FUNDAÇÃO UNIVERSIDADE FEDERAL DO VALE DO SÃO FRANCISCO

COLEGIADO DE ENGENHARIA DE COMPUTAÇÃO - CECOMP

DISCIPLINA: COMPILADORES

DOCENTE: MARCUS RAMOS

COMPILADOR PARA LINGUAGEM MINI-PASCAL

RELATÓRIO FINAL

DISCENTE:

Gustavo Marques de Souza Santos

Wesley Souza

JUAZEIRO-BA

Março/2019

Sumário

Introdução **3**

Manual de utilização **5**

Analisador sintático **8**

Tokens 8

Gramática 10

Scanning 13

Parsing 17

Árvore sintática abstrata 22

Detecção de erros 29

Analisador de Contexto **30**

Identificação 30

Visitor 33

Checagem de tipo 39

Impressão da Árvore **42**

Gerador de código **45**

Templates de código 45

Tradução 46

Avaliando expressões 49

Alocação de memória 51

Salvando código 53

Observação: 53

Conclusão **54**

Anexos **55**

Anexo A **55**

Sem erro(s): 55

Com erro(s): 63

Anexo B **64**

Sem erro(s): 64

Com erro(s): 65

Anexo C **68**

Sem erro(s): 68

Com erro(s): 69

Anexo D **71**

Sem erro(s): 71

Com erro(s): 72

Introdução

Um compilador traduz uma linguagem de alto nível, nesse trabalho a linguagem será o Mini Pascal, em linguagem de mais baixo nível. Normalmente (não é regra) um compilador gera código de máquina.

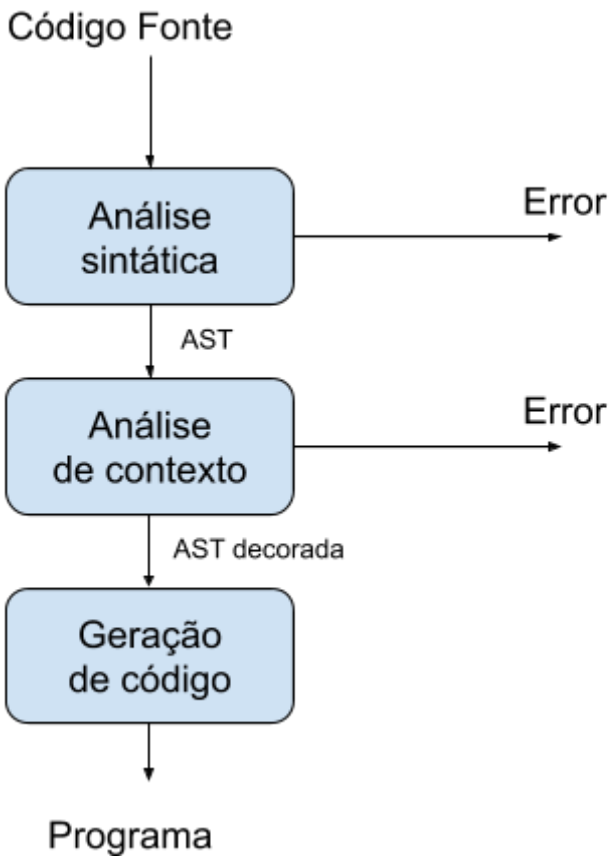
Durante todo o documento, trataremos como código fonte o arquivo contendo o código escrito na linguagem MiniPascal. E definiremos diversos aspectos da linguagem, tais como :

1. A sintaxe do código fonte:
    - A sintaxe define quais os símbolos são aceitos e como esses símbolos são sequenciados.
2. As restrições de contexto:
    - Verifica se as regras de escopo e de tipo estão de acordo, ou seja, verifica as declarações de variáveis e seus valores além de avaliar o tipo das expressões.
3. A semântica do programa:
    - Define o comportamento do programa, quando executado em uma máquina.

Durante a compilação, o código fonte passa por diversas transformações, antes de que o código seja gerado. Essas transformações também são chamadas de fases de compilação, este projeto conta com um compilador de 3-fases. Cada fase está diretamente ligada aos aspectos da linguagem. As fases que esse compilador implementa, ou pelo menos deveria implementar:

1. Análise sintática;
2. Análise de contexto;
3. Geração de código.

Este compilador funciona então de acordo com o diagrama abaixo:



De maneira resumida, o compilador conta com um módulo, que coordena, acoplados a outros três módulos especializados (fases do compilador), são eles: analisador sintático, analisador de contexto e gerador de código.

A sequência de execução fica então, a classe Main chama o analisador sintático, que lê o código fonte e constrói uma AST completa. Ao final do analisador sintático e caso nenhum erro ocorra, a classe Main chama o analisador de contexto, que verifica as restrições de contexto na AST e decora a árvore com os tipos. Por fim a classe Main chama o gerador de código.

Manual de utilização

O projeto está organizado dessa forma:

Pasta	Conteúdo
Compiler_V4	Todo conteúdo do compilador, o que incluí código até os binários Java.
Programas	Todos os programas testes utilizados e analisados.

O texto a seguir considera o local atual como sendo a pasta Compiler_V4. No terminal linux basta executar:

Ordem	Comando
1	$ cd Compiler_V4

De maneira a manter uma boa organização do projeto, separamos os códigos dos binários, de forma que o projeto está estruturado dessa forma:

Pasta	Conteúdo
src/	A pasta “src” contém todos os códigos do compilador;
bin/	A pasta “bin” contém todos os binários java, obviamente, depois de compilados.

Sendo que o analisador sintático consiste de um pacote (*package*) minipascal.syntatic_analyser que contém as classes Parser, scanner e Token.

Os arquivos da AST estão todos dentro da pasta src/AST, sendo que a mesma forma o pacote minipascal.ast.

Para compilar o projeto executamos, em ordem, os seguintes comandos:

Ordem	Comando
1	$ mkdir bin
2	$ javac -d bin src/Encoder.java src/Instruction.java src/Runtime/*.java src/AST/*.java src/Token.java src/scanner.java src/Parser.java src/Print.java src/Checker.java src/identificationTable.java
3	$ javac -d bin -cp "bin/" src/Main.java

Outra forma, mais simples, de compilar o projeto é por meio do arquivo de *makefile*. Para compilar basta:

Ordem	Comando
1	$ make

Embora seja possível compilar o projeto separadamente, não será possível executar uma vez que a Main depende de todas as classes. Depois de compilado, é possível executar por partes.

O compilador conta com uma interface por meio da linha de comando. Para ter acesso as opções disponíveis basta executar o compilador com a opção “-h” conforme segue:

```
$ java -cp "bin/" Main -h
```

ou

Ordem	Comando
1	$ cd bin
2	java Main -h

As opções disponíveis são:

Opção	Descrição
-h	Menu de ajuda.
-i	Endereço do arquivo com o código fonte.
-s	Executar até o Scanner e gerar uma saída mais completa do Scanner.
-p	Executar até o parser e gerar uma saída mais completa do Parser.
-c	Executar até o Checker (análise de contexto) e gerar uma saída mais completa do Checker.
-o	Imprime a árvore.
-e	Executar até o Encoder (gerador de código) e gerar uma saída mais completa do Encoder.
-f	Endereço para salvar o código gerado

- Todos os comandos, exceto o “-h” devem conter o endereço do arquivo fonte;
- Não existe exigência de extensão, basta ser arquivo de texto;
    - Contudo, todos os programas utilizados como testes utilizam a extensão “.mp”

- Todos os comandos foram executados em uma máquina com sistema operacional de base Linux, os comandos devem ser adaptados para a realidade de outros sistemas operacionais

Analisador sintático

O propósito principal do analisador sintático é realizar a leitura do código fonte com intuito de verificar sua estrutura. Esse trabalho utiliza um método bastante popular e conhecido como recursivo descendente.

O analisador é subdividido em fases, contendo:

1. Analisador sintático, responsável por ler o arquivo contendo o código fonte e transformar esse código em uma sequência de Tokens.
2. Parsing, responsável por analisar se a sequência de tokens está corretamente estruturada.
3. Estrutura de dados utilizadas para representação, uma vez que o compilador é de mais de um passo é necessária a construção de um estrutura para representar o código fonte por meio dos Tokens.

Tokens

Primeiramente, um *token* é um símbolo atômico (indivisível e único). Esses tokens servem como interface entre as classes Scanner e Parser.

Todos os tokens podem ser descritos pelo seu tipo, por sua escrita e por sua posição no arquivo fonte. O compilador utiliza uma classe, chamada Token, para definir os diversos tokens dos códigos.

```
public class Token
{
	// instance variables - replace the example below with your own
	…

	/**
	* Constructor for objects of class Token
	*/
	public Token(byte kind, String spelling, int col, int line)
	{
	// initialise instance variables
	this.kind          = kind;
	this.spelling    = spelling;
	this.col           = col;
	this.line          = line;

	if(kind == IDENTIFIER) {
```

```
            for(int k = PROGRAM; k<= FALSE; k++){
                  if(spelling.equals(spellings[k])) {
                  this.kind =(byte) k; break;
                  }
            }
      }
      }

      public void print() {
      System.out.println("[TOKEN]:\t"+spellings[kind].toUpperCase());
      System.out.println("\tKind:\t"+kind);
      System.out.println("\tSpell:\t"+spelling);
      }

      // Constants denoting different kings of token
      public final static byte
      IDENTIFIER  = 0,
      INTLITERAL  = 1,

      ...

      L_SQUARE    = 31,
      R_SQUARE    = 32,
      L_PAREN     = 33,
      R_PAREN     = 34;
      //
      public final static String[] spellings = {
      "<IDENTIFIER>",
      "<INTLITERAL>",
      "<OPERADOR>",

      ...

      "]",
      "[",
      ")",
      "("
      };
}
```

Gramática

Antes de realizar qualquer tratativa do Parser, a gramática da linguagem precisa ser analisada. A análise da linguagem consiste em verificar se será possível gerar uma analisador descendente para a mesma. Esse ponto é de extrema importância na continuação do projeto, deve-se adequar a linguagem realizando transformações mas sem perder o poder de gerar as mesmas cadeias.

As transformações utilizadas serão:

1. Fatoração à esquerda:
    a. Regra: **XY | XZ = X(Y|Z)**

Antes	Depois
<op-rel> ::= | > | <= | >= | = | <>	<op-rel> ::= (**<**(**=**|**>**|<vazio>) | **>**(**=**|<vazio>) | **=**)

2. Eliminação de fatoração à esquerda:
    a. Regra: **N ::= X | N Y** => **N ::= X (Y)***

Antes	Depois
<id> ::= | <id> <letra> | <id> <dígito>	<id> ::= <letra>(<letra>|<dígito>)*

3. Substituição de Não Terminais:
    a. Regra: **N ::= X** => **X (No lugar de N)**

Antes	Depois
<bool-lit> ::= | false <literal> ::= | <int-lit> | <float-lit>	<literal> ::= **true** | **false** | <int-lit> (**.**(<int-lit> | <vazio>) | <vazio>) | **.**<int-lit>

A linguagem manipulada, utilizando as transformações anteriores, ficou então:

<letra> ::=
a|b|c|d|e|f|g|h|i|j|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|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
<digito> ::= 0|1|2|3|4|5|6|7|8|9
<vazio> ::= ε
<id> ::= <letra>(<letra>|<dígito>)*
<programa> ::= **program** <id> **;** <corpo> **.**
<corpo> ::= (<declaração> **;**)* **begin** (<comando> **;**)* **end**
<declaração> ::= **var** <id> (**,** <id>)* **:** <tipo>
<tipo> ::=
array [<literal> **..** <literal>] **of** <tipo>
| **integer**
| **real**
| **boolean**

<literal> ::=
true
| **false**
| <int-lit> (**.**(<int-lit> | <vazio>) | <vazio>)
| **.**<int-lit>

<int-lit> ::= <digito>(<digito>)*
<comando> ::=
<variável> **:=** <expressão>
| **if** <expressão> **then** <comando> (**else** <comando> | <vazio>)
| **while** <expressão> **do** <comando>
| **begin** (<comando> **;**)* **end**

<variável> ::= <id> (**[** <expressão> **]**)*
<expressão> ::= <expressão-simples> (<op-rel> <expressão-simples> | <vazio>)
<expressão-simples> ::= <termo> ((<op-ad><termo>)* | <vazio)
<op-ad> ::= (**+ | -** | **or**)
<op-mul> ::= (***** | **/** | **and**)
<op-rel> ::= (**<**(**=**|**>**|<vazio>) | **>**(**=**|<vazio>) | **=**)
<termo> ::= <fator>((<op-mul> <fator>)* | <vazio>)
<fator> ::=
<variável>
| <literal>
| **(** <expressão> **)**

Onde os Starter sets são:

starter[<letra>] ::= {
a,b,c,d,e,f,g,h,i,j,l,m,n,op,q,r,s,t,u,v,w,x,y,z,A,B,C,D,E,F,G,H,I,J,L,M,N,O,P,Q,R,S,T,U,V,W,
X,Y,Z }
starter[<digito>] ::= { 0,1,2,3,4,5,6,7,8,9 }
starter[<vazio>] ::= { ε }
starter[<id>] ::= starter[<letra>]
starter[<programa>] ::= { **p** }
starter[<corpo>] ::= starter[<declaração>]
starter[<declaração>] ::= { **v** }
starter[<tipo>] ::= { **a, i, r , b** }
starter[<literal>] ::= { **t, f, .** } U starter[<dígito>]
starter[<int-lit>] ::= starter[<dígito>]
starter[<comando>] ::= starter[<variável>] U { **i, w, b** }
starter[<variável>] ::= starter[<id>]
starter[<expressão>] ::= starter [<expressão-simples>]
starter[<expressão-simples>] ::= starter [<termo>]
starter[<op-ad>] ::= {**+ , -** , **o** }
starter[<op-mul>] ::= { ***** , */* , **a** }
starter[<op-rel>] ::= { **<** , **>**, **=** }
starter[<termo>] ::= starter[<fator>]
starter[<fator>] ::= starter [<variável>] U starter [<literal>] U { **(** }

Esse conjunto de starter ainda não é LL(1), conforme demonstrado abaixo:

starter[<fator>] ::= starter [<variável>] U starter [<literal>] U { **(** }
starter[<fator>] ::= starter[<id>] U { **t, f, .** } U starter[<dígito>] U { **(** }
starter[<fator>] ::= starter[<letra>] U { **t, f, .** } U { 0,1,2,3,4,5,6,7,8,9 } U { **(** }
starter[<fator>] ::=
a,b,c,d,e,**f**,g,h,i,j,l,m,n,op,q,r,s,**t**,u,v,w,x,y,z,A,B,C,D,E,F,G,H,I,J,L,M,N,O,P,Q,R,S,T,U,V,W,X,
Y,Z } U { **t, f, .** } U { 0,1,2,3,4,5,6,7,8,9 } U { **(** }
starter[<fator>] ::= {t, f } U { t, f }

Para solucionar esse problema, basta verificar que as palavras chaves (<*keywords*>), pré-definidos (<*predefined*>) são tipos de **identificadores**. Dessa forma a classe Scanner irá classificar os Tokens de acordo com as regras abaixo:

Tokens ::= <id> | <literal> | <special>
<id> ::= <letra>(<letra>|<dígito>)*
<literal> ::=
<digito>(<digito>)* (**.**(<digito>(<digito>)* | <vazio>) | <vazio>)
| **.** <digito>(<digito>)*
<special> ::= **;** | **:** | **[** | **]** | **(** | **)** | **+** | **-** | ***** | **/** | **<** | **>** | **<=** | **>=** | **=** | **<>** | **..** | **,** | **:=**

<keywords> ::= **program** | **begin** | **end** | **var** | **array** | **integer** | **real** | **boolean** | **if** | **else** | **then** | **while** | **do** | **or** | **and** | **of**

<predefined> ::= **true** | **false**

<comments>::= **!**(<gráfico>*<fim de linha>|!<gráfico>*!)

Assim, o conjunto de **starter[<Tokens>] é LL(1)**.

Scanning

A leitura do arquivo e separação dos Tokens é feita na classe Scanner (src/scanner.java). Com base nessa classe, um conjunto de programas, com e sem erros, foram analisados e suas respectivas análises estão disponíveis no Anexo A.

A implementação das regras fica então assim:

<id> ::= <letra>(<letra>|<dígito>)*

```
switch(currentChar){
            case 'q':case 'w':case 'e':case 'r':case 't':case 'y':
            case 'u':case 'i':case 'o':case 'p':case 'a':case 's':
            case 'd':case 'f':case 'g':case 'h':case 'j':case 'k':
            case 'l':case 'z':case 'x':case 'c':case 'ç':case 'v':
            case 'b':case 'n':case 'm':
            case 'Q':case 'W':case 'E':case 'R':case 'T':case 'Y':
            case 'U':case 'I':case 'O':case 'P':case 'A':case 'S':
            case 'D':case 'F':case 'G':case 'H':case 'J':case 'K':
            case 'L':case 'Z':case 'X':case 'C':case 'Ç':case 'V':
            case 'B':case 'N':case 'M':
                  takeIt();
                  while(isLetter(currentChar)
                        || isDigit(currentChar))
                        takeIt();
                  return Token.IDENTIFIER;
```

<literal> ::=
<digito>(<digito>)* (.(<digito>(<digito>)* | <vazio>) | <vazio>)
| . <digito>(<digito>)*

```
case '0':case '1':case '2':case '3':case '4':case '5':
            case '6':case '7':case '8':case '9':
                  takeIt();
                  while(isDigit(currentChar))
```

```
                takeIt();

                if(currentChar == '.')
                {
                takeIt();
                while(isDigit(currentChar))
                      takeIt();
                return Token.FLOATLIT;
                }

                return Token.INTLITERAL;
```

```
case '.':
                takeIt();
                if(isDigit(currentChar))
                {
                while(isDigit(currentChar))
                      takeIt();
                return Token.FLOATLIT;
                }
                return Token.DOT;
```

<special> ::= ; | : | [|] | (|) | + | - | * | / | < | > | <= | >= | = | <> | .. | , | :=

```
case ',':
                takeIt();
                return Token.COMMA;
            case ':':
                takeIt();
                if(currentChar == '=')
                {
                takeIt();
                return Token.IS;

                }
                return Token.COLON;
            case '.':
                takeIt();
                if(isDigit(currentChar))
                {
                while(isDigit(currentChar))
                      takeIt();
                return Token.FLOATLIT;
                }
```

```
            return Token.DOT;

        case ';':
            takeIt();
            return Token.SEMICOLON;
        case '+':case '-':
            takeIt();
            return Token.OP_AD;
        case '*':case '/':
            takeIt();
            return Token.OP_MUL;
        case '<':
            takeIt();
            if( currentChar == '=' ||
            currentChar == '>')
            takeIt();
            return Token.OP_REL;
        case '>':
            takeIt();
            if( currentChar == '=')
            takeIt();
            return Token.OP_REL;
        case '=':
            takeIt();
            return Token.OP_REL;
        case '[':
            takeIt();
            return Token.R_SQUARE;
        case ']':
            takeIt();
            return Token.L_SQUARE;
        case '(':
            takeIt();
            return Token.R_PAREN;
        case ')':
            takeIt();
            return Token.L_PAREN;
        case '\0':
            //takeIt();
            return Token.EOT;
        default:
            errorHandler(true);
            return -1;
```

Os comentários, também no arquivo src/scanner.java, são implementados dessas forma:

```
while((currentChar == '!'
                  || currentChar == ' '
                  || currentChar == '\t'
                  || currentChar == '\n') && file_Stream.available() >
0) {
            scanSeparator();
}
```

```
case '!':
                  takeIt();

                  multiline = false;
                  if(currentChar == '!')
                        multiline = true;

                  while( isSymbol(currentChar)
                              || multiline)
                        {

                              takeIt();
                              if(multiline == true && currentChar == '!')
                              {
                              multiline = false;
                              takeIt();
                              }
                        }

                  if(multiline)
                  {
                        errorHandler(true, '!');
                  }
                  take('\n');
```

Parsing

Para análise da estrutura dos tokens, o projeto conta com um *parser* que implementa um analisador utilizando a estratégia Top-Down por meio de um recursivo descendente.

Novos programas foram desenvolvidos, uma vez que o *parser* analisa a estrutura das frases nenhum programa utilizado na etapa do *scanning* passou (observe que falta program em todos). O anexo B conta com análise dos programas utilizados para verificação do funcionamento.

O *parser* está implementado no arquivo src/Parser.java. A seguir os detalhes da implementação serão discutidos.

Com base na regra

<programa> ::= **program** <id> **;** <corpo> **.**

o parser deve reconhecer primeiro o identificador **program**.

O construtor do Parser instância um scanner e realiza a leitura do primeiro Token. Após leitura do primeiro Token, a análise baseado na regra <programa> será iniciado por meio da função parseProgram();

```
/**
* Constructor for objects of class Parser
*/
public Parser(File file_object) throws Exception
{
 P = null;
current_scanner = new scanner(file_object);

currentToken = current_scanner.scan();
//currentToken.print();
P = parseProgram();

}
```

A função parseProgram então implementa a regra:

```
// <programa> ::= program <id> ; <corpo> .
private Program parseProgram () throws Exception
{
 Id identifier;
 Corpo Cr;

Program_line = currentToken.line;
accept(Token.PROGRAM);

identifier = parserIdentifier();

accept(Token.SEMICOLON);

Cr = parseCorpo();

accept(Token.DOT);
accept(Token.EOT);
```

```
      Program P = new Program(identifier, Cr);
      return P;
      }
```

A função `accept(Token.PROGRAM);` verifica se o Token lido é igual ao Token esperado, nesse caso espera-se que o token seja o Token.PROGRAM definido na classe Token.

Caso o token lido seja diferente do token esperado, o procedimento para tratamento de erros é disparado. Esse procedimento informa ao usuário sobre o erro, além de passar informações que auxiliarão na solução do problema.

```
private void acceptIt() throws Exception
{
      currentToken = current_scanner.scan();
      //currentToken.print();
}

private void accept (byte expectedKind) throws Exception
{
      if(currentToken.kind == expectedKind)
            acceptIt();
    else
   {
            handle_error(expectedKind);
    }
}
```

Essa função, *parseProgram,* chama outras funções para análise da regra <id> e da regra <corpo>. Ambas implementadas por meio das funções parserIdentifier e parseCorpo, respectivamente.

```
// <corpo> ::= (<declaração> ;)*  begin  ( <comando> ; )* end
      private Corpo parseCorpo () throws Exception
      {
       Declaracao D = null;
       Comando C = null;

      // (<declaração> ;)*
      if(currentToken.kind == Token.VAR)
      {
             D = parseDeclaracao();
            accept(Token.SEMICOLON);
      }
      while(currentToken.kind == Token.VAR)
```

```
	{
		Declaracao D1 = parseDeclaracao();
		accept(Token.SEMICOLON);
		D = new Declaracao_seq(D,D1);
	}
	accept(Token.BEGIN);
	if(currentToken.kind != Token.END)
	{
		 C = parseComando();
		 accept(Token.SEMICOLON);
	}
	while(currentToken.kind != Token.END)
	{
		Comando C1 = parseComando();
		accept(Token.SEMICOLON);
		C = new Comando_Seq(C,C1,currentToken.line);
	}
	accept(Token.END);
	Corpo Cr = new Corpo(D,C);
	return Cr;
	}
```

```
//<id> ::= <letra>(<letra>|<dígito>)*
	private Id parserIdentifier() throws Exception
	{
	 Id_simples idAST = null;
	if(currentToken.kind == Token.IDENTIFIER)
	{
		 idAST = new Id_simples(currentToken.spelling);
		acceptIt();
	}
	else
	{
		handle_error(Token.IDENTIFIER);
	}
	return idAST;
	}
```

Para não tornar esse documento tedioso e desnecessariamente grande, a seguir segue a lista relacionando o método da classe Parser com a respectiva regra que ela implementa.

Regra	Método
<programa> ::= program <id> ; <corpo> .	`private Program parseProgram () throws Exception`

<corpo> ::= (<declaração> ;)* begin (<comando> ;)* end	`private Corpo parseCorpo () throws Exception`

<comando> ::=
  <variável> := <expressão>
  | if <expressão> then <comando> (else <comando> | <vazio>)
  | while <expressão> do <comando>
  | begin (<comando> ;)* end
	`private Comando parseComando () throws Exception`

// <expressão> ::= <expressão-simples> (<op-rel> <expressão-simples> | <vazio>)	`private Expressao parserExpressao () throws Exception`
//<expressão-simples> ::= <termo> ((<op-ad><termo>)* | <vazio)	`private Expressao_simples parserExpressaoSimples () throws Exception`
//<termo> ::= <fator>((<op-mul> <fator>)* | <vazio>)	`private Termo parserTermo () throws Exception`

<fator> ::=
  <variável>
  | <literal>
  | (<expressão>)
	`private Fator parserFator () throws Exception`

// <variável> ::= <id> ([<expressão>])*	`private Variavel parserVariavel () throws Exception`
// <declaração> ::= var <id> (, <id>)* : <tipo>	`private Declaracao parseDeclaracao () throws Exception`

<tipo> ::=
  array [<literal> .. <literal>] of <tipo>
  | integer
  | real
  | boolean
	`private Tipo parserTipo() throws Exception`

<literal> ::=
  true
  | false
  | <int-lit> (.(<int-lit> | <vazio>) | <vazio>)
  | .<int-lit>
	`private String parseLiteral() throws Exception`

<id> ::= <letra>(<letra>|<dígito>)*	`private Id parserIdentifier() throws Exception`

Os pontos mais importantes da implementação serão apresentados a seguir. Todos os detalhes da implementação estão disponíveis no arquivo src/Parser.java.

A regra

<corpo> ::= (<declaração> **;**)* **begin** (<comando> **;**)* **end**

pode conter zero ou mais <declarações>, para implementar essa regra utilizamos:

```
// (<declaração> ;)*
      if(currentToken.kind == Token.VAR)
      {
             D = parseDeclaracao();
            accept(Token.SEMICOLON);
      }
      while(currentToken.kind == Token.VAR)
      {
            Declaracao D1 = parseDeclaracao();
            accept(Token.SEMICOLON);
            D = new Declaracao_seq(D,D1);
      }
```

A regra

<fator> ::=
<variável>
| <literal>
| **(** <expressão> **)**

foi implementada por meio do comando switch, conforme descrito abaixo:

```
private Fator parserFator () throws Exception
      {
       Fator F = null;
       byte kind;

      switch(currentToken.kind)
      {
            case Token.IDENTIFIER:
                  Variavel VAR = parserVariavel();
                  F = new Fator_VAR(VAR);
            break;
            case Token.INTLITERAL: case Token.FLOATLIT: case Token.TRUE:
            case Token.FALSE:
                   kind = currentToken.kind;
                   String L1 = parseLiteral();
                   F = new Fator_LIT(L1, kind);
            break;
            case Token.R_PAREN:
```

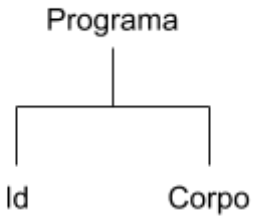
```
                accept(Token.R_PAREN);
                Expressao E = parserExpressao();
                 F = new Fator_EXP(E);
                accept(Token.L_PAREN);
          break;
          default:
                handle_error(1002);
          break;
    }

     return F;
    }
```

Árvore sintática abstrata

Conforme definido no início do projeto, esse trabalho trata de um compilador com vários estágios. Em compiladores que utilizam essa estratégia deve-se existir uma estrutura de dados que represente o programa fonte lido. Para esse trabalho utilizamos como estrutura de dados uma árvore sintática abstrata (AST - em inglês: *abstract syntax tree*).

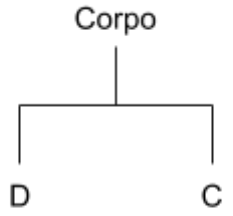
A seguir é apresentado, de forma macro, como é definido a árvore.

Program (programa):

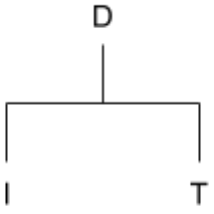


ID (Id):

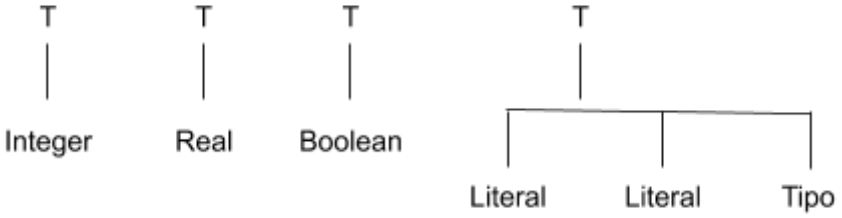
Corpo (Cr):



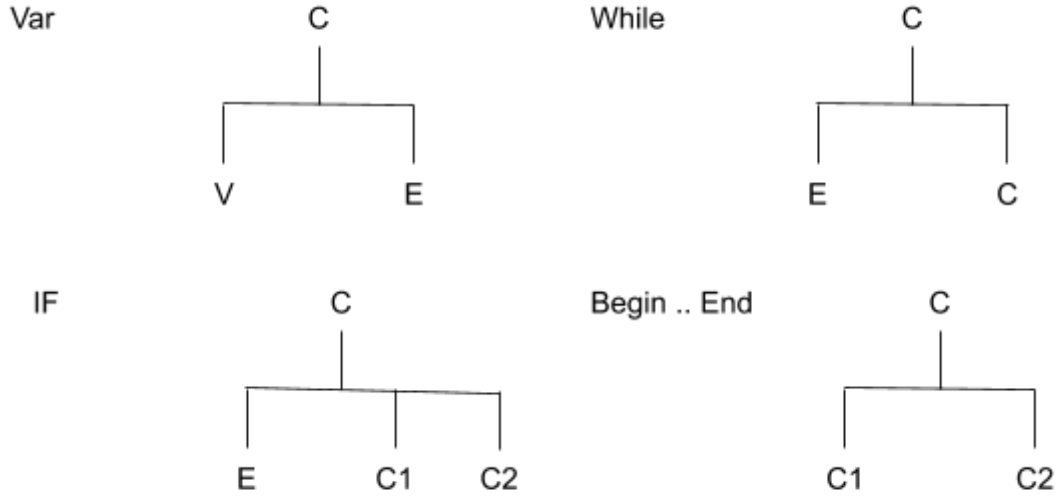
Declaração (D):



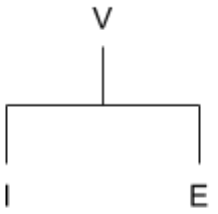
Tipo (T):



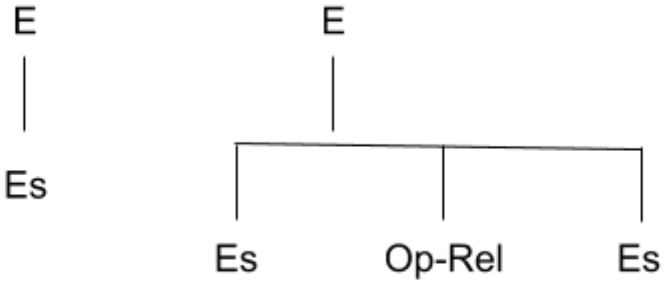
Comando (C):



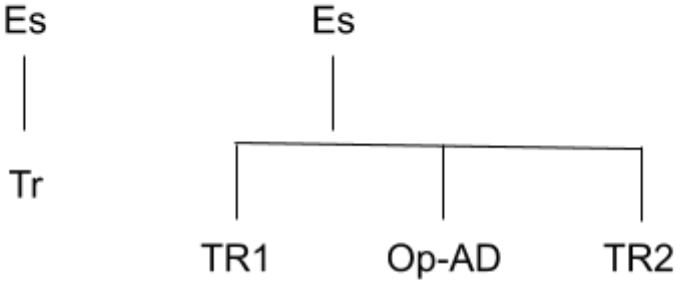
Variável (V):



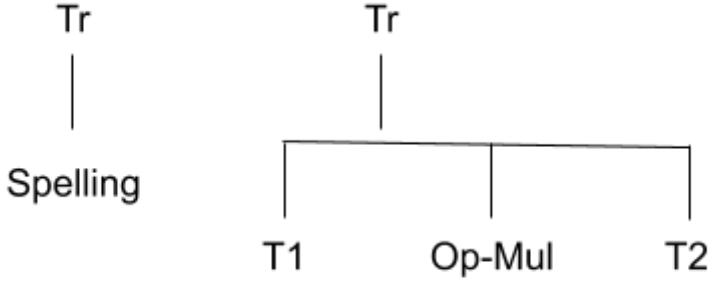
Expressão (E):



Expressão simples (Es):



Termo (Tr):



A árvore é definida inicialmente por meio de uma classe abstrata AST, representado no quadro abaixo. Todos os nós da árvore são objetos da classe AST.

```
package minipascal.ast;
public abstract class AST
{
      public abstract Object visit (Visitor v, Object arg);
};
```

Para cada não-terminal uma classe é definida, como é o caso do não-terminal “Corpo”. Caso o não-terminal contenha diversas formas, caso do não-terminal “Comando”, uma classe abstrata é definida e cada caso implementa uma classe concreta.

A implementação do não terminal “Corpo” fica então assim:

```
package minipascal.ast;
//========================================================================
public class Corpo extends AST {
    public Declaracao D;
    public Comando C;

    public Corpo(Declaracao D, Comando C)
    {
        this.D = D;
        this.C = C;
    }

    public Object visit(Visitor v, Object arg) {
      return v.visitCorpo(this, arg);
    }
};
```

A ïmplementação do não-terminal Comando fica assim:

```
package minipascal.ast;
//========================================================================
public abstract class Comando extends AST {};
```

Comando While:

```
package minipascal.ast;
//========================================================================
public class Comando_WHILE extends Comando {
    public Expressao E1;
    public Comando C1;
    public int line;

    public Comando_WHILE(Expressao E1, Comando C1, int line)
    {
```

```
        this.E1 = E1;
        this.C1 = C1;
        this.line = line;
    }

    public Object visit(Visitor v, Object arg) {
      return v.visitWhileCommand(this, arg);
    }
};
```

Comando VAR:

```
package minipascal.ast;

//=====================================================================
public class Comando_VAR extends Comando {
    public Variavel V1;
    public Expressao E1;
    public int line;

    public Comando_VAR(Variavel V1, Expressao E1, int line)
    {
        this.V1 = V1;
        this.E1 = E1;
        this.line = line;
    }

    public Object visit(Visitor v, Object arg) {
      return v.visitVarCommand(this, arg);
    }
};
```

Comando Sequencial:

```
package minipascal.ast;
//
=====================================================================
public class Comando_Seq extends Comando {
    public Comando C1, C2;
    public int line;

    public Comando_Seq(Comando C1, Comando C2, int line)
    {
        this.C1 = C1;
        this.C2 = C2;
        this.line = line;
```

```
    }

    public Object visit(Visitor v, Object arg) {
      return v.visitIfSequentialCommand(this, arg);
    }
};
```

Comando If:

```
package minipascal.ast;
//============================================================================
public class Comando_IF extends Comando {
    public Expressao E1;
    public Comando C1;
    public int line;

    public Comando_IF(Expressao E1, Comando C1, int line)
    {
        this.E1 = E1;
        this.C1 = C1;
        this.line = line;
    }

    public Object visit(Visitor v, Object arg) {
      return v.visitIfCommand(this, arg);
    }
};
```

Comando If Else:

```
package minipascal.ast;
//============================================================================
public class Comando_IF_composto extends Comando {
    public Expressao E1;
    public Comando C1, C2;
    public int line;

    public Comando_IF_composto(Expressao E1, Comando C1, Comando C2, int
line)
    {
        this.E1 = E1;
        this.C1 = C1;
        this.C2 = C2;
        this.line = line;
    }
```

```
    public Object visit(Visitor v, Object arg) {
      return v.visitIfElseCommand(this, arg);
    }
};
```

O mesmo acontece com os demais não terminais. O resumo da implementação é representada no quadro a seguir. Para verificar detalhes de implementação os arquivos estão disponíveis em src/AST.

```
// PROGRAM
public class Program extends AST { ... }

// ID
public abstract class Id extends AST {};
public class Id_simples extends Id {...};
public class Id_seq extends Id { ... };

// Declaração
public abstract class Declaracao extends AST {};
public class Declaracao_seq extends Declaracao { ... };
public class Declaracao_simples extends Declaracao { ... };

// TIPO
public abstract class Tipo extends AST {};
public class Tipo_array extends Tipo { ... };
public class Tipo_simples extends Tipo { ... };

//VARIÁVEL
public abstract class Variavel extends AST {};
public class Var_simples extends Variavel { ... };

// EXPRESSAO
public abstract class Expressao extends AST {};
public class Expressao_composta extends Expressao { ... };
public class Expressao_s extends Expressao { ... };

// EXPRESSAO SIMPLES
public abstract class Expressao_simples extends AST {};
public class Expressao_simples_composta extends Expressao_simples { ...
};
public class Expressao_simples_simples extends Expressao_simples { ...
};

// TERMO
```

```
public abstract class Termo extends AST {};
public class Termo_composto extends Termo { ... };
public class Termo_unico extends Termo { ... };
```

Detecção de erros

Caso o programa fonte não esteja de acordo com as regras da linguagem, o analisador informa ao usuário o erro e para a compilação do programa. A mensagem de erro contém informações que ajudarão na solução do problema, informando o local e se possível a causa do erro.

A árvore sintática enquanto é montada, armazena a posição de cada token em relação ao arquivo lido. Com esse prática, é possível informar ao programar o local exato onde o analisador sintático encontrou um erro.

Programas com erros estão disponíveis no Anexo, com suas respectivas saídas.

Analisador de Contexto

O propósito do analisador de contexto é checar se as restrições de contextos são respeitadas. O contexto desse projeto consiste de:

- ❖ Analisador de escopo:
  - ➢ Define o alcance das variáveis dentro de um escopo. Existem 3 principais métodos, sendo: Monolítico, por blocos planos (flat) ou Nested.
- ❖ Analisador de tipo:
  - ➢ Define as regras que governam os tipos das expressões, além de verificar a compatibilidade entre os tipos.

Definido dessa forma, o analisador de contexto contará com duas fases principais. Cada fase será responsável por uma etapa. Uma consideração que deve ser feito é sobre a linguagem utilizada, é estática binding e de tipo estático.

Identificação

A primeira etapa, de identificação, consiste de verificar cada variável declarada com intuito de verificar se a regra de escopo não é quebrada. Uma variável pode assumir duas principais regras de escopo, sendo local ou global. A estrutura de blocos da linguagem mini Pascal é monolítico, dessa forma, todas as variáveis são globais e as regras do mini Pascal para identificação são:

(1) Nenhum identificador pode ser declarado mais de uma vez;
(2) Para cada aplicação de um identificador P, deve existir uma declaração correspondente P.

De acordo com o livro texto da matéria, um bom método para associar os identificadores é por meio da tabela de identificação. A tabela de identificação consiste de quatro métodos, com suas respectivas funções descritas abaixo:

Método	Descrição
public identificationTable()	Inicializa a tabela, sem nenhuma variável.
public void setTipo(Tipo T)	Seta o tipo da variável sendo analisada.
public void enter (String Id)	Verifica se a variável já foi declara, caso sim: - Dispara um erro de contexto e para a execução do compilador. caso não: - Adiciona a variável na tabela.
public Type retrieve (String Id)	Verifica se a variável já foi declara, caso não: - Dispara um erro de contexto e para a execução do compilador. caso sim: - Recupera o tipo da variável utilizando o

	identificador como chave.

Para armazenar a tabela foi adotado o uso de tabela Hash já disponibilizada pelo Java por meio do pacote `java.util.` Após a análise dos identificadores, a tabela deve conter o tipo e nome de todas as variáveis, conforme pode ser visualizado na Tabela abaixo.

```
program Lesson1Program3;
        var Int1, Int2 : integer;
        var Float : real;
        var Bool : boolean;
      var Nums : array[ 1 .. 3 ] of real;
begin

end.
```

identificationTable	
Id	**Tipo**
Int1	Integer
Int2	Integer
Float	Real
Bool	Boolean
Nums	array de real

Exemplo de programas com e sem erros nos identificadores estão no Anexo C. A implementação da tabela de identificação fica assim:

```
import java.util.*;
import minipascal.ast.Tipo;
import minipascal.ast.Tipo_simples;
import minipascal.ast.Tipo_array;
import minipascal.ast.Type;
public class identificationTable {
      private Tipo T;
      private int index;
      public Hashtable<String, Tipo> identificationTable;
    public identificationTable()
    {
      identificationTable = new Hashtable<String, Tipo>();
      T = null;
    }
    public void setTipo(Tipo T)
    {
      this.T = T;
    }
      public void enter (String Id)
```

```
	{
		if(T != null)
		{
			//System.out.println("ID:"+Id);
			//System.out.println("Type:"+T);
			Tipo t = identificationTable.get(Id);
			if(t == null)
			{
				identificationTable.put(Id, T);
			}else
			{
				System.out.println("\n\t[CONTEXT ERROR]");
				System.out.println("\tVariable already declared!
- " + Id );
				System.exit(0);
			}
		}
	}
	public Type retrieve (String Id)
	{
		//System.out.println("ID:"+Id);
		//System.out.println("Type:"+identificationTable.get(Id));
		Tipo t = identificationTable.get(Id);
		String spelling = null;
		Type type = Type.error;
		if( t != null)
		{
			if(t instanceof Tipo_simples)
			{
				Tipo_simples t_s = (Tipo_simples) t;
				spelling = t_s.spelling;
			}
			if(t instanceof Tipo_array)
			{
				Tipo_array t_a = (Tipo_array) t;
				Tipo_simples t_s = (Tipo_simples) t_a.T1;
				spelling = t_s.spelling;
			}
			if(spelling.equals(spellings[BOOL])) {
					type = Type.bool;
			}
			else if(spelling.equals(spellings[INTEGER])) {
					type = Type.integer;
			}
			else if(spelling.equals(spellings[REAL])) {
```

```
                              type = Type.real;
                  }
                  else if(spelling.equals(spellings[LITERAL])) {
                              type = Type.lit;
                  }
            }
            else
            {
                  System.out.println("\n\t[CONTEXT ERROR]");
                  System.out.println("\tVariable not declared! - " +
Id);
                  System.exit(0);
            }
            return type;
      }
      public static final byte BOOL             = 0,
                                          INTEGER    = 1,
                                          REAL             = 2,
                                          LITERAL    = 3,
                                          ERROR            = -1;

      public final static String[] spellings = {
        "boolean",
        "integer",
        "real",
        "literal" };

}
```

Visitor

O compilador deve ser desenvolvido de forma que o acesso a árvore montada seja padronizada. Para este trabalho, utilizamos o padrão de projeto Visitor, para organizar o acesso a AST.

Para um conjunto de nós, a classe visitor disponibiliza um conjunto de métodos para visitação. A classe Visitor é uma interface, definida no arquivo Visitor.java (`/src/AST/Visitor.java`) e contém as definições dos método de visita, cada classe concreta definida na AST possui um, conforme representado na implementação abaixo:

```
package minipascal.ast;

public interface Visitor
{
```

```
            // Program
            public Object visitProgram (Program prog, Object arg);
            // Commands
            public Object visitIfCommand(Comando_IF com, Object arg);
            public Object visitIfElseCommand(Comando_IF_composto com,
Object arg);
            public Object visitIfSequentialCommand(Comando_Seq com,
Object arg);
            public Object visitVarCommand(Comando_VAR com, Object arg);
            public Object visitWhileCommand(Comando_WHILE com, Object
arg);
            // Body
            public Object visitCorpo(Corpo com, Object arg);
            // Declaration
            public Object visitSimpleDeclaration(Declaracao_simples com,
Object arg);
            public Object visitCompoundDeclaration(Declaracao_seq com,
Object arg);
            // Expressions
            public Object visitSimpleExpression(Expressao_s com, Object
arg);
            public Object visitCompoundExpression(Expressao_composta
com, Object arg);
            // Expression Simple
            public Object
visitSimpleExpressionSimple(Expressao_simples_simples com, Object arg);
            public Object
visitCompoundExpressionSimple(Expressao_simples_composta com, Object
arg);
            // Fator
            public Object visitFatorVar(Fator_VAR com, Object arg);
            public Object visitFatorLit(Fator_LIT com, Object arg);
            public Object visitFatorExp(Fator_EXP com, Object arg);
            // Identification
            public Object visitSimpleId(Id_simples com, Object arg);
            public Object visitCompoundId(Id_seq com, Object arg);
            // Term
            public Object visitSimpleTerm(Termo_unico com, Object arg);
            public Object visitCompoundTerm(Termo_composto com, Object
arg);
            // Type
            public Object visitSimpleType(Tipo_simples com, Object arg);
            public Object visitArrayType(Tipo_array com, Object arg);
            // Variable
            public Object visitSimpleVar(Var_simples com, Object arg);
```

```
}
```

Um objeto que implementa a interface Visitor, chamado também de objeto visitor, contém a implementação desses métodos. O analisador de contexto é um objeto visitor, que realiza identificação e checagem de tipo.

Cada nó concreto da AST deve implementar o método de visitação. Por exemplo, SimpleVar implementa o método de visitação dessa forma:

```
public Object visit(Visitor v, Object arg) {
      return v.visitSimpleVar(this, arg);
    }
```

Assim cada nó da AST chama o método de visitação de acordo com o seu "tipo". Por exemplo:

- O método visit do SimpleVar chama o método `visitSimpleVar`;
- O método visit do TipoSimples chama o método `visitSimpleType`.

Por fim cada método de visitação `visitN` irá verificar se as restrições de contextos são respeitadas para a classe N. Um resumo com os métodos utilizados nesse projeto é disponibilizado a seguir, e em seguida detalhes dos códigos de alguns métodos. Com intenção de evitar o prolongamento do documento, uma vez que a implementação é muito parecida, apenas algumas implementações dos métodos serão analisadas.

Regra	Método	Descrição
Program (P)	`visitProgram`	Verifica se o programa está bem formado, retorna Null.
Identificador (Id)	`visitSimpleId` `visitCompoundId`	Verifica se o Id está bem formado, retorna ou o Id ou Null. Dependendo da fase de compilação: - Insere na tabela de identificação - Verificar se o Id foi inserido.
Corpo (Cr)	`visitCorpo`	Verifica se o Corpo está bem formado, retorna ou o Id ou Null.
Declaração (D)	`visitSimpleDeclaration` `visitCompoundDeclaration`	Verifica se a Declaração está bem formado, retorna Null. Durante a análise, seta o tipo da variável.
Tipo (T)	`visitSimpleType` `visitArrayType`	Verifica se o Array está bem formado, retorna Null.

Comando (C)	`visitIfCommand`
	`visitIfElseCommand`
	`visitIfSequentialCommand`
	`visitVarCommand`
	`visitWhileCommand`
		Verifica se o Comando está bem formado, retorna Null.

Variável (V)	`visitSimpleVar`	Verifica se a Variável está bem formado, retorna o tipo da variável.
Expressão (E)	`visitSimpleExpression` `visitCompoundExpression`	Verifica se a Expressão está bem formado, retorna o tipo da expressão.
Expressão Simples (Es)	`visitSimpleExpressionSimple` `visitCompoundExpressionSimple`	Verifica se a Expressão está bem formado, retorna o tipo da expressão.
Termo (Tr)	`visitSimpleTerm` `visitCompoundTerm`	Verifica se o Termo está bem formado, retorna o tipo do Termo.

A implementação dos métodos é simples. No caso da regra Programa (<programa> ::= program <id> ; <corpo> .), a implementação do `visitProgram` chama primeiro `I.visit`, para verificar se o Id está bem formulado, depois chama `Cr.visit`, para verificar se o Corpo está bem formulado. Segue a implementação:

```
// Program
      public Object visitProgram (Program prog, Object arg)
      {
            prog.I.visit(this, null);
            prog.Cr.visit(this, null);
            return null;
      }
```

Uma implementação interessante é do visitCompoundTerm. A regra é:

(1) <op-mul> ::= (* | / | and)

(2) <termo> ::= <fator>((<op-mul> <fator>)* | <vazio>)

Caso o <fator> seja booleano, não é/será possível realizar as operações de multiplicação ou de divisão, contudo, se o <fator> for um real, essas operações são permitidas e o operador **and** não. E mais complicado ainda, se a operações envolvem dois operandos do tipo Real, o resultado deve ser do tipo Real, caso os operandos sejam do tipo Inteiro, o resultado será Inteiro. **Todas essas decisões dependem do contexto!**

A implementação segue abaixo:

```
public Object visitCompoundTerm(Termo_composto com, Object arg)
```

```
	{
		Type t = null;
		Type T1 = (Type) com.F1.visit(this, null);
		Type T2 = (Type) com.F2.visit(this, null);
		String Operador = com.OP_MUL;

		// <op-mul> ::= ( * | / | and )
		//  ( * | / )     ONLY WITH INTEGER, REAL, LITERAL
		// and                    ONLY WITH BOOLEAN
		if(Operador.equals("*") || Operador.equals("/"))
		{
			if(T1.getKind() == Type.BOOL || T2.getKind() ==
Type.BOOL)
			{
				t = Type.error;
			}
			else
			{
				if(T1.getKind() == Type.REAL || T2.getKind() ==
Type.REAL)
					t = Type.real;
				else
					t = Type.integer;
			}
		}
		else if(Operador.equals("and"))
		{
			if(T1.getKind() != Type.BOOL || T2.getKind() !=
Type.BOOL)
			{
				t = Type.error;
			}
			else
			{
				t = Type.bool;
			}
		}

		return t;
	}
```

Durante análise do contexto, as regras de identificação são checadas pelo método visitSimpleId e visitSimpleId; A seguir a implementação do visitSimpleId, a implementação de visitSimpleId é similar:

```
// Identification
      public Object visitSimpleId(Id_simples com, Object arg)
      {
            if(phase == 0)
                  mapTable.enter(com.spelling);
            else
                  mapTable.retrieve(com.spelling);

            return com.spelling;
      }
```

Além de verificar as regras de contexto, esses métodos, quando necessário, decoram a AST. Decorar a AST quer dizer, atualiza as informações da AST de acordo com a análise feita no contexto atual. Por exemplo, o método visitFatorLit, durante a análise de contexto verifica o tipo do literal e atualiza a AST, conforme a implementação abaixo:

```
public Object visitFatorLit(Fator_LIT com, Object arg)
      {
            if(com.kind == Token.TRUE || com.kind == Token.FALSE)
                  com.type = Type.bool;
            else if(com.kind == Token.INTLITERAL)
                  com.type = Type.integer;
            else if(com.kind == Token.FLOATLIT)
                  com.type = Type.real;
            else
                  com.type = Type.lit;                // DECORATION

            return com.type;
      }
```

Checagem de tipo

O Mini-Pascal conta com três tipos simples e um tipo agregado, conforme na regra abaixo:

(1) <tipo> ::=
array [<literal> **..** <literal>] **of** <tipo>
| **integer**
| **real**
| **boolean**

Esses tipos estão definidos na classe Type.java (/src/AST/Type.java), conforme implementação a seguir:

```
package minipascal.ast;
```

```
 public class Type {
      private byte kind;

      public static final byte BOOL = 0, INTEGER = 1, REAL = 2, ARRAY =
3, LITERAL = 4, ERROR = 5;

      public Type (byte kind) {
            this.kind = kind;
      }
      public byte getKind()
      {
            return kind;
      }
      public boolean equals (Object other) {
            // Test whether this type is equivalent to other.
            Type otherType = (Type) other;
            return (this.kind == otherType.kind
                              || this.kind == ERROR
                              || otherType.kind == ERROR );
      }
      public static Type bool = new Type(BOOL);
      public static Type integer = new Type(INTEGER);
      public static Type real = new Type(REAL);
      public static Type array = new Type(ARRAY);
      public static Type lit = new Type(LITERAL);
      public static Type error = new Type(ERROR);

      public final static String[] spellings = {

"Boolean",

"Integer",

"Real",

"Array",

"Literal",

"ERROR" };

 }
```

Diversos programas são analisados no Anexo D com base nas regras de contexto definidas abaixo:

Regra	**if** <expressão> **then** <comando> (**else** <comando> | <vazio>)
Descrição da regra de contexto: (1) A <expressão> deve ser do tipo Booleano;	

Regra	<variável> **:=** <expressão>
Descrição da regra de contexto: (1) Se a <variável> for Booleana: (a) A <expressão> deve ser do mesmo tipo (Booleano); (2) Se a <variável> for Inteira: (a) A <expressão> deve ser do tipo Inteira; (3) Se a <variável> for Real: (a) A <expressão> pode ser Inteira ou Real;	

Regra	**while** <expressão> **do** <comando>
Descrição da regra de contexto: (1) A <expressão> deve ser do tipo Booleano;	

Regra	<fator> ::= <variável>
Descrição da regra de contexto: (1) <fator> será do tipo da variável	

Regra	<fator> ::= <literal>
Descrição da regra de contexto: (1) Se o literal for: (a) True | False - <fator> será do tipo Booleano; (b) <INTLIT> - <fator> será do tipo Inteiro. (i) <INTLIT> é conjunto de número sem ponto (Ex: 1 2 3); (c) <FLOATLIT> - <fator> será do tipo Real; (i) <FLOATLIT> - conjunto de número com ponto (Ex: 1.1 .1 1.)	

Regra	<fator> ::= **(** <expressão> **)**

Descrição da regra de contexto:
(1) <fator> será do tipo da expressão, menos Booleana.

Regra	<termo> ::= <fator>((<op-mul> <fator>)* | <vazio>)

Descrição da regra de contexto:
(1) Se o <termo> for simples:
  (a) <termo> recebe o tipo de acordo com <fator>;
(2) Se o <termo> conter um <op-mul>:
  (a) Os operadores < * | / > só funcionam com operandos do tipo Inteiro ou Real;
    (i) <termo> será Real se um dos operandos for Real, será inteiro caso contrário.
  (b) O operando < **and** > somente funciona sobre tipo Booleano;
    (i) <termo> será do tipo Booleano.

Regra	<expressão-simples> ::= <termo> ((<op-ad><termo>)* | <vazio)

Descrição da regra de contexto:
(1) Se a <expressão-simples> conter <op-ad>:
  (a) Os operadores < * | / > só funcionam com operandos do tipo Inteiro ou Real;
    (i) <termo> será Real se um dos operandos for Real, será inteiro caso contrário.
  (b) O operando < **or** > somente funciona sobre tipo Booleano;
    (i) <termo> será do tipo Booleano.
(2) Caso contrário:
  (a) <expressão-simples> recebe o tipo de acordo com <termo>.

Regra	<expressão> ::= <expressão-simples> (<op-rel> <expressão-simples> | <vazio>)

Descrição da regra de contexto:
(3) Se a <expressão> contiver um <op-rel>:
  (a) Os operadores < < | <= | > | >= > só funcionam com operandos diferentes de Booleanos:
  (b) Os operadores < = | <> > funcionam sobre Booleanos desde que todos os dois operandos sejam booleanos;
  (c) Os operadores < = | <> > funcionam sobre Inteiros e Reais:
  (d) <expressão> será Booleana.
(4) Caso contrário:
  (a) <expressão> recebe o tipo de acordo com <expressão-simples>.

O analisador de contexto implementado até aqui funciona bem e reconhece a maioria dos erros de contextos. Existem casos ainda não implementados para análise do contexto, são eles:

- Análise de tipo agregado mais complexos;
- Análise da expressão dos indexadores dos tipos agregados;

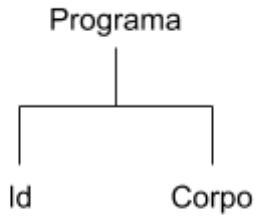
Impressão da Árvore

Para imprimir a árvore, um objeto visitor foi implementado. O Printer percorre toda a AST e reescreve o programa no terminal (utilizando apenas a AST!). A seguir detalhes da implementação do printer.

De acordo com a AST, tem-se que a regra:

<programa> ::= **program** <id> **;** <corpo> **.**

é representada assim na árvore:



Dessa forma, para realizar a imprssẽao da árvore, o Parser (/src/Parser.java), implementa:

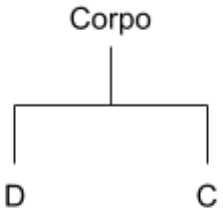
(Observe o comando o que está em negrito na regra acima é passado para impressão por meio do comando print);

```
// Program
	public Object visitProgram (Program prog, Object arg)
	{
		print("program ");
		prog.I.visit(this, null);
		print(";");
		print_NewLine();
		prog.Cr.visit(this, null);
		print(".");
	print_NewLine();
		return null;
	}
```

De maneira semelhante para a regra:

<corpo> ::= (<declaração> ;)* **begin** (<comando> **;**)* **end**

com representação na AST:



A implementação fica então:

```
public Object visitCorpo(Corpo com, Object arg)
    {
       phase = 0;
       //System.out.print("\tChecking Declarations");
       com.D.visit(this, null);
       print_NewLine();
       print("begin");
       print_NewLine();
       increase_level();

       phase = 1;
       //System.out.print("\tChecking Commands");
       if(com.C != null)
             com.C.visit(this, null);
       decrease_level();
       print("end");
       return null;
    }
```

As demais regras são bastantes parecidas com as apresentadas. A seguir o exemplo de funcionamento do Printer:

Comando para impressão:
$ java Main -o -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Printer/P1.mp

Como entrada utilizamos um programa não identado.

```
!!
----------------------------------------------------------------------
   THIS IS A PROGRAM TO TEST A MINI-PASCAL COMPILER.
----------------------------------------------------------------------
PROJECT TITLE: PRETTY PRINT TESTER
PURPOSE OF PROJECT: READ THE FILE AND PRINT THE CODE INDENTED
VERSION or DATE: 13/01/2019
HOW TO START THIS PROJECT:
AUTHORS:    GUSTAVO MARQUES ( @GUTODISSE )
USER INSTRUCTIONS:
```

```
!

program Lesson1Program3; var Nums : array[ 1 .. 3 ] of array[ 1 .. 3 ] of real; var Num1, Num2, Num3, Sum : integer; var testes : boolean; begin  Num1 := 1; Num3 := 11; Num3 := 1; testes := true; Sum  := Num1 + Num2;  if Num1 <= 2 then testes := false; while (testes) do begin testes := false; end; begin if Num1 <= 2 then begin testes := false; end ; Sum  := Num1 + Num2 + 3 + ( 2 < 10 ); end; end.
```

O programa imprime, durante a leitura da AST, o código já identado. A seguir a saída do compilador:

```
program Lesson1Program3;
var Nums : array[ 1 .. 3 ] of array[ 1 .. 3 ] of real
var Num1, Num2, Num3, Sum : integer;t
var testes : boolean;
begin
   Num1 := 1;
   Num3 := 11;
   Num3 := 1;
   testes := true;
   Sum := Num1 + Num2;
   if Num1 <= 2 then testes := false;
   while  ( testes )  do
         testes := false;
   begin
         if Num1 <= 2 then testes := false;
         Sum := Num1 + Num2 + 3 +  ( 2 < 10 ) ;
   end
end.
```

Novamente, o código impresso no terminal só depende da leitura da AST. Uma montagem errada da AST, geraria uma montagem de código errada. **Diversos aspectos do parser foram corrigidos após execução do Printer, por exemplo: a leitura de expressões.**

Gerador de código

O gerador de código é responsável por traduzir o código fonte em código objeto. Para realizar essa tradução, o gerador de código depende da linguagem fonte e da máquina que será executada. Esse compilador utiliza a linguagem miniPascal, como linguagem fonte, e a máquina TAM como alvo.

Do ponto de vista da implementação, o gerador de código é um objeto visitor, ou seja de um conjunto de métodos que percorre a AST, onde cada método visit implementa a geração/tradução de código apropriada. Para os comandos de controle utiliza-se a técnica de *backpatching* na correção dos *Jumps*.

A linguagem miniPascal é monolítica, não contém funções ou procedimentos. Todas essas característica reduzem a complexidade da implementação do gerador de código.

A descrição da máquina TAM pode ser encontrado no Apêndice C do livro texto, não sendo necessário descrever aqui.

Semântica

Comandos:

V := E - Análise E para produzir um valor que será armazenado em V;

C1;C2 - Executa C1 depois C2;

if E then C1 - Análise E para produzir um valor booleano, caso seja verdadeiro executa C1;

if E then C1 else C2 - Análise E para produzir um valor booleano, caso seja verdadeiro executa C1, caso contrário, executa C2;

while E do C - Análise E para produzir um valor booleano, caso E seja verdadeiro (*true*) então executa C e repete. Caso E seja falso (false) então o comando encerra.

Expressões:

Uma operação binária, E1 O E2, produz um valor aplicando o operador O sobre as duas expressões E1 e E2.

Declaração:

var I : T - aloca espaço de memória para a variável I. O conteúdo de I é desconhecido;

Templates de código

Tratando-se de tradução, existem muitas maneiras de organizar instruções de maneira a executar a mesma operação (mesma semântica), logo, conclui-se que existem diversas traduções corretas de um dado programa. Com intuito de padronizar a geração de código, adotou-se os templates de código abaixo, para tradução dos comandos.

execute[<programa> ::= program <id> ; <corpo> .]

1. **EXECUTE** <corpo>
2. POP (0) (memória usada por <corpo>)
3. **HALT**

execute[<variável> **:=** <expressão>]

1. **EVALUATE** E
2. **STORE** address[<variável>]

execute[if <expressão> then <comando>]

1. **EVALUATE** E
2. **JUMPIF** (0) 4
3. **EXECUTE** C1
4.

execute[if <expressão> then <comando> else <comando>]

1. **EVALUATE** E
2. **JUMPIF** (0) 5
3. **EXECUTE** C1
4. **JUMPIF** (0) 6
5. **EXECUTE** C2
6.

execute[while <expressão> do <comando>]

1. **JUMPIF** (0) 3
2. **EXECUTE** C
3. **EVALUATE** E
4. **JUMPIF** (1) 2

execute[begin (<comando> ;)* end]

1. **EXECUTE** C1 (considerando pelo menos 1 <comando>)
2. **EXECUTE** C2

…. N. **EXECUTE** CN

execute[var <id> (**,** <id>)* **:** <tipo>] O. **PUSH** size(Tamanho do tipo vezes tamanho)

Tradução

Com base nos templates, gera-se os códigos a seguir:

Arquivo: Programas/Code/P1.mp	Arquivo: Programas/Code/P1.txt
Código fonte	Código de máquina gerado
program Lesson1Program3; var Num1: real; begin Num1 := 1.1 + 1 + 2 * (1 + 1); end.	0 : PUSH 1 1 : LOADL 1.1 2 : LOADL 1 3 : CALL (0) add[0] 4 : LOADL 2 5 : LOADL 1 6 : LOADL 1 7 : CALL (0) add[0] 8 : CALL (0) mult[0] 9 : CALL (0) add[0] 10 : STORE (1) 0[0] 11 : POP (1) 0 12 : HALT
Mapa da memória: [ADDR] Identifier [0] Num1	

Arquivo: Programas/Code/P2.mp	Arquivo: Programas/Code/P2.txt
Código fonte	Código de máquina gerado

program Lesson1Program3;
var Num1: real;
begin
Num1 := 0;
 	
if(Num1 < 1) then Num1 := 1;
 	
end.
	0 : PUSH 1
	1 : LOADL 0
	2 : STORE (1) 0[0]
	3 : LOADA 0[0]
	4 : LOADL 1
	5 : CALL (0) lt[0]
	6 : JUMPIF (0) 10[0]
	7 : LOADL 1
	8 : STORE (1) 0[0]

	9 : JUMP 0[0] 10 : POP (1) 0 11 : HALT
Mapa da memória: [ADDR] Identifier [0] Num1	

Arquivo: Programas/Code/P3.mp	Arquivo: Programas/Code/P3.txt
Código fonte	Código de máquina gerado

program Lesson1Program3;
var Num1: real;
begin
Num1 := 0;
 	
if(Num1 < 1) then Num1 := 1 else Num1 := 0;
 	
end.
	0 : PUSH 1
	1 : LOADL 0
	2 : STORE (1) 0[0]
	3 : LOADA 0[0]
	4 : LOADL 1
	5 : CALL (0) lt[0]
	6 : JUMPIF (0) 10[0]
	7 : LOADL 1
	8 : STORE (1) 0[0]
	9 : JUMP 12[0]
	10 : LOADL 0
	11 : STORE (1) 0[0]
	12 : POP (1) 0
	13 : HALT

Mapa da memória: [ADDR] Identifier [0] Num1	

Arquivo: Programas/Code/P4.mp	Arquivo: Programas/Code/P4.txt
Código fonte	Código de máquina gerado

program Lesson1Program3;
var Bool: boolean;
begin
Bool := true;
 	
while(Bool) do Bool := false;
 	
end.
	0 : PUSH 1
	1 : LOADL true
	2 : STORE (1) 0[0]
	3 : JUMPIF (0) 6[0]
	4 : LOADL false
	5 : STORE (1) 0[0]
	6 : LOADA 0[0]
	7 : JUMPIF (1) 4[0]
	8 : POP (1) 0
	9 : HALT

Mapa da memória: [ADDR] Identifier [0] Bool	

Avaliando expressões

Expressão: 1+10+(6/2)*6

Código gerado:

```
1 :    LOADL 1
2 :    LOADL 10
3 :    CALL (0) add[0]
4 :    LOADL 6
5 :    LOADL 2
6 :    CALL (0) div[0]
7 :    LOADL 6
8 :    CALL (0) mult[0]
9 :    CALL (0) add[0]
```

Análise da pilha:

1:

1

2:

10
1

3:

11

4:

5
11

5:

2
6
11

6:

3
11

7:

6
3
11

8:

18
11

9:

29

Expressão: (2*(2*(2+2))+((1+1)*(2+2))

Código gerado:

```
1 :    LOADL 2
2 :    LOADL 2
3 :    LOADL 2
4 :    LOADL 2
5 :    CALL (0) add[0]
6 :    CALL (0) mult[0]
7 :    CALL (0) mult[0]
8 :    LOADL 1
9 :    LOADL 1
10 :   CALL (0) add[0]
11 :   LOADL 2
12 :   LOADL 2
13 :   CALL (0) add[0]
14 :   CALL (0) mult[0]
15 :   CALL (0) add[0]
```

Análise da pilha:

4:

2
2
2
2

5:

4
2
2

6:

8
2

7:

16

8:

1
16

9:

1
1
16

10:

2
16

11:

2
2
16

12:

2
2
2
16

13:

4
2
16

14:

8
16

15:

24

Alocação de memória

Uma vez que o minipascal não aceita funções ou procedimentos e as variáveis são declaradas antes de seu uso, todas as variáveis de um dado programa são então globais.

Considere o programa abaixo:

Código fonte	Código gerado

`program Lesson1Program3;`
`  var Array1: array[ 1 .. 3 ] of real;`
`  var Array2: array[ 2 .. 30 ] of real;`
`  var Real: real;`
`  var Int:  integer;`
`  var Bool: boolean;`
`begin`
`  Int := 1;`
`  Int := 1 + Int;`
`  Array2[2] := 1;`
`end.`
	`0 :  PUSH 3`
	`1 :  PUSH 29`
	`2 :  PUSH 1`
	`3 :  PUSH 1`
	`4 :  PUSH 1`
	`5 :  LOADL 1`
	`6 :  STORE (1) 33[0]`
	`7 :  LOADL 1`
	`8 :  LOADA 33[0]`
	`9 :  CALL (0) add[0]`
	`10 :  STORE (1) 33[0]`
	`11 :  LOADL 2`
	`12 :  LOADL 3`
	`13 :  CALL (0) add[0]`
	`14 :  LOADL 1`
	`15 :  STOREI (1)`
	`16 :  POP (35) 0`
	`17 :  HALT`

A alocação de memória fica dessa forma:

Variável	Endereço
Array1	0
Array2	3
Real	32
Int	33
Bool	34

Análise da pilha e das variáveis por linha do código gerado:

Execução linha por linha:

5:
Pilha:

1

6:
Pilha:

Variáveis:

33	1

7:
Pilha:

1

Variáveis:

33	1

8:
Pilha:

1
1

Variáveis:

33	1

9:
Pilha:

2

Variáveis:

33	1

10:
Pilha:

Variáveis:

Addr	Valor
33	2

11:
Pilha:

2

Variáveis:

Addr	Valor
33	2

12:
Pilha:

3
2

Variáveis:

Addr	Valor
33	2

13:
Pilha:

5

Variáveis:

Addr	Valor
33	2

14:
Pilha:

1
5

Variáveis:

Addr	Valor
33	2

15:
Variáveis:

Addr	Valor
33	2
5	1

16:

17:

Ambiente:

Variável	Endereço
Array1	0
Array2	3
Real	32
Int	33
Bool	34

Salvando código

Para o compilador salvar o código gerado basta adicionar o parâmetro “-f” no comando. Por exemplo:

```
$ java Main -i P7.mp -f P7.txt
```

Observação:

Antes de seguir o relatórios, o gerador de código não está completo. A única operação com Array que funciona atualmente é o de atribuição, e ainda assim não está completa. O comando de atribuição implementado não funciona com vetores multidimensionais.

O gerador de código não foi completamente implementado devido principalmente ao tempo, ou melhor, a falta de tempo.

Conclusão

Desenvolver um compilador, dado a complexidade do projeto, se mostrou um grande desafio, ao final obter um compilador capaz de gerar código é gratificante. Alinhar o estudo teórico da matéria junto ao desenvolvimento do projeto é interessante, fortalece o conhecimento visto que enfrentamos problemas que só acontecem durante a execução/implementação de um projeto.

O maior desafio foi o padrão de projeto visitor (Visitor Pattern). Como solução buscamos conteúdo extra online, a maior ajuda encontramos em videoaulas no youtube sobre o Visitor.

Um ponto crítico do projeto é o tamanho do projeto, atualmente esse documento conta com 60 páginas e ainda faltam itens que serão acrescentados. O projeto é grande e complexo, compartilhar o tempo da matéria com prova e outras matérias é crítico.

Por fim, esse projeto conta com muitos detalhes e eu, particularmente, tentei abordar o máximo de detalhe nesse documento, contudo o relatório tornou-se extremamente grande e o trabalho de documentar cansativo, o que me levou nos últimos tópicos a resumi muito. Devido a isso, os últimos tópicos, em especial a geração de código, pode está faltando informações ou descrito de maneira imprecisa, ao que peço desculpas.

Anexos

Anexo A

Para executar apenas o analisador sintático, basta acrescentar o comando “-s”;

Comando:
java Main -s -i <endereço do código fonte>

A seguir diversos programas (input) são analisados pela classe scanner, e a seguir a resposta da classe (output).

Sem erro(s):

Comando:
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P1.mp
Programa P1.mp:
!! ---------- THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL. ---------- PROJECT TITLE: LEXICAL TESTER PURPOSE OF PROJECT: USE ALMOST ALL TOKENS OF MINI-PASCAL LANGUAGE VERSION or DATE: 5/01/2019 HOW TO START THIS PROJECT: AUTHORS: GUSTAVO MARQUES (@GUTODISSE) USER INSTRUCTIONS: !
Output:

Scanner on: P1.mp
[TOKEN]: <EOT>
  Kind: 20
  Spell:

Comando:
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P2.mp

Programa P2.mp:

```
program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
```

Output:

```
Scanner on: P2.mp
[TOKEN]:    PROGRAM
   Kind:    4
   Spell:    program
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    Lesson1Program3
[TOKEN]:    <SEMICOLON>
   Kind:    21
   Spell:    ;
[TOKEN]:    <EOT>
   Kind:    20
   Spell:
```

Comando:

```
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P3.mp
```

Programa P3.mp:

(PROGRAMA COM ERRO NA ESTRUTURA, O SCANNER NÃO ESTÁ PERCEBENDO QUE FALTA DECLARAÇÃO "VAR" ANTES DAS DECLARAÇÕES! O SCANNER FAZ APENAS O PAPEL DELE, QUE É CLASSIFICAR OS TOKENS)

```
var
        Num1, Num2, Num3, Sum : integer;
        Nums : array[ 1 .. 3 ] of real;
        testes : boolean;
```

Output:

```
Scanner on: P3.mp
[TOKEN]:    VAR
   Kind:    7
   Spell:    var
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    Num1
[TOKEN]:    <COMMA>
   Kind:    26
   Spell:    ,
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    Num2
[TOKEN]:    <COMMA>
```

```
   Kind:    26
   Spell:    ,
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    Num3
[TOKEN]:    <COMMA>
   Kind:    26
   Spell:    ,
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    Sum
[TOKEN]:    <COLON>
   Kind:    25
   Spell:    :
[TOKEN]:    INTEGER
   Kind:    9
   Spell:    integer
[TOKEN]:    <SEMICOLON>
   Kind:    21
   Spell:    ;
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    Nums
[TOKEN]:    <COLON>
   Kind:    25
   Spell:    :
[TOKEN]:    ARRAY
   Kind:    8
   Spell:    array
[TOKEN]:    [
   Kind:    32
   Spell:    [
[TOKEN]:    <INTLITERAL>
   Kind:    1
   Spell:    1
[TOKEN]:    <DOT>
   Kind:    27
   Spell:    .
[TOKEN]:    <DOT>
   Kind:    27
   Spell:    .
[TOKEN]:    <INTLITERAL>
   Kind:    1
   Spell:    3
[TOKEN]:    ]
   Kind:    31
   Spell:    ]
[TOKEN]:    OF
   Kind:    19
   Spell:    of
```

```
[TOKEN]:   REAL
   Kind:   10
   Spell:   real
[TOKEN]:   <SEMICOLON>
   Kind:   21
   Spell:   ;
[TOKEN]:   <IDENTIFIER>
   Kind:   0
   Spell:   testes
[TOKEN]:   <COLON>
   Kind:   25
   Spell:   :
[TOKEN]:   BOOLEAN
   Kind:   11
   Spell:   boolean
[TOKEN]:   <SEMICOLON>
   Kind:   21
   Spell:   ;
[TOKEN]:   <EOT>
   Kind:   20
   Spell:
```

Comando:

```
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P4.mp
```

Programa P4.mp:

```
        Num1 := 1;  ! TEST COMMENT
        Num3 := 1.1;
        Num3 := .1;
        testes := true;
```

Output:

```
Scanner on: P4.mp
[TOKEN]:   <IDENTIFIER>
   Kind:   0
   Spell:   Num1
[TOKEN]:   <IS>
   Kind:   28
   Spell:   :=
[TOKEN]:   <INTLITERAL>
   Kind:   1
   Spell:   1
[TOKEN]:   <SEMICOLON>
   Kind:   21
   Spell:   ;
[TOKEN]:   <IDENTIFIER>
```

```
   Kind:    0
   Spell:    Num3
[TOKEN]:    <IS>
   Kind:    28
   Spell:    :=
[TOKEN]:    <FLOATLIT>
   Kind:    3
   Spell:    1.1
[TOKEN]:    <SEMICOLON>
   Kind:    21
   Spell:    ;
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    Num3
[TOKEN]:    <IS>
   Kind:    28
   Spell:    :=
[TOKEN]:    <FLOATLIT>
   Kind:    3
   Spell:    .1
[TOKEN]:    <SEMICOLON>
   Kind:    21
   Spell:    ;
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    testes
[TOKEN]:    <IS>
   Kind:    28
   Spell:    :=
[TOKEN]:    TRUE
   Kind:    29
   Spell:    true
[TOKEN]:    <SEMICOLON>
   Kind:    21
   Spell:    ;
[TOKEN]:    <EOT>
   Kind:    20
   Spell:
```

Comando:
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P5.mp
Programa P5.mp:
Sum := Num1 + Num2; ! TEST COMMENT

Output:

```
Scanner on: P5.mp
[TOKEN]    <IDENTIFIER>
   Kind:    0
   Spell:    Sum
[TOKEN]    <IS>
   Kind:    28
   Spell:    :=
[TOKEN]    <IDENTIFIER>
   Kind:    0
   Spell:    Num1
[TOKEN]    <OP_AD>
   Kind:    22
   Spell:    +
[TOKEN]    <IDENTIFIER>
   Kind:    0
   Spell:    Num2
[TOKEN]    <SEMICOLON>
   Kind:    21
   Spell:    ;
[TOKEN]    <EOT>
   Kind:    20
   Spell:
```

Comando:

```
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P6.mp
```

Programa P6.mp:

```
        if Num1 <= 2 then testes := false;
        while testes do testes := false;
```

Output:

```
Scanner on: P6.mp
[TOKEN]    IF
   Kind:    12
   Spell:    if
[TOKEN]    <IDENTIFIER>
   Kind:    0
   Spell:    Num1
[TOKEN]    <OP_REL>
   Kind:    24
   Spell:    <=
[TOKEN]    <INTLITERAL>
   Kind:    1
   Spell:    2
```

```
[TOKEN]:    THEN
   Kind:    14
   Spell:    then
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    testes
[TOKEN]:    <IS>
   Kind:    28
   Spell:    :=
[TOKEN]:    FALSE
   Kind:    30
   Spell:    false
[TOKEN]:    <SEMICOLON>
   Kind:    21
   Spell:    ;
[TOKEN]:    WHILE
   Kind:    15
   Spell:    while
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    testes
[TOKEN]:    DO
   Kind:    16
   Spell:    do
[TOKEN]:    <IDENTIFIER>
   Kind:    0
   Spell:    testes
[TOKEN]:    <IS>
   Kind:    28
   Spell:    :=
[TOKEN]:    FALSE
   Kind:    30
   Spell:    false
[TOKEN]:    <SEMICOLON>
   Kind:    21
   Spell:    ;
[TOKEN]:    <EOT>
   Kind:    20
   Spell:
```

Comando:

```
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P7.mp
```

Programa P7.mp:

```
        begin
        !! MULTILINE COMMENT
```

```
WORKING
!
end
```

Output:

```
Scanner on: P7.mp
[TOKEN]:    BEGIN
   Kind:    5
   Spell:    begin
[TOKEN]:    END
   Kind:    6
   Spell:    end
[TOKEN]:    <EOT>
   Kind:    20
   Spell:
```

Comando:

```
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P8.mp
```

Programa P8.mp:

```
begin
end.
```

Output:

```
Scanner on: P8.mp
[TOKEN]:    BEGIN
   Kind:    5
   Spell:    begin
[TOKEN]:    END
   Kind:    6
   Spell:    end
[TOKEN]:    <DOT>
   Kind:    27
   Spell:    .
[TOKEN]:    <EOT>
   Kind:    20
   Spell:
```

Com erro(s):

A seguir um programa com erro é analisado pela classe scanner, conforme demonstra a saída do programa. A classe reconhece um símbolo não esperado na linha 14, coluna 2 do arquivo E1.mp. Além de dizer o local, exibe a linha com uma indicação visual para o erro.

Comando:
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Scanning/P8.mp
Programa P8.mp:
begin end.
Output:
Scanner on: E1.mp [TOKEN]: BEGIN Kind: 5 Spell: begin [TOKEN]: <IDENTIFIER> Kind: 0 Spell: en E1.mp:14-2: error: $ unexpected en$d. ^

Anexo B

Para executar até o analisador sintático, basta acrescentar o comando “-p”;

Comando:
java Main -p -i <endereço do código fonte>

A seguir diversos programas (input) são analisados pela classe scanner, e a seguir a resposta da classe (output).

Sem erro(s):

Comando:
$ java Main -s -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Parsing/P1.mp
Programa P1.mp:

```
!!
-------------------------------------------------------------------------
        THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL.
-------------------------------------------------------------------------
PROJECT TITLE: LEXICAL TESTER
PURPOSE OF PROJECT: USE ALMOST ALL TOKENS OF MINI-PASCAL LANGUAGE
VERSION or DATE: 16/03/2019
HOW TO START THIS PROJECT:
AUTHORS:     GUSTAVO MARQUES ( @GUTODISSE )
USER INSTRUCTIONS:
!

program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
var Num1, Num2, Num3, Sum : integer;
var Nums : array[ 1 .. 3 ] of real;
var testes : boolean;
begin
        Num1 := 1;  ! TEST COMMENT
        Num3 := 1;
        Num3 := .1;
        testes := true;

        Sum  := Num1 + Num2; ! TEST COMMENT

        if Num1 <= 2 then testes := false;
        while testes do testes := false;

        begin
        !! MULTILINE COMMENT
```

```
        WORKING
        !
        end;

end.
```

Output:

```
Checking Syntax                 OK
```

Com erro(s):

A seguir um programa com erro é analisado pela classe scanner, conforme demonstra a saída do programa. A classe reconhece um símbolo não esperado na linha 15, coluna 5 do arquivo E1.mp. Além de dizer o local, exibe a linha com uma indicação visual para o erro.

O erro que o programa E1 contém está na regra:

<declaração> ::= **var** <id> (**,** <id>)* **:** <tipo>

O programa E1 declara variáveis sem o Token VAR, conforme representado abaixo:

Comando:

```
$ java Main -p -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Parsing/E1.mp
```

Programa P1.mp:

```
!!
-----------------------------------------------------------------------
        THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL.
-----------------------------------------------------------------------
PROJECT TITLE: LEXICAL TESTER
PURPOSE OF PROJECT: USE ALMOST ALL TOKENS OF MINI-PASCAL LANGUAGE
VERSION or DATE: 16/03/2019
HOW TO START THIS PROJECT:
AUTHORS:    GUSTAVO MARQUES ( @GUTODISSE )
USER INSTRUCTIONS:
!

program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
var Num1, Num2, Num3, Sum : integer;
    Nums : array[ 1 .. 3 ] of real;
    testes : boolean;
begin
        Num1 := 1;  ! TEST COMMENT
        Num3 := 1;
        Num3 := .1;
        testes := true;
```

```
        Sum  := Num1 + Num2; ! TEST COMMENT

        if Num1 <= 2 then testes := false;
        while testes do testes := false;

        begin
        !! MULTILINE COMMENT
        WORKING
        !
        end;

end.
```

Output:

```
Checking Syntax
   [SYNTACTIC ERROR]
         EXPECTED:    begin
         FOUND:         <IDENTIFIER>

   E1.mp:15-5: error:   unexpected
 Nums : array[ 1 .. 3 ] of real;
        ^
```

O erro que o programa E2 contém está na regra:

<variável> **:=** **<expressão>**

O programa E2 utiliza o comando de maneira incompleta, conforme representado abaixo:

Comando:

```
$ java Main -p -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Parsing/E2.mp
```

Programa P1.mp:

```
!!
------------------------------------------------------------------------
        THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL.
------------------------------------------------------------------------
PROJECT TITLE: LEXICAL TESTER
PURPOSE OF PROJECT: USE ALMOST ALL TOKENS OF MINI-PASCAL LANGUAGE
VERSION or DATE: 16/03/2019
HOW TO START THIS PROJECT:
AUTHORS:     GUSTAVO MARQUES ( @GUTODISSE )
USER INSTRUCTIONS:
!
```

```
program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
begin
        Num1 := 1;  ! TEST COMMENT
        Num3 := ;

end.
```

Output:

```
Checking Syntax
   [SYNTACTIC ERROR]
         EXPECTED:    FATOR
         FOUND:         <SEMICOLON>

   E2.mp:16-13: error:
 unexpected
        Num3 := ;
                 ^
```

Anexo C

Para executar até o analisador contextual, basta acrescentar o comando “-c”;

Comando:
java Main -c -i <endereço do código fonte>

A seguir diversos programas (input) são analisados pela classe Checker, e a seguir a resposta da classe (output).

Sem erro(s):

Comando:
$ java Main -c -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Context/P1.mp
Programa P1.mp:

!!
--- 	
THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL.
--- 	
PROJECT TITLE: LEXICAL TESTER
PURPOSE OF PROJECT: TEST CONTEXTUAL IDENTIFIER
VERSION or DATE: 17/03/2019
HOW TO START THIS PROJECT:
AUTHORS: GUSTAVO MARQUES (@GUTODISSE)
USER INSTRUCTIONS:
!
 	
program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
  var Num1, Num2, Num3, Sum : integer;
  var Nums : array[1 .. 3] of real;
  var testes : boolean;
begin
 	
end.

Output:

Checking Syntax OK
Checking Context:
  Checking Declarations OK
  Checking Commands OK
Context OK

Com erro(s):

Regra quebrada: Nenhum identificador pode ser declarado mais de uma vez;

Comando:

```
$ java  Main -c -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Context/E2.mp
```

Programa E2.mp:

```
!!
-----------------------------------------------------------------------
   THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL.
-----------------------------------------------------------------------
PROJECT TITLE: LEXICAL TESTER
PURPOSE OF PROJECT: TEST CONTEXTUAL IDENTIFIER
VERSION or DATE: 17/03/2019
HOW TO START THIS PROJECT:
AUTHORS:    GUSTAVO MARQUES ( @GUTODISSE )
USER INSTRUCTIONS:
!

program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
        var Num1, Num1 : integer;
begin
          Num1 := 1;
end.
```

Output:

```
Checking Syntax                         OK
Checking Context:
        Checking Declarations
        [CONTEXT ERROR]
        Variable already declared! - Num1
```

Regra quebrada: Para cada aplicação de um identificador P, deve existir uma declaração correspondente P.

Comando:

```
$ java  Main -c -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Context/E1.mp
```

Programa E1.mp:

```
!!
-----------------------------------------------------------------------
   THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL.
```

```
----------------------------------------------------------------------
PROJECT TITLE: LEXICAL TESTER
PURPOSE OF PROJECT: TEST CONTEXTUAL IDENTIFIER
VERSION or DATE: 17/03/2019
HOW TO START THIS PROJECT:
AUTHORS:    GUSTAVO MARQUES ( @GUTODISSE )
USER INSTRUCTIONS:
!

program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
        var Num1, Sum : integer;
begin
          Sum := Num1 + Num2;
end.
```

Output:

```
Checking Syntax                 OK
Checking Context:
        Checking Declarations   OK
        Checking Commands
        [CONTEXT ERROR]
        Variable not declared! - Num2
```

Anexo D

Para executar até o analisador contextual, basta acrescentar o comando “-c”;

Comando:
java Main -c -i <endereço do código fonte>

A seguir diversos programas (input) são analisados pela classe Checker, e a seguir a resposta da classe (output).

Sem erro(s):

Comando:
$ java Main -c -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Context/P3.mp
Programa P3.mp:

```
!!
-----------------------------------------------------------------------
    THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL.
-----------------------------------------------------------------------
PROJECT TITLE: LEXICAL TESTER
PURPOSE OF PROJECT: TEST CONTEXTUAL
VERSION or DATE: 17/03/2019
HOW TO START THIS PROJECT:
AUTHORS:    GUSTAVO MARQUES ( @GUTODISSE )
USER INSTRUCTIONS:
!

program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
        var Int1, Int2, Int3, Sum : integer;
        var Bool : boolean;
        var Float : real;
begin
        Int1 := 1;
        Int2 := Int1 + (Int2 / 2) + ( Int3 * 10);

        Bool := (Int1 >= Int2);

        Float := ( 10.1 * 10) + Int1;

        if Int1 <= Int2 then Bool := false;

        if Bool then Bool := false else Bool := true and true;
```

```
        while (true) do
        begin
                Bool := false;
        end;


        begin
                if Int1 <= 2 then
                        begin
                           Bool := false;
                        end ;

                Int1 := 1 + 2 + 3 * ( 3 + 4 * ( 2 / 2));
                Float  := 1 + 1.1 + 2 + .2 + Float + Float;
        end;
end.
```

Output:

```
Checking Syntax                 OK
Checking Context:
        Checking Declarations   OK
        Checking Commands OK
Context                         OK
```

Com erro(s):

Regra quebrada: Usando <expressão> de tipo incompatível com comando while (O mesmo é válido para o comando If)

Comando:

```
$ java  Main -c -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Context/P1.mp
```

Programa E3.mp:

```
program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
        var Int1, Int2, Int3, Sum : integer;
        var Bool : boolean;
        var Float : real;
begin
        Int1 := 1;

        while (Int1) do
        begin
                Bool := false;
        end;
end.
```

Output:

```
Checking Syntax                 OK
Checking Context:
        Checking Declarations   OK
        Checking Commands
        [CONTEXT ERROR]
                The line 23 contain a context error.
                The type of Expression is Integer but the compiler expect Bolean.
```

Regra quebrada: Comparando dois tipos incompatíveis:

Comando:

```
$ java  Main -c -i /home/lampiao/Projects/MiniPascal_Compiler/Programs/Context/E4.mp
```

Programa E3.mp:

```
!!
-----------------------------------------------------------------------
   THIS IS A PROGRAM TO TEST THE LEXICAL GRAMMAR OF MINI-PASCAL.
-----------------------------------------------------------------------
PROJECT TITLE: LEXICAL TESTER
PURPOSE OF PROJECT: TEST CONTEXTUAL IDENTIFIER
VERSION or DATE: 17/03/2019
HOW TO START THIS PROJECT:
AUTHORS:    GUSTAVO MARQUES ( @GUTODISSE )
USER INSTRUCTIONS:
!

program Lesson1Program3; ! THIS IS A ONLY LINE COMMENT
        var Int1, Int2, Int3, Sum : integer;
        var Bool : boolean;
        var Float : real;
begin
        Int1 := (10 < Bool) ;

end.
```

Output:

```
Checking Syntax                 OK
Checking Context:
        Checking Declarations   OK
        Checking Commands
        [CONTEXT ERROR]
                The line 18 contain a context error.
                The type of Variable are Integer and the value are ERROR
```