin;
}

public int getCharEnd() {
    return charEnd;
}
public int getPosicaoTs(){
    return this.PosicaoTs;
}

// Setters
public void setIndex(int index) {
    this.index = index;
}

public void setName(String name) {
    this.name = name;
}

public void setlexema(String lexema) {
    this.lexema = lexema;
}

public void setLine(int line) {
    this.line = line;
}

public void setCharBegin(int charBegin) {
    this.charBegin = charBegin;
}

public void setCharEnd(int charEnd) {
    this.charEnd = charEnd;
}
 public void setPosicaoTs( Integer ID){
     this.PosicaoTs = ID;
}

@Override
public String toString() {
    return "Posicao na TS: " + PosicaoTs +
            "\nlexema: " + lexema +
            "\nIndex: " + index +
            "\nName: " + name +
            "\nLine: " + line +
            "\nBegin: " + charBegin +
            "\nEnd: " + charEnd;
```

```
    }
}
```

Dentro da classe token, encontram-se os atributos "Nome", "Lexema", "Line", "CharBegin" e "CharEnd", responsáveis por armazenar informações relevantes sobre o token identificado. Além disso, as variáveis "PosiçãoTS" e "Index" são utilizadas para a integração do analisador léxicos com os demais estágios do compilador. A primeira armazena uma chave de acesso ao token na tabela de símbolos, enquanto a segunda é um inteiro que identifica o token para o analisador semântico.

Após o desenvolvimento da classe que armazena o token, é necessário criar uma classe para gerar a tabela de símbolos.

TabelaSimbolo.java

```
import java.util.HashMap;
import java.util.Map;

class Dado {
    private String lexema;
    private String nome;
    private int index;

    public Dado(String lexema, String nome, int index) {
        this.lexema = lexema;
        this.nome = nome;
        this.index = index;
    }

    public String getLexema() {
        return lexema;
    }

    public String getNome() {
        return nome;
    }

    public int getIndex() {
```

```
        return index;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Dado dado = (Dado) o;
        // Compare os campos relevantes para determinar a igualdade
        if (index != dado.index) return false;
        if (lexema != null ? !lexema.equals(dado.lexema) : dado.lexema != null) return false;
        return nome != null ? nome.equals(dado.nome) : dado.nome == null;
    }

    @Override
    public int hashCode() {
        int result = lexema != null ? lexema.hashCode() : 0;
        result = 31 * result + (nome != null ? nome.hashCode() : 0);
        result = 31 * result + index;
        return result;
    }
}

public class TabelaSimbolos {

    private Map<Integer, Dado> tabela;
    private int proximoId;

    public TabelaSimbolos() {
        this.tabela = new HashMap<>();
        this.proximoId = 0;
        inicializarTabelaPalavrasReservadas();
    }
    public Integer obterChave(Dado dado) {
        for (Map.Entry<Integer, Dado> entry : tabela.entrySet()) {
            if (entry.getValue().equals(dado)) {
                return entry.getKey();
            }
        }
        return null;
    }

    public void  adicionarEntrada(Yytoken token) {
        Dado dado = new Dado(token.getlexema(), token.getName(), token.getIndex());
        if (!contemToken(token)){
            int id = obterProximoId();
            tabela.put(id, dado);
```

```
            token.setPosicaoTs(id);
        }else{
            int id  = obterChave(dado);
            token.setPosicaoTs(id);
        }

    }


    public Dado obterDado(int posição) {
        return tabela.get(posição);
    }

    public boolean contemToken(Yytoken token) {
        Dado dado = new Dado(token.getlexema(), token.getName(), token.getIndex());
        return tabela.containsValue(dado);
    }

    public void imprimirTabela() {
        System.out.println("Tabela de Símbolos:");
        for (Map.Entry<Integer, Dado> entry : tabela.entrySet()) {
            System.out.println("ID: " + entry.getKey() +
                    ", Lexema: " + entry.getValue().getLexema() +
                    ", Nome: " + entry.getValue().getNome() +
                    ", Index: " + entry.getValue().getIndex());
        }
    }

    private int obterProximoId() {
        return proximoId++;
    }
    private void inicializarTabelaPalavrasReservadas() {
        adicionarEntrada(new Yytoken( "program", "Palavra reservada: program" ,
Sym.PROGRAM));
        adicionarEntrada(new Yytoken("declare", "Palavra reservada: declare", Sym.DECLARE));
        adicionarEntrada(new Yytoken("begin", "Palavra reservada: begin", Sym.BEGIN));
        adicionarEntrada(new Yytoken("integer", "Palavra reservada: integer", Sym.INTEGER));
        adicionarEntrada(new Yytoken("decimal", "Palavra reservada: decimal", Sym.DECIMAL));
        adicionarEntrada(new Yytoken("if", "Palavra reservada: if", Sym.IF));
        adicionarEntrada(new Yytoken("then", "Palavra reservada: then", Sym.THEN));
        adicionarEntrada(new Yytoken("else", "Palavra reservada: else", Sym.ELSE));
        adicionarEntrada(new Yytoken ("for", "Palavra reservada: for", Sym.FOR));
        adicionarEntrada(new Yytoken ("end", "Palavra reservada: end", Sym.END));
        adicionarEntrada(new Yytoken("do", "Palavra reservada: do", Sym.DO));
        adicionarEntrada(new Yytoken("to", "Palavra reservada: to", Sym.TO));
        adicionarEntrada(new Yytoken ("while", "Palavra reservada: while", Sym.WHILE));
        adicionarEntrada(new Yytoken("read", "Palavra reservada: read", Sym.READ));
```

```
        adicionarEntrada(new Yytoken ("write", "Palavra reservada: write", Sym.WRITE));
        adicionarEntrada(new Yytoken ("or", "Palavra reservada: or", Sym.OR));
        adicionarEntrada(new Yytoken("and", "Palavra reservada: and", Sym.AND));
        adicionarEntrada(new Yytoken("mod", "Palavra reservada: mod", Sym.MOD));
        adicionarEntrada(new Yytoken("not", "Palavra reservada: not", Sym.NOT));
    }
}
```

Nessa classe, foi empregada uma tabela hash para armazenar as informações relevantes de um token. A classe "Dado" é utilizada para armazenar os principais pontos do token, e ao final é construída uma tabela hash para representar a nossa tabela de símbolos. As palavras-chave da linguagem Mini são inseridas antecipadamente na tabela de símbolos para tornar o código mais performático, e as informações presentes na classe "Sym" são geradas pelo analisador Sintático.

Por fim , é possível criar um arquivo jflex para gerar um scanner capaz de identificar os tokens presentes na linguagem. A primeira etapa é estabelecer os macros da linguagem:

scanner.java

```
WS = [\n| |\t|\r]
letter = [A-Za-z]
digit = [0-9]
INTEGER_CONSTANT = {digit}+
DECIMAL_CONSTANT = {digit}+\.{digit}+
identifier = {letter}({letter}|{digit})*
caractere = [^0-9A-Za-z\" \t\r\n]
Comentario = "%"(.*)\n
Literal = ``[^`]*´´
```

Esses macros são usados para identificar facilmente alguns elementos relevantes da linguagem . O objetivo do léxico é inserir um token na tabela de símbolos e entregar ao sintático o token em conjunto com a posição que aquele token ocupa na tabela de símbolos . Para realizar essa ação para cada elemento da linguagem foi criado essa ação :

scanner.jflex

```
YYINITIAL>{
"program" {
   Yytoken token = new Yytoken(Sym.PROGRAM, yytext(), yyline, yychar, yychar + 7,
"Palavra reservada: program");
   tabelaSimbolos.adicionarEntrada(token);
   return token;
}

"declare" {
   Yytoken token = new Yytoken(Sym.DECLARE, yytext(), yyline, yychar, yychar + 7, "Palavra
reservada: declare");
   tabelaSimbolos.adicionarEntrada(token);
   return token;
}

"begin" {
   Yytoken token = new Yytoken(Sym.BEGIN, yytext(), yyline, yychar, yychar + 5, "Palavra
reservada: begin");
   tabelaSimbolos.adicionarEntrada(token);
   return token;
}

"integer" {
   Yytoken token = new Yytoken(Sym.INTEGER, yytext(), yyline, yychar, yychar + 7, "Palavra
reservada: integer");
   tabelaSimbolos.adicionarEntrada(token);
   return token;
}

"decimal" {
   Yytoken token = new Yytoken(Sym.DECIMAL, yytext(), yyline, yychar, yychar + 7, "Palavra
reservada: decimal");
      tabelaSimbolos.adicionarEntrada(token);

   return token;
}

"if" {
   Yytoken token = new Yytoken(Sym.IF, yytext(), yyline, yychar, yychar + 2, "Palavra
reservada: if");
      tabelaSimbolos.adicionarEntrada(token);

   return token;
}

"then" {
   Yytoken token = new Yytoken(Sym.THEN, yytext(), yyline, yychar, yychar + 4, "Palavra
```

```
reservada: then");
        tabelaSimbolos.adicionarEntrada(token);

    return token;
}

"else" {
    Yytoken token = new Yytoken(Sym.ELSE, yytext(), yyline, yychar, yychar + 4, "Palavra
reservada: else");
        tabelaSimbolos.adicionarEntrada(token);

    return token;
}

"for" {
    Yytoken token = new Yytoken(Sym.FOR, yytext(), yyline, yychar, yychar + 3, "Palavra
reservada: for");
        tabelaSimbolos.adicionarEntrada(token);

    return token;
}

// …….
WS} {
    // Ignorar espaços em branco e quebras de linha
}
{Comentario} {
    // Ignorar comentarios
}

{Literal}  {
    Yytoken token = new Yytoken(Sym.TEXTO, yytext(), yyline, yychar, yychar + yylength(),
"Literal");
        tabelaSimbolos.adicionarEntrada(token);

    return token;
}

{INTEGER_CONSTANT} {
    Yytoken token = new Yytoken(Sym.INTEGERCONSTANT, yytext(), yyline, yychar, yychar +
yylength(), "Numero Inteiro ");
        tabelaSimbolos.adicionarEntrada(token);

    return token;
}
/……
}
```

As regras completas podem ser encontradas no código presente do repositório . Para relacionar o léxico com as demais etapas do código foi adicionado o seguinte trecho na área de usuário do código jflex.

scanner.java

```
%{

   TabelaSimbolos tabelaSimbolos =  new TabelaSimbolos() ;
   public TabelaSimbolos getTabelaSimbolos (){
      return tabelaSimbolos;
   }
   Ylex(String  in) {
    main(in);

  }
%}
```

D) Resultados

Para realizar os testes necessários para validar o analisador léxico, construímos um shell script capaz de testar os sete casos de teste solicitados, além de dois testes criados pelo grupo.

test.sh

```
#!/bin/bash

# Cria os diretórios de saída se não existirem
mkdir -p Test/Saidas

# Loop de 1 a 9 para processar os arquivos de entrada
for i in {1..9}; do
  input_file="Test/Entradas/Entrada${i}.txt"
  output_file="Test/Saidas/Saida${i}.txt"

  # Verifica se o arquivo de entrada existe
  if [ ! -f "$input_file" ]; then
    echo "Erro: Arquivo de entrada não encontrado: $input_file"
    exit 1
```

```
  fi

  # Compila o código Java
  javac Yylex.java9

  # Executa o programa Java com o arquivo de entrada correspondente e redireciona a saída
  java Yylex "$input_file" > "$output_file"

  # Limpa os arquivos .class gerados durante a compilação
  rm *.class
done
```

Primeiramente é necessário compilar o arquivo scanner.flex para gerar o nosso identificador léxico, usando o terminal realize o comando

```
jflex scanner.jflex
```

Depois é necessário executar o arquivo de teste :

```
./test.sh
```

Após a execução do arquivo de teste, foram gerados nove arquivos na pasta Teste/Saída, cada um contendo a resposta de um dos testes realizados. Como exemplo, apresentamos a resposta do primeiro teste.

```
line: 1 char: 0 match: --program--
action [42] { Yytoken token = new Yytoken(Sym.PROGRAM, yytext(), yyline, yychar, yychar +
7, "Palavra reservada: program");
   tabelaSimbolos.adicionarEntrada(token);
   return token; }
Posicao na TS: 0
lexema: program
Index: 2
```

```
Name: Palavra reservada: program
Line: 0
Begin: 0
End: 7
line: 1 char: 7 match: -- --
action [336] { // Ignorar espaços em branco e quebras de linha }
line: 1 char: 8 match: --teste1--
action [329] { Yytoken token = new Yytoken(Sym.IDENTIFIER, yytext(), yyline, yychar, yychar
+ yylength(), "Identifier");
        tabelaSimbolos.adicionarEntrada(token);

    return token; }
Posicao na TS: 19
lexema: teste1
Index: 21
Name: Identifier
Line: 0
Begin: 8
End: 14
line: 1 char: 14 match: --\u000A--
action [336] { // Ignorar espaços em branco e quebras de linha }
line: 2 char: 15 match: --\u000A--
action [336] { // Ignorar espaços em branco e quebras de linha }
line: 3 char: 16 match: --declare--
action [48] { Yytoken token = new Yytoken(Sym.DECLARE, yytext(), yyline, yychar, yychar +
7, "Palavra reservada: declare");
    tabelaSimbolos.adicionarEntrada(token);
    return token; }
Posicao na TS: 1
lexema: declare
Index: 3
Name: Palavra reservada: declare
Line: 2
Begin: 16
End: 23
line: 3 char: 23 match: --\u000A--
action [336] { // Ignorar espaços em branco e quebras de linha }
line: 4 char: 24 match: -- --
action [336] { // Ignorar espaços em branco e quebras de linha }
line: 4 char: 25 match: -- --
action [336] { // Ignorar espaços em branco e quebras de linha }
line: 4 char: 26 match: -- --
action [336] { // Ignorar espaços em branco e quebras de linha }
line: 4 char: 27 match: -- --
action [336] { // Ignorar espaços em branco e quebras de linha }
line: 4 char: 28 match: --integer--
action [60] { Yytoken token = new Yytoken(Sym.INTEGER, yytext(), yyline, yychar, yychar +
```

7, "Palavra reservada: integer");
    tabelaSimbolos.adicionarEntrada(token);
    return token; }
Posicao na TS: 3
lexema: integer
Index: 5
Name: Palavra reservada: integer
Line: 3
Begin: 28
End: 35
…….. Continua ……
Tabela de Símbolos:
ID: 0, Lexema: program, Nome: Palavra reservada: program, Index: 2
ID: 1, Lexema: declare, Nome: Palavra reservada: declare, Index: 3
ID: 2, Lexema: begin, Nome: Palavra reservada: begin, Index: 4
ID: 3, Lexema: integer, Nome: Palavra reservada: integer, Index: 5
ID: 4, Lexema: decimal, Nome: Palavra reservada: decimal, Index: 6
ID: 5, Lexema: if, Nome: Palavra reservada: if, Index: 7
ID: 6, Lexema: then, Nome: Palavra reservada: then, Index: 8
ID: 7, Lexema: else, Nome: Palavra reservada: else, Index: 9
ID: 8, Lexema: for, Nome: Palavra reservada: for, Index: 10
ID: 9, Lexema: end, Nome: Palavra reservada: end, Index: 11
ID: 10, Lexema: do, Nome: Palavra reservada: do, Index: 12
ID: 11, Lexema: to, Nome: Palavra reservada: to, Index: 13
ID: 12, Lexema: while, Nome: Palavra reservada: while, Index: 14
ID: 13, Lexema: read, Nome: Palavra reservada: read, Index: 15
ID: 14, Lexema: write, Nome: Palavra reservada: write, Index: 16
ID: 15, Lexema: or, Nome: Palavra reservada: or, Index: 17
ID: 16, Lexema: and, Nome: Palavra reservada: and, Index: 18
ID: 17, Lexema: mod, Nome: Palavra reservada: mod, Index: 19
ID: 18, Lexema: not, Nome: Palavra reservada: not, Index: 20
ID: 19, Lexema: teste1, Nome: Identifier, Index: 21
ID: 20, Lexema: a, Nome: Identifier, Index: 21
ID: 21, Lexema: ,, Nome: Comma, Index: 25
ID: 22, Lexema: b, Nome: Identifier, Index: 21
ID: 23, Lexema: c, Nome: Identifier, Index: 21
ID: 24, Lexema: ;, Nome: Semicolon, Index: 26
ID: 25, Lexema: result, Nome: Identifier, Index: 21
ID: 26, Lexema: (, Nome: Left Parenthesis, Index: 28
ID: 27, Lexema:), Nome: Right Parenthesis, Index: 27
ID: 28, Lexema: :=, Nome: ASSIGN_OP, Index: 24
ID: 29, Lexema: 10, Nome: Numero Inteiro , Index: 22
ID: 30, Lexema: *, Nome: Mult, Index: 37
ID: 31, Lexema: /, Nome: Div, Index: 38
ID: 32, Lexema: +, Nome: Plus, Index: 31
ID: 33, Lexema: 5, Nome: Numero Inteiro , Index: 22

Ao final do arquivo é mostrado como ficou a tabela de símbolos após o final da execução do programa.

2) Analisador Sintático

A Análise Sintática é uma etapa subsequente à Análise Léxica no processo de compilação. Enquanto a Análise Léxica lida com a identificação de tokens, a Análise Sintática concentra-se na estrutura gramatical do programa fonte. Seu principal propósito é reconhecer a organização e a hierarquia das construções sintáticas presentes no código fonte. O Analisador Sintático utiliza a sequência de tokens gerada pela Análise Léxica como entrada.

A) Metodologia

Para implementar o analisador sintático da linguagem MINI, utilizamos a ferramenta Java Cup, versão 11b. Java Cup é um gerador de analisador sintático, também conhecido como gerador de parser, desenvolvido em Java. Para utilizá-lo, é necessário fazer o download dos arquivos JAR e incluí-los como bibliotecas (libs) no seu projeto Java.

A gramática presente no arquivo parser.cup não é uma tradução exata da gramática da linguagem MINI. Para compatibilidade com o Java Cup, foram realizadas algumas alterações a fim de adequar as regras do Java Cup e resolver problemas de shift e reduce.

B) Desenvolvimento

O arquivo jflex desenvolvido pelo grupo é descrito na tabela abaixo :

Parser.cup
import java_cup.runtime.Symbol; /* Define o conjunto de terminais */ terminal PROGRAM, DECLARE, BEGIN, INTEGER, DECIMAL, IF, THEN, ELSE, FOR, END, DO, TO, WHILE, READ, WRITE, OR, AND, MOD, NOT, IDENTIFIER, INTEGERCONSTANT,

```
DECIMALCONSTANT,
  ASSIGN_OP, COMMA, SEMICOLON, RIGHT_PAREN, LEFT_PAREN,
  EQ, MINUS, PLUS, LT, LE, NE, GT, GE, MULT, DIV, DGT, DLT, TGT, TLT,
QUESTION_MARK, COLON, TEXTO;

/* Define o conjunto de não-terminais */
non terminal
  Startprogram, Body, DeclList, Decl, IdentList, Type, Dlist, Commoindetifier, Semicolondecl,
  Stmtlist, Stmt, SemiColonstmt, Assignstmt, Simpleexpr, Term,
  FactoA, Factor, Expression, Relop, Addop, Mulop,
  Shiftop, Literal, Condition, Writable, Readstmt,
  Writestmt, Stmtsuffix, Whilestmt,
  Forstmt,Ifstmt,IFComp,ExpTail,Dowhilestmt;

/* Definição das precedências */
precedence right ASSIGN_OP;
precedence left DGT, DLT, TGT, TLT;
precedence left OR;
precedence left AND;
precedence left EQ, NE;
precedence left LT, LE, GT, GE;
precedence left PLUS, MINUS;
precedence left MULT, DIV, MOD;


start with Startprogram;


/* Produções da gramática */
Startprogram ::= PROGRAM IDENTIFIER Body;

Body ::= Dlist BEGIN Stmtlist END;

Dlist ::= DECLARE DeclList | ;

DeclList ::= Decl Semicolondecl;

Semicolondecl ::= SEMICOLON Decl Semicolondecl | SEMICOLON;

Decl ::= Type IdentList;

IdentList ::= IDENTIFIER Commoindetifier;

Commoindetifier ::= COMMA IDENTIFIER Commoindetifier | ;

Type ::= INTEGER | DECIMAL;
```

```
Stmtlist ::= Stmt SemiColonstmt;

SemiColonstmt ::= SEMICOLON Stmt SemiColonstmt | ;

Stmt ::= Assignstmt
        | Readstmt
        | Writestmt
        | Whilestmt
        |Dowhilestmt
        | Ifstmt
        | Forstmt;

Assignstmt ::= IDENTIFIER ASSIGN_OP Simpleexpr;



Expression ::= Simpleexpr ExpTail ;



ExpTail ::= Relop Simpleexpr
         | Shiftop Simpleexpr
         | ;



Simpleexpr ::= Term
            | Simpleexpr Addop Term
            | RIGHT_PAREN Expression LEFT_PAREN QUESTION_MARK Simpleexpr COLON
Simpleexpr;

Writable ::= Simpleexpr | Literal;

Condition ::= Expression;


Ifstmt ::= IF Condition THEN  Stmtlist  IFComp;

IFComp ::= END
         | ELSE  Stmtlist END;


Term ::= FactoA | Term Mulop FactoA;

FactoA ::= Factor | NOT Factor | MINUS Factor;

Readstmt ::= READ RIGHT_PAREN IDENTIFIER LEFT_PAREN;
```

```
Writestmt ::= WRITE RIGHT_PAREN Writable LEFT_PAREN ;



Factor ::= IDENTIFIER | INTEGERCONSTANT | DECIMALCONSTANT | RIGHT_PAREN
Expression LEFT_PAREN  ;

/* Loops */

Forstmt ::= FOR Assignstmt TO Condition  DO Stmtlist  END ;
Whilestmt ::= Stmtsuffix DO Stmtlist  END;
Stmtsuffix ::=  WHILE Condition ;
Dowhilestmt ::=  DO Stmtlist   Stmtsuffix;




Relop ::= EQ | GT | GE | LT | LE | NE;

Addop ::= PLUS | MINUS | OR;

Mulop ::= MULT | DIV | MOD | AND;

Shiftop ::= TLT
        | TGT
        | DLT
        | DGT;


Literal ::= TEXTO;
```

C) Resultados

Para verificar os resultados da linguagem foi inserido algumas explicações semânticas nas regras produzidas na gramática.

parser.java

```
import java_cup.runtime.Symbol;

parser code {:
   // conectar esse parser ao scanner!
```

```
    Yylex s;
    TabelaSimbolos Ts ;
    Parser(Yylex s , TabelaSimbolos Ts ){
      this.s=s;
      this.Ts = Ts;
    }
:}

/* conectar esse parser ao scanner! */
init with {:  :};
scan with {: return s.next_token(); :};


/* Define o conjunto de terminais */
terminal
  PROGRAM, DECLARE, BEGIN, INTEGER, DECIMAL, IF, THEN, ELSE, FOR, END, DO,
TO,
  WHILE, READ, WRITE, OR, AND, MOD, NOT, IDENTIFIER, INTEGERCONSTANT,
DECIMALCONSTANT,
  ASSIGN_OP, COMMA, SEMICOLON, RIGHT_PAREN, LEFT_PAREN,
  EQ, MINUS, PLUS, LT, LE, NE, GT, GE, MULT, DIV, DGT, DLT, TGT, TLT,
QUESTION_MARK, COLON, TEXTO;

/* Define o conjunto de não-terminais */
non terminal
  Startprogram, Body, DeclList, Decl, IdentList, Type, Dlist, Commoindetifier, Semicolondecl,
  Stmtlist, Stmt, SemiColonstmt, Assignstmt, Simpleexpr, Term,
  FactoA, Factor, Expression, Relop, Addop, Mulop,
  Shiftop, Literal, Condition, Writable, Readstmt,
  Writestmt, Stmtsuffix, Whilestmt,
  Forstmt,Ifstmt,IFComp,ExpTail,Dowhilestmt;

/* Definição das precedências */
precedence right ASSIGN_OP;
precedence left DGT, DLT, TGT, TLT;
precedence left OR;
precedence left AND;
precedence left EQ, NE;
precedence left LT, LE, GT, GE;
precedence left PLUS, MINUS;
precedence left MULT, DIV, MOD;


start with Startprogram;


/* Produções da gramática */
```

```
Startprogram ::= PROGRAM IDENTIFIER Body
            {: System.out.println("OK"); :};

Body ::= Dlist BEGIN Stmtlist END  {: System.out.println("OK"); :};

Dlist ::= DECLARE DeclList |   {: System.out.println("OK"); :};

DeclList ::= Decl Semicolondecl  {: System.out.println("OK"); :};

Semicolondecl ::= SEMICOLON Decl Semicolondecl | SEMICOLON  {:
System.out.println("OK"); :};

Decl ::= Type IdentList  {: System.out.println("OK"); :};

IdentList ::= IDENTIFIER Commoindetifier  {: System.out.println("OK"); :};

Commoindetifier ::= COMMA IDENTIFIER Commoindetifier |   {: System.out.println("OK");
:};

Type ::= INTEGER | DECIMAL  {: System.out.println("OK"); :};

Stmtlist ::= Stmt SemiColonstmt  {: System.out.println("OK"); :};

SemiColonstmt ::= SEMICOLON Stmt SemiColonstmt |   {: System.out.println("OK"); :};

Stmt ::= Assignstmt
       | Readstmt
       | Writestmt
       | Whilestmt
       |Dowhilestmt
       | Ifstmt
       | Forstmt  {: System.out.println("OK"); :};

Assignstmt ::= IDENTIFIER ASSIGN_OP Simpleexpr  {: System.out.println("OK"); :};



Expression ::= Simpleexpr ExpTail   {: System.out.println("OK"); :};



ExpTail ::= Relop Simpleexpr
       | Shiftop Simpleexpr
       |   {: System.out.println("OK"); :};
```

```
Simpleexpr ::= Term
            | Simpleexpr Addop Term
            | RIGHT_PAREN Expression LEFT_PAREN QUESTION_MARK Simpleexpr COLON
Simpleexpr  {: System.out.println("OK"); :};

Writable ::= Simpleexpr | Literal  {: System.out.println("OK"); :};

Condition ::= Expression  {: System.out.println("OK"); :};


Ifstmt ::= IF Condition THEN  Stmtlist  IFComp  {: System.out.println("OK"); :};

IFComp ::= END
        | ELSE  Stmtlist END  {: System.out.println("OK"); :};


Term ::= FactoA | Term Mulop FactoA  {: System.out.println("OK"); :};

FactoA ::= Factor | NOT Factor | MINUS Factor  {: System.out.println("OK"); :};

Readstmt ::= READ RIGHT_PAREN IDENTIFIER LEFT_PAREN  {:
System.out.println("OK"); :};

Writestmt ::= WRITE RIGHT_PAREN Writable LEFT_PAREN   {: System.out.println("OK"); :}

Factor ::= IDENTIFIER | INTEGERCONSTANT | DECIMALCONSTANT | RIGHT_PAREN
Expression LEFT_PAREN    {: System.out.println("OK"); :};

/* Loops */

Forstmt ::= FOR Assignstmt TO Condition  DO Stmtlist  END   {: System.out.println("OK"); :};
Whilestmt ::= Stmtsuffix DO Stmtlist  END  {: System.out.println("OK"); :};
Stmtsuffix ::=  WHILE Condition   {: System.out.println("OK"); :};
Dowhilestmt ::=  DO Stmtlist   Stmtsuffix  {: System.out.println("OK"); :};




Relop ::= EQ | GT | GE | LT | LE | NE  {: System.out.println("OK"); :};

Addop ::= PLUS | MINUS | OR  {: System.out.println("OK"); :};

Mulop ::= MULT | DIV | MOD | AND  {: System.out.println("OK"); :};

Shiftop ::= TLT
        | TGT
        | DLT
```

```
        | DGT  {: System.out.println("OK"); :};

Literal ::= TEXTO  {: System.out.println("OK"); :};
```

Para executar esse código e gerar o analisador léxico é necessário executar essa comando:

```
java -jar java-cup-11b.jar -parser Parser -symbols Sym parser.cup
```

Conclusão

Em resumo, durante a implementação do compilador para a linguagem MINI, foram desenvolvidos os componentes léxico e sintático com sucesso. O analisador léxico, construído com o auxílio do JFlex, o analisador sintático, implementado utilizando a ferramenta Java Cup. Entretanto, não foi possível realizar a implementação dos analisadores semânticos e a fase de geração de código.