



FUNDAÇÃO UNIVERSIDADE FEDERAL DO VALE DO SÃO FRANCISCO
CAMPUS JUAZEIRO
CURSO DE ENGENHARIA DA COMPUTAÇÃO

Edjair Aguiar Gomes Filho
Mateus Amorim Silva

**Documentação do compilador de Mini-Pascal para
máquina-objeto TAM**

Juazeiro, BA
2020

Edjair Aguiar Gomes Filho

Mateus Amorim Silva

**Documentação do compilador de Mini-Pascal para máquina-objeto
TAM**

Documento requerido
pelo professor Marcus Ramos,
como parte das exigências de
avaliação da disciplina
Compiladores, do curso de
Engenharia da Computação,
da Universidade do Vale do
São Francisco.

Juazeiro, BA

2020

Sumário

1. Linguagem-fonte.....	4
2. Descrição geral da arquitetura do compilador.....	11
3. Fundamentação teórica e técnicas empregadas na análise sintática	12
4. Análise léxica	15
4.1. Relação de tokens	15
4.2. Gramática Léxica	15
4.3. Manipulações efetuadas na gramática manipulada e verificação da condição LL(1)	15
4.4. Técnicas utilizadas e visão geral do funcionamento	18
4.5. Exemplos de entradas e saídas.....	23
5. Análise sintática	24
6. Erros gerados no compilador	38
7. Montagem e impressão da árvore de sintaxe abstrata:	42
8. Análise de contexto	53
9. Ambiente de execução	69
10. Linguagem-objeto	71
11. Geração de código	72
12. Manual de compilação	81
13. Instruções De execução	83
14. Mensagens de erro.....	88
15. Conclusões Finais.....	92

1. Linguagem-fonte

1.1. Sintaxe livre de contexto

$\langle \text{atribuição} \rangle ::= \langle \text{variável} \rangle := \langle \text{expressão} \rangle$

$\langle \text{bool-lit} \rangle ::= \text{true} \mid \text{false}$

$\langle \text{comando} \rangle ::= \langle \text{atribuição} \rangle \mid \langle \text{condicional} \rangle \mid \langle \text{iterativo} \rangle \mid \langle \text{comando-composto} \rangle$

$\langle \text{comando-composto} \rangle ::= \text{begin } \langle \text{lista-de-comandos} \rangle \text{ end}$

$\langle \text{condicional} \rangle ::= \text{if } \langle \text{expressão} \rangle \text{ then } \langle \text{comando} \rangle \text{ (else } \langle \text{comando} \rangle \mid \langle \text{vazio} \rangle)$

$\langle \text{corpo} \rangle ::= \langle \text{declarações} \rangle \langle \text{comando-composto} \rangle$

$\langle \text{declaração} \rangle ::= \langle \text{declaração-de-variável} \rangle$

$\langle \text{declaração-de-variável} \rangle ::= \text{var } \langle \text{lista-de-ids} \rangle : \langle \text{tipo} \rangle$

$\langle \text{declarações} \rangle ::= \langle \text{declaração} \rangle ; \mid \langle \text{declarações} \rangle \langle \text{declaração} \rangle ; \mid \langle \text{vazio} \rangle$

$\langle \text{digito} \rangle ::= 0 \mid 1 \mid 2 \mid \dots \mid 9$

$\langle \text{expressão} \rangle ::= \langle \text{expressão-simples} \rangle \mid \langle \text{expressão-simples} \rangle \langle \text{op-rel} \rangle \langle \text{expressão-simples} \rangle$

$\langle \text{expressão-simples} \rangle ::= \langle \text{expressão-simples} \rangle \langle \text{op-ad} \rangle \langle \text{termo} \rangle \mid \langle \text{termo} \rangle$

$\langle \text{fator} \rangle ::= \langle \text{variável} \rangle \mid \langle \text{literal} \rangle \mid "(" \langle \text{expressão} \rangle ")"$

$\langle \text{float-lit} \rangle ::= \langle \text{int-lit} \rangle . \langle \text{int-lit} \rangle \mid \langle \text{int-lit} \rangle . \mid . \langle \text{int-lit} \rangle$

$\langle \text{id} \rangle ::= \langle \text{letra} \rangle \mid \langle \text{id} \rangle \langle \text{letra} \rangle \mid \langle \text{id} \rangle \langle \text{digito} \rangle$

$\langle \text{int-lit} \rangle ::= \langle \text{digito} \rangle \mid \langle \text{int-lit} \rangle \langle \text{digito} \rangle$

$\langle \text{iterativo} \rangle ::= \text{while } \langle \text{expressão} \rangle \text{ do } \langle \text{comando} \rangle$

$\langle \text{letra} \rangle ::= a \mid b \mid c \mid \dots \mid z$

$\langle \text{lista-de-comandos} \rangle ::= \langle \text{comando} \rangle ; \mid \langle \text{lista-de-comandos} \rangle \langle \text{comando} \rangle ; \mid \langle \text{vazio} \rangle$

$\langle \text{lista-de-ids} \rangle ::= \langle \text{id} \rangle \mid \langle \text{lista-de-ids} \rangle , \langle \text{id} \rangle$

$\langle \text{literal} \rangle ::= \langle \text{bool-lit} \rangle \mid \langle \text{int-lit} \rangle \mid \langle \text{float-lit} \rangle$
 $\langle \text{op-ad} \rangle ::= + \mid - \mid \text{or}$
 $\langle \text{op-mul} \rangle ::= * \mid / \mid \text{and}$
 $\langle \text{op-rel} \rangle ::= < \mid > \mid \leq \mid \geq \mid = \mid \neq$
 $\langle \text{outros} \rangle ::= ! \mid @ \mid \# \mid \dots$
 $\langle \text{programa} \rangle ::= \text{program } \langle \text{id} \rangle ; \langle \text{corpo} \rangle .$
 $\langle \text{seletor} \rangle ::= \langle \text{seletor} \rangle "[\langle \text{expressão} \rangle]" \mid "[\langle \text{expressão} \rangle]" \mid \langle \text{vazio} \rangle$
 $\langle \text{termo} \rangle ::= \langle \text{termo} \rangle \langle \text{op-mul} \rangle \langle \text{fator} \rangle \mid \langle \text{fator} \rangle$
 $\langle \text{tipo} \rangle ::= \langle \text{tipo-agregado} \rangle \mid \langle \text{tipo-simples} \rangle$
 $\langle \text{tipo-agregado} \rangle ::= \text{array } [\langle \text{literal} \rangle .. \langle \text{literal} \rangle] \text{ of } \langle \text{tipo} \rangle$
 $\langle \text{tipo-simples} \rangle ::= \text{integer} \mid \text{real} \mid \text{boolean}$
 $\langle \text{variável} \rangle ::= \langle \text{id} \rangle \langle \text{seletor} \rangle$
 $\langle \text{vazio} \rangle ::= \epsilon$

1.2. Semântica

1.2.1. Comando Atribuição

$\langle \text{variável} \rangle := \langle \text{expressão} \rangle$

A variável recebe o conteúdo da expressão, desde que ambos possuam o mesmo tipo.

1.2.2. Comando condicional

$\text{if } \langle \text{expressão} \rangle \text{ then } \langle \text{comando} \rangle (\text{ else } \langle \text{comando} \rangle \mid \langle \text{vazio} \rangle)$

Caso a $\langle \text{expressão} \rangle$ seja do tipo $\langle \text{bool-lit} \rangle$, executa $\langle \text{comando} \rangle$ se a expressão for verdadeira, caso a expressão seja falsa e exista o “else” após o primeiro $\langle \text{comando} \rangle$, executa o $\langle \text{comando} \rangle$ após ele.

1.2.3. Comando iterativo

$\text{while } \langle \text{expressão} \rangle \text{ do } \langle \text{comando} \rangle$

Dada uma $\langle \text{expressão} \rangle$ do tipo $\langle \text{bool-lit} \rangle$, executa o $\langle \text{comando} \rangle$ enquanto esta for verdadeira.

1.2.4. Seletores

A expressão deve ter como retorno o tipo <int-lit>.

1.2.5. Variáveis

Variáveis que se deseja utilizar no corpo do programa devem ser previamente declaradas e utilizando-se apenas letras minúsculas.

Para variáveis do tipo agregado, temos:

array [<int-lit> ~ <int-lit>] of <tipo>

Foi realizada uma alteração na relação de tokens da gramática, o token “double dot” (..) foi alterado pelo token til (~) de forma a simplificar a análise léxica no que diz respeito à análise do token ponto (.). Isso foi feito visando também o determinismo da gramática a ser utilizada.

Apesar de ser <int-lit> os índices do array devem ser sempre positivos.

O primeiro <int-lit> deve ser menor ou igual que o segundo <int-lit> e o número de elementos do array é igual a $(\text{<int-lit>}^2 - \text{<int-lit>}^1 + 1)$ onde, <int-lit>^1 e <int-lit>^2 são o primeiro e segundo literal respectivamente.

Ao ser utilizado no corpo do programa o número de dimensão deve ser igual ao declarado, o valor do <int-lit> inserido deve ser maior ou igual ao <int-lit>^1 e menor ou igual ao <int-lit>^2 . Considere um array de duas dimensões, chamado “matrizQuadrada”, declarado da seguinte forma:

var matrizQuadrada : array [1 .. 6] of array [1 .. 6] of integer;

Caso esse array seja acessado como matrizQuadrada[4], ou matrizQuadrada[1][6][5], o acesso ocasionará erro, devido ao número de dimensões diferente do declarado.

Caso seja acessado como `matrizQuadrada[1][7]` ou `matrizQuadrada[0][6]`, ocasionará erro devido ao `<int-lit>` não estar na faixa mencionada anteriormente. Utilizações como `matrizQuadrada[1][6]` ou `matrizQuadrada[2][3]` são válidas.

1.2.6. Operadores relacionais

Os operadores relacionais são binários e retornam um `<bool-lit>` e pertencem a seguinte linguagem:

1.2.6.0. “<”

`< : int-lit x int-lit -> <bool-lit>.`

`< : float-lit x float-lit -> <bool-lit>.`

Verifica se o elemento a esquerda é menor que o da direita, caso verdadeiro, retorna `true`, caso contrário, `false`.

1.2.6.1. “>”

`>: int-lit x int-lit -> <bool-lit>.`

`> : float-lit x float-lit -> <bool-lit>.`

Verifica se o elemento a esquerda é maior que o da direita, caso verdadeiro, retorna `true`, caso contrário, `false`.

1.2.6.2. “<=”

`<= : int-lit x int-lit -> <bool-lit>.`

`<= : float-lit x float-lit -> <bool-lit>.`

Verifica se o elemento à esquerda é menor ou igual ao da direita, caso seja, retorna `true`, caso contrário, `false`.

1.2.6.3. “>=”

$\geq : \text{int-lit} \times \text{int-lit} \rightarrow \langle \text{bool-lit} \rangle.$

$\geq : \text{float-lit} \times \text{float-lit} \rightarrow \langle \text{bool-lit} \rangle.$

Verifica se o elemento a esquerda é maior ou igual ao da direita, caso seja, retorna true, caso contrário, false.

1.2.6.4. “=”

$= : \text{int} \times \text{int} \rightarrow \langle \text{bool-lit} \rangle.$

$= : \text{float} \times \text{float} \rightarrow \langle \text{bool-lit} \rangle.$

Verifica se o elemento a esquerda é igual ao da direita, caso seja, retorna true, caso contrário, false.

1.2.6.5. “<>”

$\neq : \text{int-lit} \times \text{int-lit} \rightarrow \langle \text{bool-lit} \rangle.$

$\neq : \text{float-lit} \times \text{float-lit} \rightarrow \langle \text{bool-lit} \rangle.$

Verifica se o elemento a esquerda é diferente ao da direita, caso seja, retorna true, caso contrário, false.

1.2.7. Operadores lógicos

Os operadores lógicos são binários e retornam um $\langle \text{bool-lit} \rangle$, são eles:

1.2.7.0. “and”

$\text{and} : \langle \text{bool-lit} \rangle \times \langle \text{bool-lit} \rangle \rightarrow \langle \text{bool-lit} \rangle$

Realiza a operação “e” entre os dois operandos e retorna o resultado equivalente.

1.2.7.1. “or”

or : <bool-lit> x <bool-lit> -> <bool-lit>

Realiza a operação “ou” entre os dois operandos e retorna o resultado equivalente.

1.2.8. Operadores aritméticos

Os operadores aritméticos são binários, cujo retorno depende dos operandos, e pertencem a seguinte linguagem:

1.2.8.0. “+”

+ : int-lit x int-lit -> int-lit.

+ : float-lit x float-lit -> float-lit.

Realiza a operação de soma entre os dois operandos e retorna o resultado equivalente.

1.2.8.1. “-”

- : int-lit x int-lit -> int-lit.

- : float-lit x float-lit -> float-lit.

Realiza a operação de subtração entre os dois operandos e retorna o resultado equivalente.

1.2.8.2. “*”

* : int-lit x int-lit -> int-lit.

* : float-lit x float-lit -> float-lit.

Realiza a operação de multiplicação entre os dois operandos e retorna o resultado equivalente.

1.2.8.3. “/”

/ : int-lit x int-lit -> int-lit.

/ : float-lit x float-lit -> float-lit.

Realiza a operação de divisão entre os dois operandos, sendo o segundo diferente de 0, e retorna o resultado equivalente.

A divisão de um int-lit por outro int-lit pode ocasionar um float-lit. Esse tipo de atribuição é tratada no analisador de contexto.

1.2.8.4. Ordem de precedência das operações

Ordem	Operação	Símbolo
1ª	Parênteses	“(”, “)”
2ª	Multiplicação, divisão.	*, /
3ª	Adição, subtração	+, -
4ª	Operadores relacionais.	>, <, >=, <=
5ª	Ou, E	and, or

Tabela 01: Ordem de precedência das operações do compilador

2. Descrição geral da arquitetura do compilador

O compilador descrito nesta documentação foi desenvolvido em Java, compilado pelo javac e interpretado pelo Java Virtual Machine (JVM). Tem como linguagem fonte o Mini-Pascal e TAM como linguagem objeto.

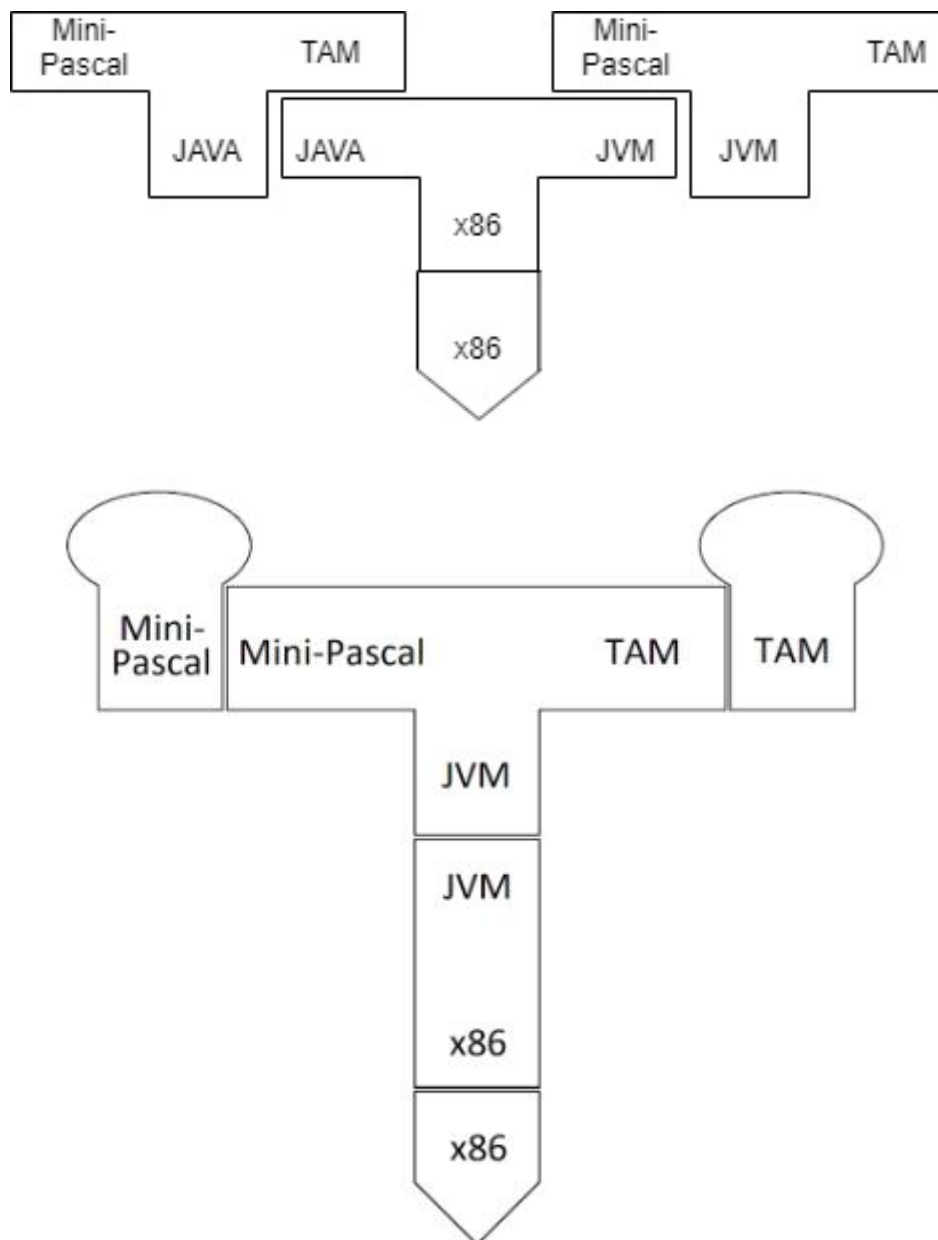


Figura 01: Diagrama T do compilador Mini-Pascal para TAM

O processo de compilação é dividido em três fases principais: análise sintática, análise contextual e geração de código.

O analisador sintático é decomposto na construção do parser, scanner e na árvore de sintaxe abstrata, que são sistematicamente construídos a partir das especificações sintáticas da linguagem fonte. Na análise contextual os tópicos principais foram a identificação, relacionada às regras de escopo da linguagem, e a checagem de tipos, relacionada com as regras de tipo da linguagem. Na geração de código a linguagem fonte é traduzida para a linguagem objeto de forma semanticamente equivalente. Para isso foi criada uma classe chamada *Coder*, que implementa o padrão de projeto *Visitor*.

3. Fundamentação teórica e técnicas empregadas na análise sintática

3.1. Fundamentação teórica

A análise sintática é dividida em três subfases (já mencionadas acima) e tem como principal objetivo analisar o programa fonte para descobrir sua estrutura frasal. O analisador sintático recebe do analisador léxico uma cadeia de tokens representando o programa fonte e verifica se a mesma pertence à linguagem gerada pela gramática. Por ser um compilador multi-passo, a estrutura de frase do programa fonte deve ser representada de forma explícita, essa representação é convenientemente e amplamente feita através de uma árvore de sintaxe abstrata (AST).

Os métodos de análise sintática para o processamento de gramáticas em compiladores geralmente são baseados em estratégias descendente ou ascendente. Os métodos de análise descendente constroem as árvores de derivação de cima (raiz) para baixo (folhas), enquanto os métodos ascendentes fazem a análise no sentido inverso. Em ambas as estratégias, a entrada do analisador sintático é consumida da esquerda para a direita, um símbolo de cada vez. Cada sentença é uma cadeia de símbolos terminais e tem uma estrutura frasal única, incorporada em uma árvore sintática.

3.2. Técnicas empregadas

3.2.1. EBNF

O Formalismo Estendido de Backus-Naur (EBNF, do inglês Backus-Naur Form ou Backus Normal Form) é uma extensão do BNF, criada para ampliar a facilidade de leitura e concisão de suas produções e é uma família de notações meta-sintaxe usada para expressar gramáticas livres de contexto, isto é, um modo formal de escrever linguagens formais. Os meta-símbolos utilizados na notação BNF são:

- $::=$ – representa “definido como”;
- $|$ – indica uma alternativa (como o OU lógico);
- $< >$ – indica uma regra

Uma expressão regular (RE) é uma notação conveniente para expressar um conjunto de sentenças de símbolos terminais de maneira organizada e fácil de interpretar. A notação EBNF combina as vantagens do BNF com as expressões regulares (RE), estende a notação com os seguinte meta-símbolos:

- $[]$ – indica uma parte opcional;
- $\{ \}$ – indica uma parte que se pode repetir 0 ou mais vezes;
- $()$ – indica precedências dentro da regra (agrupa);
- $" "$ – indica um caractere a tratar como terminal

Com a adição dos meta-símbolos do formalismo EBNF se torna possível a eliminação de regras e de recursão à direita, fato este que tem importância na construção da gramática livre de contexto, que descreve sistematicamente a sintaxe das construções de linguagem de programação, como expressões e comandos.

3.2.2. Abordagem derivativa

A abordagem derivativa é utilizada na construção da árvore de derivação, onde a partir de um símbolo inicial, a cada passo de reescrita substitui-se um não-terminal pelo corpo de suas produções.

Em função disso, foi criada a classe Parser, que é responsável por criar a árvore de sintaxe abstrata (AST) e informar erros de sintaxe. Para cada símbolo não-terminal N um método parseN() é implementado na classe Parser.

```
private Condicional parseCondicional() throws Exception{
    //Condicional == if <Expressão> then <Comando> (else <Comando> | <Vazio>)
    Condicional conditional = new Condicional();
    accept(Token.IF);
    conditional.expression = parseExpressao();
    accept(Token.THEN);
    conditional.command = parseComando();
    if(currentToken.kind == Token.ELSE){
        acceptIt();
        conditional.commandElse = parseComando();
        //commandElse: tratamento de caso para ELSE; tratado na AST para evitar else sem nada após
    } else{
        conditional.commandElse = null;
    }
    return conditional;
}
```

Figura 02: parse para o comando condicional

3.2.2.1. Derivações à esquerda

Uma sequência de derivações é dita “mais à esquerda” (leftmost) quando todas as derivações são feitas sempre sobre os símbolos não-terminais situados mais à esquerda na forma sentencial corrente.

3.2.3. Gramática LL(1)

Os analisadores sintáticos preditivos, ou seja, os analisadores que usam o método recursivo descendente, podem ser construídos para uma classe de gramáticas chamadas LL(1), podendo assim garantir a correção e a funcionalidade do programa resultante. O primeiro “L” em LL(1) significa que a cadeia de entrada é examinada da esquerda para a direita (Left-to-right); o segundo “L” representa uma derivação mais à esquerda (Leftmost); e o “1” pelo uso de um símbolo à frente na entrada utilizado em cada passo para tornar as decisões quanto à ação de análise. Uma gramática que atende a condição LL(1), gera uma linguagem onde suas sentenças podem ser analisadas de forma descendente (ou seja, por meio da ordem direta das derivações mais à esquerda) com o look-ahead de apenas um símbolo.

Isto significa, então, que é possível construir um analisador determinístico e muito simples para a linguagem.

4. Análise léxica

4.1. Relação de tokens

:=	true	false	begin	end	if	then	else]	!
var	:	;	int-lit	()	.	id	array	float-lit
while	do	,	+	-	or	*	/	~	error
and	<	>	<=	>=	=	<>	[of	

Obs: Foi adicionado o token “error” a linguagem para ser realizado o tratamento de erros na linguagem os quais serão explicitados mais à frente.

4.2. Gramática Léxica

Token ::= := true | false | begin | end | if | then | else | var | : | ; | <int-lit> | <float-lit> | (|) | . | <id> | while | do | , | + | - | or | * | / | and | < | > | <= | >= | = | <> | [|] | array | ~ | of | error

<int-lit> ::= <digito>⁺

<float-lit> ::= <digito>⁺ . <digito>^{*} | . <digito>⁺ | <digito>⁺.

<id> ::= <letra>(<letra> | <digito>)*

<letra> ::= a | b | c | ... | z

<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

4.3. Manipulações efetuadas na gramática manipulada e verificação da condição LL(1)

Token:

A manipulação realizada nesta regra foi a adição das regras <id>, <int-lit> e <float-lit>.

$\text{first}_1(\text{<id>}) = \{\text{<letra>}\}$

Que equivale a:

a | b | c | ... | z

$\text{first}_1(\langle \text{int-lit} \rangle) = \{ \langle \text{digito} \rangle \}$

Que equivale a:

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

$\text{first}_1(\langle \text{float-lit} \rangle) = \{ \langle \text{digito} \rangle \}$

.

Que equivale a:

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | .

Como os demais tokens são terminais, o first_1 são os próprios tokens exceto ($\langle \text{id} \rangle$, $\langle \text{int-lit} \rangle$ e $\langle \text{float-lit} \rangle$), como mostrado abaixo

$:=$ | true | false | begin | end | if | then | else | var | : | ; | (|) | . | while | do | , | + | - | or
| * | / | and | < | > | <= | >= | = | <> | [|] | array | .. | of

Como a interseção entre os firsts é vazia, temos que é LL(1).

$\langle \text{int-lit} \rangle$

Esta regra produz termos como:

$\langle \text{digito} \rangle$

$\langle \text{digito} \rangle \langle \text{digito} \rangle$

$\langle \text{digito} \rangle \langle \text{digito} \rangle \langle \text{digito} \rangle$

Ou seja, um ou mais $\langle \text{digito} \rangle$, logo, pode ser reescrita como:

$\langle \text{int-lit} \rangle ::= \langle \text{digito} \rangle (\langle \text{digito} \rangle)^*$

Eliminando a recursão a esquerda.

$\text{first}(1)$

$\langle \text{digito} \rangle$ logo é

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Como a interseção entre os firsts é vazia, temos que é LL(1).

<id>

$\langle id \rangle ::= \langle letra \rangle \mid \langle id \rangle \langle letra \rangle \mid \langle id \rangle \langle digito \rangle$

Exemplos de produções são:

$\langle letra \rangle \langle digito \rangle \langle digito \rangle$

$\langle letra \rangle \langle letra \rangle \langle letra \rangle$

$\langle letra \rangle \langle digito \rangle letra$

$\langle letra \rangle \langle letra \rangle \langle digito \rangle$

Ou seja, produz uma sequência de $\langle letra \rangle$ e $\langle digito \rangle$ onde a única restrição é que contenha $\langle letra \rangle$ no início, portanto pode ser reescrita como:

$\langle id \rangle ::= \langle letra \rangle (\langle letra \rangle \mid \langle digito \rangle)^*$

first(1)

$\langle letra \rangle$ logo é

$a \mid b \mid c \mid \dots \mid z$

Como a interseção entre os firsts é vazia, temos que é LL(1).

<float-lit>

$\langle float-lit \rangle ::= \langle int-lit \rangle . \langle int-lit \rangle \mid \langle int-lit \rangle . \mid . \langle int-lit \rangle$

Realizando fatoração à esquerda obtemos:

$\langle float-lit \rangle ::= \langle int-lit \rangle . (\langle int-li \rangle \mid \langle vazio \rangle) . \mid . \langle int-lit \rangle$

Substituindo $\langle int-lit \rangle$ por $\langle digito \rangle (\langle digito \rangle)^*$, como exemplificado anteriormente:

$\langle float-lit \rangle ::= \langle digito \rangle (\langle digito \rangle)^* . (\langle digito \rangle (\langle digito \rangle)^* \mid \langle vazio \rangle) \mid . \langle digito \rangle (\langle digito \rangle)^*$

Como $. (\langle digito \rangle (\langle digito \rangle)^* \mid \langle vazio \rangle)$ corresponde a $\langle digito \rangle^*$, logo temos:

$\langle float-lit \rangle ::= \langle digito \rangle \langle digito \rangle^* . \langle digito \rangle^* \mid . \langle digito \rangle \langle digito \rangle^*$

Eliminando uma regra, e facilitando a análise durante a fase de análise sintática do compilar.

first(1)

<digito>

.

logo é

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | .

Como a interseção entre os firsts é vazia, temos que é LL(1).

<letra>

first(1)

a | b | c | ... | z

Como a interseção entre os firsts é vazia, temos que é LL(1).

<digito>

first(1)

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Como a interseção entre os firsts é vazia, temos que é LL(1).

4.4. Técnicas utilizadas e visão geral do funcionamento

A análise léxica, onde o programa fonte é transformado em um fluxo de tokens, terá a missão de avaliar os tokens da linguagem avaliando-os e observando se eles são válidos ou inválidos. Torna-se necessário obter a relação sobre tokens desta linguagem, através da gramática léxica. Dessa forma, a gramática deve apresentar condição de linguagem LL(1). Vale ressaltar que comentários e espaços em branco entre tokens, cuja função é meramente informar algo ao leitor do código, são descartados. Na implementação da classe Token, o tipo é chamado de *kind* e a ortografia de *spelling*, como é mostrado na figura classe logo a baixo.

```
public class Token {  
  
    public byte kind;  
    public String value;  
    public int line;  
    public int col;  
}
```

Figura 03: Atributos da classe Token

Além disso, a classe Token é responsável por guardar a relação de tokens da linguagem e seus respectivos códigos identificadores.

```
public final static byte ID = 0, INT_LIT = 1, FLOAT_LIT = 2, SUM = 3, SUB = 4, MULT = 5,
    DIV = 6, GREATER = 7, GREATER_EQUAL = 8, LESS_EQUAL = 9, DIFF = 10,
    LESS = 11, LBRACKET = 12, RBRACKET = 13, SEMICOLON = 14,
    BECOMES = 15, COLON = 16, LPAREN = 17, RPAREN = 18,
    TILDE = 19, DOT = 20, COMMA = 21, BEGIN = 22, END = 23,
    IF = 24, THEN = 25, ELSE = 26, VAR = 27, WHILE = 28, DO = 29,
    OR = 30, AND = 31, PROGRAM = 32, ARRAY = 33, OF = 34, INTEGER = 35,
    REAL = 36, TRUE = 37, FALSE = 38, BOOLEAN = 39, EOF = 40, ERROR = 41, EQUAL = 42;

// <op-rel> ::= < | > | <= | >= | = | <>
public final static String[] SPELLINGS = {
    "<id>", "<int-lit>", "<float-lit>", "+", "-", "*", "/", ">", ">=", "<=", "<>", "<",
    "[", "]", ";", ":", "(", ")", "~", ".", ",", "begin", "end", "if",
    "then", "else", "var", "while", "do", "or", "and", "program", "array",
    "of", "integer", "real", "true", "false", "boolean", "<EOF>", "<ERRO>", "="
};
```

Figura 04: Relação de tokens e respectivos códigos identificadores da linguagem

A classe Scanner é responsável por toda a análise léxica. É nesta classe que é feita a leitura, identificação e armazenamento de cada token. Para a leitura e identificação foram usados métodos auxiliares, são eles, *take* e *takeIt*.

```
public class Scanner {

    final private BufferedReader file;
    private char currentChar;
    private int line;
    private int col, aux;
    private byte currentKind;
    private StringBuffer currentValue;

    public Scanner(String fileName) throws Exception {
        BufferedReader file = new BufferedReader(new FileReader(fileName));
        this.file = file;
        currentChar = (char) file.read(); // TO DO : pegar primeiro caractere do txt
        // System.out.println(currentChar);
        // System.out.println("valor corrente = " + currentValue);
        col = 0;
        line = 1;
    }
}
```

Figura 05: Classe Scanner e seu método construtor

No trecho de código acima vemos a declaração da classe principal do analisador léxico, onde temos o atributo *file*, que é responsável por fazer a leitura do arquivo fonte. O *BufferedReader* é um tipo específico para leitura de arquivos em Java. Os atributos *currentSpelling* e *currentKind* guardam a informação do token corrente a ser analisado, assim como o *currentChar* é utilizado para guardar o caractere corrente e o *currentValue* é responsável por associar o token ao seu valor. Além disso, *col* e *line* guardam informações a respeito da localização do cursor de leitura.

```

private void take(char expectedChar) throws Exception{
    if(currentChar == expectedChar){
        currentValue.append(currentChar);
        currentChar = (char)file.read(); //currentChar = proximo caractere;
        //System.out.println(currentChar);
    } else {
        //retorna token erro
    }
}

private void takeIt() throws Exception{
    currentValue.append(currentChar);
    currentChar = (char)file.read(); //currentChar = proximo caractere;
    //System.out.println(currentChar);

    col++;
}

```

Figura 06: Métodos take e takeIt da classe Scanner

Na figura 06 vemos dois dos métodos mais importantes para a leitura do programa fonte. No método take é feita leitura de cada caractere e a do mesmo ao currentValue, e logo depois passado para os próximos caracteres a ser lidos através do atributo file, por fim é feita a atualização da coluna do token.

Já no método takeIt, é realizada a mesma operação do método take, porém não é passado um caractere por parâmetro, esta função apenas consome o caractere e atualiza o atributo currentValue.

Abaixo vemos a verificação de caractere (char), se é letra, dígito ou gráfico.

```

private boolean isDigit(char c){
    return c >= 48 && c <= 57;
}

private boolean isLetter(char c){
    return c >= 97 && c <= 122;
}

private boolean isGraphic(char c){
    boolean x = (c >= 32 && c <= 125);
    boolean y = false;
    if(c == '\t'){
        y = true;
    }
    return x || y;
}

```

Figura 07: Métodos de verificação de caractere (dígito, letra ou gráfico)

Abaixo vemos a implementação de um importante método do analisador sintático, responsável por fazer a limpeza de espaços em branco, tabulação, quebra de linha e comentários. Foi escolhido o caractere “#” para representar os comentários nessa linguagem, logo os comentários deverão obrigatoriamente começar com “#”, e tudo que estiver após será desconsiderado, como é mostrado no switch, onde é verificado se o caractere corrente é “#”, assim é consumida toda a linha depois deste caractere.

```
private void scanSeparator() throws Exception{ //Revisar simbolos
    switch(currentChar){
        case '#':{ //marcação de linha de comentário assim como esse //
            takeIt();
            aux = col;
            while(isGraphic(currentChar)){
                takeIt();
            }
            take('\n');
            //line++;
            col=-1;
        }
        break;
        case '\n':
            line++;
            col=-1;
        case ' ':
        case 13:
        case 9:
            takeIt();
        break;
    }
}
```

Figura 08: Método responsável pela limpeza de caracteres não lidos pelo compilador

```

private byte scanToken() throws Exception{
    if(isLetter(currentChar)){
        takeIt();
        aux = col;
        while(isLetter(currentChar) || isDigit(currentChar)){ //<letra>(<letra> | <digito>)*
            takeIt();
        }
        return Token.ID;
    }
    if(isDigit(currentChar)){ //<digito><digito>*
        takeIt();
        aux = col;
        while(isDigit(currentChar)){
            takeIt();
        }
        if(currentChar == '.'){
            takeIt();
            if(isDigit(currentChar)){
                takeIt();
                while(isDigit(currentChar)){
                    takeIt();
                }
                return Token.FLOAT_LIT;
            } else{
                return Token.FLOAT_LIT;
            }
        }
    }
    return Token.INT_LIT;
}

```

Figura 09: Trecho do método scanToken

Na figura 09, o método principal da classe, o scanToken, sua função é retornar o token montado, ou seja com todas as informações necessárias para a realização da análise sintática daquele token. Vemos logo acima a implementação do scan, onde é chamado o método scanToken, que é responsável por fazer a chamadas dos métodos auxiliares já mostrados e retornar o código do token, este código é atribuído ao currentKind. Logo depois é retornado um novo token já com todas as informações preenchidas, como mostrado no trecho final da figura acima.

4.5. Exemplos de entradas e saídas

4.5.1. Exemplos de cadeias malformadas e respectivas respostas

```
program loona;  
  var loona: array[1 ~ 3] of boolean;  
  var x: real;  
  var chuu: real;  
  var y: integer;  
  
begin  
  x := 15 % 3;  
end.
```

Saída:

[!][1] ERRO SINTÁTICO (Erro 1.1)

Token inesperado.
| Esperava encontrar: ';'
| Na linha: 8
| Coluna: 7

[!][2] ERRO SINTÁTICO (Erro 1.1)

Token inesperado.
| Esperava encontrar: 'end'
| Na linha: 8
| Coluna: 7

[!][3] ERRO SINTÁTICO (Erro 1.1)

Token inesperado.
| Esperava encontrar: '.'
| Na linha: 8
| Coluna: 7

[!][4] ERRO SINTÁTICO (Erro 1.1)

Token inesperado.
| Esperava encontrar: '<EOF>'
| Na linha: 8
| Coluna: 7

Como a linguagem utilizada neste compilador não possui o símbolo de “%” como operador, o compilador finaliza a execução e mostra as informações sobre o erro, como mostrado no exemplo acima.

4.5.2. Exemplos de cadeias bem formadas e respectiva resposta

```
program loona;  
  var loona: array[1 ~ 3] of boolean;  
  var x: real;  
  var chuu: real;  
  var y: integer;  
  
begin  
  x := 15 + 3;  
end.
```

No exemplo acima, temos um exemplo muito parecido com o exemplo de cadeias mal formadas, a única modificação é que o operador '%' foi substituído por '+', tornando assim uma cadeia válida

5. Análise sintática

5.1. Gramática sintática modificada

<atribuição> ::= <variável> := <expressão>

<bool-lit> ::= true | false

<comando> ::= <atribuição> | <condicional> | <iterativo> | <comando-composto>

<comando-composto> ::= begin <lista-de-comandos> end

<condicional> ::= if <expressão> then <comando> (else <comando> | <vazio>)

<corpo> ::= <declarações> <comando-composto>

<declaração-de-variável> ::= var <lista-de-ids> : <tipo>

<declarações> ::= (<declaração-de-variável> ;)*

<expressão> ::= <expressão-simples> | <expressão-simples> <op-rel> <expressão-simples>

$\langle \text{expressão} \rangle ::= \langle \text{expressão-simples} \rangle (\langle \text{op-rel} \rangle \langle \text{expressão-simples} \rangle \mid \langle \text{vazio} \rangle)$

$\langle \text{expressão-simples} \rangle ::= \langle \text{termo} \rangle (\langle \text{op-ad} \rangle \langle \text{termo} \rangle)^*$

$\langle \text{fator} \rangle ::= \langle \text{variável} \rangle \mid \langle \text{literal} \rangle \mid "(" \langle \text{expressão} \rangle ")"$

$\langle \text{iterativo} \rangle ::= \text{while } \langle \text{expressão} \rangle \text{ do } \langle \text{comando} \rangle$

$\langle \text{letra} \rangle ::= a \mid b \mid c \mid \dots \mid z$

$\langle \text{lista-de-comandos} \rangle ::= (\langle \text{comando} \rangle ;)^*$

$\langle \text{lista-de-ids} \rangle ::= \langle \text{id} \rangle (, \langle \text{id} \rangle)^*$

$\langle \text{literal} \rangle ::= \langle \text{bool-lit} \rangle \mid \langle \text{int-lit} \rangle \mid \langle \text{float-lit} \rangle$

$\langle \text{op-ad} \rangle ::= + \mid - \mid \text{or}$

$\langle \text{op-mul} \rangle ::= * \mid / \mid \text{and}$

$\langle \text{op-rel} \rangle ::= < \mid > \mid \leq \mid \geq \mid = \mid <>$

$\langle \text{outros} \rangle ::= ! \mid @ \mid \# \mid \dots$

$\langle \text{programa} \rangle ::= \text{program } \langle \text{id} \rangle ; \langle \text{corpo} \rangle .$

$\langle \text{seletor} \rangle ::= ("[" \langle \text{expressão} \rangle "]")^*$

$\langle \text{termo} \rangle ::= \langle \text{fator} \rangle (\langle \text{op-mul} \rangle \langle \text{fator} \rangle)^*$

$\langle \text{tipo} \rangle ::= \langle \text{tipo-agregado} \rangle \mid \langle \text{tipo-simples} \rangle$

$\langle \text{tipo-agregado} \rangle ::= \text{array } [\langle \text{literal} \rangle \dots \langle \text{literal} \rangle] \text{ of } \langle \text{tipo} \rangle$

$\langle \text{tipo-simples} \rangle ::= \text{integer} \mid \text{real} \mid \text{boolean}$

$\langle \text{variável} \rangle ::= \langle \text{id} \rangle \langle \text{seletor} \rangle$

$\langle \text{vazio} \rangle ::= \epsilon$

5.2. Verificação da condição LL(1) na gramática manipulada e manipulações efetuadas na gramática original e justificativas

Segue-se abaixo a verificação da condição LL(1) para a gramática original e as modificações realizadas para obter a gramática LL(1) exibida no item 5.a.

5.2.1. $\langle \text{atribuição} \rangle ::= \langle \text{variável} \rangle := \langle \text{expressão} \rangle$

first(1):

$\langle \text{id} \rangle$

Logo, é LL(1).

5.2.2. $\langle \text{comando} \rangle ::= \langle \text{atribuição} \rangle \mid \langle \text{condicional} \rangle \mid \langle \text{iterativo} \rangle \mid \langle \text{comando-composto} \rangle$

first(1) de $\langle \text{atribuição} \rangle$:

$\langle \text{id} \rangle$

first(1) de $\langle \text{condicional} \rangle$:

if

first(1) de $\langle \text{iterativo} \rangle$:

while

first(1) de $\langle \text{comando-composto} \rangle$:

begin

Como a interseção entre os firsts é vazia, temos que é LL(1).

5.2.3. $\langle \text{comando-composto} \rangle ::= \text{begin } \langle \text{lista-de-comandos} \rangle$
 end

first(1):

begin

Logo, é LL(1).

5.2.4. $\langle \text{condicional} \rangle ::= \text{if } \langle \text{expressão} \rangle \text{ then } \langle \text{comando} \rangle ($
 $\text{else } \langle \text{comando} \rangle \mid \langle \text{vazio} \rangle)$

first(1):

if

Logo, é LL(1).

5.2.5. $(\text{else } \langle \text{comando} \rangle \mid \langle \text{vazio} \rangle)$

first(1):

else

follow(1):

“,”
;

Logo, é LL(1).

5.2.6. $\langle \text{corpo} \rangle ::= \langle \text{declarações} \rangle \langle \text{comando-composto} \rangle$

first(1):

var

Logo, é LL(1).

5.2.7. $\langle \text{declaração-de-variável} \rangle ::= \text{var } \langle \text{lista-de-ids} \rangle :$
 $\langle \text{tipo} \rangle$

first(1):

var

Logo, é LL(1).

5.2.8. $\langle \text{declarações} \rangle ::= \langle \text{declaração} \rangle ; \mid \langle \text{declarações} \rangle \langle \text{declaração} \rangle ; \mid \langle \text{vazio} \rangle$

Possui recursão a esquerda, portanto não é LL(1);

Como:

$\langle \text{declaração} \rangle ::= \langle \text{declaração-de-variável} \rangle$ (2)

Substituindo (2) em (1) :

$\langle \text{declarações} \rangle ::= \langle \text{declaração-de-variável} \rangle ; \mid \langle \text{declarações} \rangle \langle \text{declaração-de-variável} \rangle ; \mid \langle \text{vazio} \rangle$

Ao analisar a regra, percebe-se que ela produz nenhuma, ou várias vezes o não terminal $\langle \text{declaração-de-variável} \rangle$ seguido de “;”, logo, pode ser reescrita da seguinte maneira:

$\langle \text{declarações} \rangle ::= (\langle \text{declaração-de-variável} \rangle ;)^*$

Eliminando neste processo, a recursão a esquerda presente na regra original.

first(1):

var

Logo, é LL(1).

5.2.9. $\langle \text{expressão} \rangle ::= \langle \text{expressão-simples} \rangle \mid \langle \text{expressão-simples} \rangle \langle \text{op-rel} \rangle \langle \text{expressão-simples} \rangle$

Possui duas regras iniciadas por $\langle \text{expressão-simples} \rangle$, portanto não é LL(1), realizando fatoração a esquerda obtemos:

$\langle \text{expressão} \rangle ::= \langle \text{expressão-simples} \rangle (\langle \text{op-rel} \rangle \langle \text{expressão-simples} \rangle \mid \langle \text{vazio} \rangle)$

first(1):

$\langle \text{id} \rangle$
 $\langle \text{bool-lit} \rangle$
 $\langle \text{int-lit} \rangle$
 $\langle \text{float-lit} \rangle$
 $($

Logo, é LL(1).

5.2.10. ($\langle \text{op-rel} \rangle \langle \text{expressão-simples} \rangle \mid \langle \text{vazio} \rangle$)

first(1):

<
>
<=
>=
=
<>

follow(1):

do
“]”
then
“)”
else
“,”
;

Como a interseção do first(1) e follow(1) é vazia, portanto é LL(1).

5.2.11. $\langle \text{expressão-simples} \rangle ::= \langle \text{expressão-simples} \rangle \langle \text{op-ad} \rangle \langle \text{termo} \rangle \mid \langle \text{termo} \rangle$

Possui recursão a esquerda, logo não é LL(1).

Ao analisar a regra, percebe-se que a mesma produz um ou vários ($\langle \text{op-ad} \rangle \langle \text{termo} \rangle$) e finaliza com um $\langle \text{termo} \rangle$ a esquerda, logo, pode ser reescrita como:

$\langle \text{expressão-simples} \rangle ::= \langle \text{termo} \rangle (\langle \text{op-ad} \rangle \langle \text{termo} \rangle)^*$

first(1):

<id>
<bool-lit>
<int-lit>
<float-lit>
(

Logo, é LL(1).

5.2.12. $(\langle \text{op-ad} \rangle \langle \text{termo} \rangle)^*$

first(1):

+
-
or

Logo, é LL(1).

5.2.13. $\langle \text{fator} \rangle ::= \langle \text{variável} \rangle \mid \langle \text{literal} \rangle \mid "(" \langle \text{expressão} \rangle ")"$

first(1) de $\langle \text{variável} \rangle$:

$\langle \text{id} \rangle$

first(1) de $\langle \text{literal} \rangle$

$\langle \text{bool-lit} \rangle$

$\langle \text{int-lit} \rangle$

$\langle \text{float-lit} \rangle$

first(1) de $"(" \langle \text{expressão} \rangle ")"$

"("

Como a interseção entre os first é vazia, é LL(1).

5.2.14. $\langle \text{iterativo} \rangle ::= \text{while } \langle \text{expressão} \rangle \text{ do } \langle \text{comando} \rangle$

first(1):

while

Logo, é LL(1).

5.2.15. $\langle \text{lista-de-comandos} \rangle ::= \langle \text{comando} \rangle ; \mid \langle \text{lista-de-comandos} \rangle \langle \text{comando} \rangle ; \mid \langle \text{vazio} \rangle$

Possui recursão a esquerda, portanto não é LL(1).

Pode-se perceber que $\langle \text{lista-de-comandos} \rangle$ produz nenhum ou vários ($\langle \text{comando} \rangle ;$) concatenados entre si, portanto, reescrevendo obtemos:

$\langle \text{lista-de-comandos} \rangle ::= (\langle \text{comando} \rangle ;)^*$

first(1):

$\langle \text{id} \rangle$

if

while

begin

follow(1):

end

Como a interseção entre o first(1) e o follow(1) é vazia, podemos afirmar que é LL(1).

5.2.16. $\langle \text{lista-de-ids} \rangle ::= \langle \text{id} \rangle \mid \langle \text{lista-de-ids} \rangle , \langle \text{id} \rangle$

Possui, recursão a esquerda, não é LL(1).

Esta regra produz termos como:

<id>
<id>,<id>
<id>,<id>,<id>

Ou seja, um ou mais <id> separados por “,” logo, pode ser reescrita como:

<lista-de-ids> ::= <id> (, <id>)*

first(1):
 <id>

Logo, é LL(1).

5.2.17. (, <id>)*

first(1):
 “ , ”

Logo, é LL(1).

5.2.18. <literal> ::= <bool-lit> | <int-lit> | <float-lit>

first(1) de <bool-lit>:

true
false

first(1) de <int-lit>:

<int-lit>

first(1) de <float-lit>:

<float-lit>

Como a interseção entre os first é vazia, logo, é LL(1).

5.2.19. <op-ad> ::= + | - | or

first(1):

+
-

or

Logo, é LL(1).

5.2.20. $\langle \text{op-mul} \rangle ::= * \mid / \mid \text{and}$

first(1):

*

/

and

Logo, é LL(1).

5.2.21. $\langle \text{op-rel} \rangle ::= < \mid > \mid <= \mid >= \mid = \mid <>$

first(1):

<

>

<=

>=

=

<>

Logo, é LL(1).

5.2.22. $\langle \text{programa} \rangle ::= \text{program } \langle \text{id} \rangle ; \langle \text{corpo} \rangle .$

first(1):

program

Logo, é LL(1).

5.2.23. $\langle \text{seletor} \rangle ::= (\text{"[" } \langle \text{expressão} \rangle \text{"}]"})^*$

first(1):

[

follow(1):

:=

*

/

and

Como a interseção entre o first(1) e o follow(1) é vazia, podemos afirmar que é LL(1).

5.2.24. $\langle \text{termo} \rangle ::= \langle \text{termo} \rangle \langle \text{op-mul} \rangle \langle \text{fator} \rangle \mid \langle \text{fator} \rangle$

Possui recursão a esquerda, portanto não é LL(1).

Ao analisar a regra, percebe-se que a mesma produz um ou vários ($\langle \text{op-mul} \rangle \langle \text{fator} \rangle$) e finaliza com um $\langle \text{fator} \rangle$ a esquerda, logo, pode ser reescrita como:

$\langle \text{termo} \rangle ::= \langle \text{fator} \rangle (\langle \text{op-mul} \rangle \langle \text{fator} \rangle)^*$

Eliminando neste processo, a recursão a esquerda.

first(1):

$\langle \text{id} \rangle$
true
 $\langle \text{int-lit} \rangle$
 $\langle \text{float-lit} \rangle$
“(“

Logo, é LL(1).

5.2.25. $(\langle \text{op-mul} \rangle \langle \text{fator} \rangle)^*$

first(1):

*
/
and

LL(1)

5.2.26. $\langle \text{tipo} \rangle ::= \langle \text{tipo-agregado} \rangle \mid \langle \text{tipo-simples} \rangle$

first(1) de $\langle \text{tipo-agregado} \rangle$:

array

first(1) de $\langle \text{tipo-simples} \rangle$:

integer
real
boolean

Como a interseção entre os first é vazia, podemos afirmar que é LL(1).

5.2.27. <tipo-agregado> ::= array [<literal> .. <literal>] of
<tipo>

first(1):

array

Logo, é LL(1).

5.2.28. <tipo-simples> ::= integer | real | boolean

first(1):

integer

real

boolean

Logo, é LL(1).

5.2.29. <variável> ::= <id> <seletor>

first(1):

<id>

Logo, é LL(1).

5.2.30. <bool-lit> ::= true | false

first(1):

true

false

Logo, é LL(1).

5.3. Técnicas utilizadas e visão geral do funcionamento

A classe Parser é composta pela variável `currentToken` (símbolo corrente), métodos auxiliares e métodos `parse`. A tarefa do analisador sintático é determinar se a string de entrada é uma sentença (cadeia aceita) da gramática e, em caso afirmativo, descobrir sua estrutura frasal. Cabe aos métodos `parse` decidir qual regra de produção deve ser aplicada a um elemento da sentença.

```

public class Parser {

    private Token currentToken;
    private Token lastToken;
    private Scanner scanner;
    public int erroS = 0;

    public Parser(){

    }

    // MÉTODOS DA ANÁLISE SINTÁTICA
    public Programa parse(String fileName) throws Exception{
        Programa program;
        scanner = new Scanner(fileName);
        currentToken = this.scanner.scan();
        System.out.println("\n>>> 1. ANALISE SINTATICA <<<");
        program = parsePrograma();

        System.out.println(" > Análise Sintática concluída;\n > Total de Erros Sintáticos: 0");

        return program;
    }
}

```

Figura 10: Classe Parser e o método parse do tipo Programa

Na figura 10, foi declarada uma variável “erroS” que é incrementada ao ocorrerem erros durante alguma etapa da análise sintática e encerra o programa assim que for finalizada a etapa, enviando a quantidade de erros e o protocolo de erro para a classe Erros.

O símbolo terminal corrente é acessado pelo método auxiliar “accept”.

```

private void accept (byte expectedKind) throws Exception{
    if (currentToken.kind == expectedKind){
        lastToken = currentToken;
        currentToken = scanner.scan();
    }

    else {
        // erro sintatico, esperava 'expectedKind'
        Erros.error(11, erroS+1);
        System.out.println("      | Esperava encontrar: '"+ Token.SPELLINGS[expectedKind] + "'");
        System.out.println("      | Na linha: " + lastToken.line);
        System.out.println("      | Coluna: " + lastToken.col);
        erroS++;
        Erros.error(1, erroS);
    }
}

private void acceptIt () throws Exception{
    currentToken = scanner.scan();
}

```

Figura 11: Método accept e acceptIt

O Parser irá chamar o método “accept (e)” quando ele espera que o símbolo terminal corrente seja “e” e deseja verificar se é realmente “e”, antes de buscar o próximo símbolo de entrada. O método “acceptIt ()” não exige que o token esperado seja passado pela passagem de parâmetro.

```
private Programa parsePrograma() throws Exception {  
    //Programa ::= program <Id> ; <Corpo> . [EOF]  
  
    Programa program = new Programa();  
    accept(Token.PROGRAM);  
    accept(Token.ID);  
    accept(Token.SEMICOLON);  
    program.body = parseCorpo();  
    accept(Token.DOT);  
    accept(Token.EOF);  
    return program;  
}
```

Figura 12: Método parser parsePrograma

Durante a verificação, cada método parser segue suas próprias regras de produção definidas na gramática. Como, por exemplo, na figura 12, cujo método “accept()” segue a mesma estrutura das regras da linguagem, além disso há também a chamada do método parse de <Corpo>, que também possui suas próprias regras e devem ser validadas. O mesmo modelo é seguido pelo restantes dos métodos parser.

5.4. Exemplos de Entradas e Saídas

5.4.1. Exemplos de cadeias malformadas e respectiva resposta

```
program loona;  
  var loona: array[1 ~ 3] of boolean of array[1 ~ 3] of boolean;  
  var x: real;  
  var chuu: real;  
  var y: integer;  
  
begin  
  x := 15 + 3.5 - 7.4*3;  
  loona[1] := false;  
  loona := 133 >= 5;  
  if true and false then  
    begin  
      x := x + 1;
```

```
    x := x * x;  
    y := 1.4 * 7;  
    loona := 5 and false;  
end  
else  
    x := 0;  
end.
```

Saída:

[!][1] ERRO SINTÁTICO (Erro 1.1)
Token inesperado.
| Esperava encontrar: ';'
| Na linha: 2
| Coluna: 28

[!][2] ERRO SINTÁTICO (Erro 1.1)
Token inesperado.
| Esperava encontrar: 'begin'
| Na linha: 2
| Coluna: 28

[!][3] ERRO SINTÁTICO (Erro 1.1)
Token inesperado.
| Esperava encontrar: 'end'
| Na linha: 2
| Coluna: 28

[!][4] ERRO SINTÁTICO (Erro 1.1)
Token inesperado.
| Esperava encontrar: '.'
| Na linha: 2
| Coluna: 28

[!][5] ERRO SINTÁTICO (Erro 1.1)
Token inesperado.
| Esperava encontrar: '<EOF>'
| Na linha: 2
| Coluna: 28

As regras de produção da gramática exigem que na declaração da estrutura do array seja necessário definir o seu tamanho, que fica entre colchetes, além de definir o tipo dos elementos desse array ao final da declaração. Em decorrência disso, foi levantado um erro sintático porque o código não seguiu esse formato.

5.4.2. Exemplos de cadeias bem formadas e respectiva resposta

```
program loona;  
  var loona: array[1 ~ 3] of boolean;  
  var x: real;  
  var chuu: real;  
  var y: integer;  
  
begin  
  #x := 15 + 3;  
  x := 15 + 3.5 - 7.4*3;  
  loona[1] := false;  
  loona := 133 >= 5;  
  if 3>5 then  
    begin  
      x := x + 1;  
      x := x * x;  
      y := 1.4 * 7;  
      loona := 5 and false;  
    end.  
  end.  
end.
```

Nesse exemplo, a declaração do array seguiu o formato de declaração definido na gramática da linguagem.

6. Erros gerados no compilador

Como forma de tornar mais intuitiva a geração de mensagens de erros do compilador foi criada a classe Erros, cujo construtor recebe dois argumentos do tipo inteiro, o primeiro com o protocolo do erro e o segundo com a quantidade de erros. Essa classe gera mensagens para erros sintáticos e de contexto, os mesmos serão explicitados na próxima subseção.

Obs: Apesar da classe Erros receber como argumento uma quantidade de erros, o compilador exibirá apenas o primeiro erro detectado.

```

public class Erros {

    private static String message;

    public static void error(int protocol, int erroN) throws IOException {
        switch (protocol) {
            // ERROS SINTÁTICOS:
            case 1:

                message = String.format("Compilação derrogada durante a ANÁLISE SINTATICA. "
                    + "\nQuantidade de erros encontrada: " + erroN +
                    "\n\nOs erros de compilação são mostrados no terminal.");
                JOptionPane.showMessageDialog(null, message);
                System.out.println("\nPressione ENTER no prompt para encerrar a compilacao.");
                System.in.read();
                System.exit(1);
                break;

            case 11:
                System.out.println(" \n[!]" + erroN + "] ERRO SINTÁTICO (Erro 1.1)");
                System.out.println("      Token inesperado.");
                break;

            case 12:
                System.out.println(" \n[!]" + erroN + "] ERRO SINTATICO (Erro 1.2)");
                System.out.println("      Booleano invalido.");
                System.out.println("      > Atribuição Lógica deve conter 'true' ou 'false'");
                break;
        }
    }
}

```

Figura 12: Classe Erros - erros sintáticos

```

//ERROS DE CONTEXTO:
case 2:

    message = String.format("Compilação derrogada durante a ANÁLISE DE CONTEXTO. "
        + "\nQuantidade de erros encontrada: " + erroN +
        "\n\nOs erros de compilação são mostrados no terminal.");
    JOptionPane.showMessageDialog(null, message);
    System.out.println("\nPressione ENTER para encerrar a compilacao.");
    System.in.read();
    System.exit(1);
    break;

case 211:
    System.out.println(" \n[!]" + erroN + "] ERRO DE CONTEXTO (Erro 2.1.1)");
    break;

case 212:
    System.out.println(" \n[!]" + erroN + "] ERRO DE CONTEXTO (Erro 2.1.2)");
    break;

case 221:
    System.out.println(" \n[!]" + erroN + "] ERRO DE CONTEXTO (Erro 2.2.1)");
    System.out.println("      Atribuicao de valores incompativeis.");
    break;

```

Figura 13: Classe Erros - erros de contexto

6.1. Erros Tipo 1: Sintático

1.1: Token inesperado:

Ocorre quando o token analisado não estava na relação de tokens da linguagem.

1.2: Booleano inválido:

Ocorre quando a expressão booleana analisada não é válida para a linguagem.

1.3: Comando inválido:

Ocorre quando o comando analisado não segue a estrutura esperada pela especificação da linguagem.

1.4: Fator inválido:

Ocorre quando o fator analisado não segue a estrutura esperada pela especificação da linguagem.

1.5: Literal inválido:

Ocorre quando, ao atribuir um literal à uma variável, o mesmo não é válido, ou seja, não faz parte da linguagem.

1.6: Tipo inválido:

Ocorre quando o tipo-agregado e o tipo-simples não atribuídos ou criados de forma correta.

6.2. Erros Tipo 2: De Contexto

6.2.1. Erros 2.1: Declarações

2.1.1: Identificador já declarado:

Ocorre ao declarar novamente um mesmo identificador no programa.

2.1.2: Identificador não declarado:

Ocorre ao utilizar um identificador que não foi declarado previamente no programa.

6.2.2. Erros 2.2: Tipos

2.2.1: Atribuição de valores incompatíveis:

Ocorre ao atribuir valores com tipos incompatíveis para variáveis declaradas.

Ex:

```
program loona;  
  var x: real;  
  
begin  
  x := true  
end.
```

2.2.2: Esperava-se expressão booleana:

Ocorre ao inserir uma expressão que não resulta em um booleano em um condicional.

Ex:

```
program loona;  
  var x: real;  
  
begin  
  if (2+3) then  
    x := 5;  
end.
```

2.2.3: Comparação entre valores incompatíveis:

Ocorre ao realizar comparações entre valores de tipos que exigem propriedade ou valor especificado pela operação.

2.2.4: Operandos inválidos:

Ocorre quando o operando não diz respeito a operação realizada.

2.2.5: Seletor inválido:

Ocorre quando o tipo do seletor utilizado não é compatível com o tipo esperado.

7. Montagem e impressão da árvore de sintaxe abstrata

7.1. Descrição das estruturas de dados e algoritmos utilizados

Para construir a AST foi utilizado o padrão de projeto Visitor, ele é responsável por separar a representação física da estrutura de dados das operações que devem ser aplicadas sobre a mesma. O resultado disso são os métodos “visit” utilizados.

Para a construção da árvore de sintaxe abstrata (AST), inicialmente foram criadas, tendo como base a gramática livre de contexto ASTs para cada método parser criado na classe Parser. Seguem exemplos de árvores abaixo:

Programa:



Figura 14: AST para a regra “programa”

```
package AST;
import Visitor.Visitor;

/**
 *
 * @author edjair
 */

public class Programa {
    public Corpo body;

    public void visit(Visitor v) {
        v.visitPrograma(this);
    }
}
```

Figura 15: Classe Programa que utiliza o padrão Visitor para gerar a AST

Corpo:

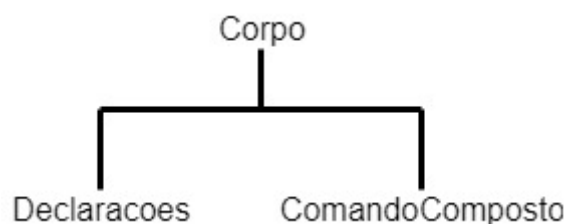


Figura 16: AST para a regra “corpo”

```

package AST;
import Visitor.Visitor;

/**
 *
 * @author edjair
 */

public class Corpo {
    public Declaracoes declarations;
    public ComandoComposto compositeCommand;

    public void visit(Visitor v){
        v.visitCorpo(this);
    }
}

```

Figura 17: Classe Corpo

Cada AST foi transformada em uma classe, e as folhas em ponteiros para a classe correspondente. Como Comando, Tipo e Fator possuem mais de uma AST, para estas, foram criadas classes abstratas onde as ASTs que fazem parte do conjunto estendem a classe mãe. A classe Fator é um exemplo disso, a mesma se estende para a classe Literal.

No início da classe Parser são importadas as classes do pacote AST, cada classe importada é utilizada na criação do respectivo método parse de maneira intuitiva, como podemos ver no exemplo abaixo.

```

private Atribuicao parseAtribuicao() throws Exception{
    //Atribuicao ::= <Variavel> := <Expressao>
    Atribuicao becomes = new Atribuicao();
    becomes.variable = parseVariavel();
    accept(Token.BECOMES);
    becomes.expression = parseExpressao();
    return becomes;
}

```

Figura 18: Método parseAtribuicao do tipo Atribicao da classe Parser

Como podemos notar na figura 18, o método parseAtribuicao() possui o tipo Atribicao, ou seja, é um método que possui todos os atributos e métodos que a classe Atribicao possui, inclusive o método de “visit(Visitor v)”, responsável por visitar a hierarquia de nós da AST.

```

public class Atribuicao extends Comando{
    public Variavel variable;
    public Expressao expression;
    public String type;

    public void visit(Visitor v){
        v.visitAtribuicao(this);
    }
}

```

Figura 19: Classe Atribuicao

A montagem da árvore é feita pela classe Parser, que além de realizar a análise sintática, realiza a montagem da árvore ao conectar as ASTs através dos ponteiros.

Cada método iniciado com parse possui um método Visitor, com isso os métodos são responsáveis por montar a árvore com os ponteiros retornados, até que todos estejam conectados e toda a árvore possa ser acessada partindo da raiz (Programa).

A impressão desta árvore é realizada pela classe Printer, que implementa o padrão de projeto Visitor, a ideia desse padrão é separar as operações que serão executadas em determinada estrutura de sua representação. Assim, incluir ou remover operações não terá nenhum efeito sobre a interface da estrutura, permitindo que o resto do sistema funcione sem depender de operações específicas. Abaixo podemos ver sua implementação:

```

public interface Visitor {

    public void visitAtribuicao(Atribuicao becomes);
    public void visitBoolLit(BoolLit boolLit);
    public void visitComandoComposto(ComandoComposto compositeCommands);
    public void visitCondicional(Condicional conditional);
    public void visitCorpo(Corpo body);
    public void visitDeclaracaoDeVariavel(DeclaracaoDeVariavel variableDeclaration);
    public void visitDeclaracoes(Declaracoes declarations);
    public void visitExpressao(Expressao expression);
    public void visitExpressaoSimples(ExpressaoSimples simpleExpression);
    public void visitIterativo(Iterativo iterative);
    public void visitListaDeComandos(ListaDeComandos listOfCommands);
    public void visitListaDeIds(ListaDeIds listOfIds);
    public void visitLiteral(Literal literal);
    public void visitPrograma(Programa program);
    public void visitSeletor(Seletor selector);
    public void visitTermo(Termo term);
    public void visitTipoAgregado(TipoAgregado type);
    public void visitTipoSimples(TipoSimples type);
    public void visitVariavel(Variavel variable);
}

```

Figura 20: Interface da classe Visitor

Ao implementar o Visitor em Printer e inserir os respectivos métodos visit nas demais classes acima citadas, como pode ser observado abaixo, garantimos que toda a árvore será percorrida, e todos terminais serão impressos, permitindo também determinar o nível em que se encontram, através de um contador e determinar o caminho até a folha partindo da raiz, através do uso de um StringBuilder, onde é inserido um nome em cada método.

```

public class Printer implements Visitor {

    //Métodos da classe:
    int i = 0;
    int lvl = 0;
    StringBuilder bd = new StringBuilder();

    public String print(Programa program) {
        //System.out.println("\n>>> IMPRESSAO DA ARVORE <<<\n");
        bd.append("\t\n - IMPRESSÃO: ÁRVORE SINTÁTICA ABSTRATA");
        program.visit(this);
        String arvorest = bd.toString();

        return (arvorest);
    }

    //Indenta os níveis de saída da árvore para cada nível aprofundado
    private void out(String nodeInfo, int indentLevel) {
        for (int x = 0; x <= indentLevel / 2; x++) {
            //System.out.print("| ");
            bd.append("| ");
            //System.out.print(" ");
            bd.append(" ");
        }

        bd.append(nodeInfo).append("\r\n");
    }
}

```

Figura 21: Classe Printer e os métodos print() e out()

Na figura 21, podemos ver dois importantes métodos da classe Printer, o método print(), que é responsável por fazer a impressão da árvore e o método out(), responsável pela indentação da árvore.

Outro ponto que deve ser salientado é a sobrescrição dos métodos criados na interface Visitor, expostos na figura abaixo, isso garante que a visitação dos nós e a escrita da árvore será feita de forma condizente.

```

@Override
public void visitAtribuicao(Atribuicao becomes) {
    i++;
    out("Atribuicao", i);
    if(becomes.variable != null){
        i++; if (i > lvl) lvl = i;
        becomes.variable.visit(this);
        i--;
    }
    if(becomes.expression != null ){
        i++; if (i > lvl) lvl = i;
        becomes.expression.visit(this);
        i--;
    }
    i--;
}

@Override
public void visitBoolLit(BoolLit boolLit) {
    out("Boolean Literal", i);
    out(boolLit.name.value,i+2);
}

```

Figura 22: Exemplos de métodos Visitor sobrescritos

7.2. Exemplos de programas-fonte e respectivas árvores geradas.

7.2.1. Exemplo 1

Programa
<pre> program loona; var loona: array[1 ~ 3] of boolean; var x: real; var chuu: real; var y: integer; begin loona[1] := false; end. </pre>

Árvore Gerada:

- IMPRESSÃO: ÁRVORE SINTÁTICA ABSTRATA

Programa

| Corpo

| | Declaracoes

| | | Declaracao de Variavel

| | | | Lista de IDs

| | | | | loona

| | | | Tipo Agregado

| | | | | 1

| | | | | 3

| | | | Tipo Simples

| | | | | boolean

| | | Declaracao de Variavel

| | | | Lista de IDs

| | | | | x

| | | | Tipo Simples

| | | | | real

| | | Declaracao de Variavel

| | | | Lista de IDs

| | | | | chuu

| | | | Tipo Simples

| | | | | real

| | | Declaracao de Variavel

| | | | Lista de IDs

| | | | | y

| | | | Tipo Simples

| | | | | integer

| | Comando Composto

| | | Lista de Comandos

| | | | Atribuicao

| | | | | Variavel

| | | | | | loona

| | | | | Selektor

| | | | | | Expressao

| | | | | | | Expressao Simples

| | | | | | | | Termo

| | | | | | | | | Literal

| | | | | | | | | 1

| | | | | Expressao

| | | | | | Expressao Simples

| | | | | | | Termo

| | | | | | | | Boolean Literal

| | | | | | | | | false

7.2.2. Exemplo 2

Programa
<pre>program loona; var loona: array[1 ~ 3] of boolean; var x: real; var chuu: real; var y: integer; var isabella: integer; begin loona[1] := false; isabella := 1; loona := 133 >= 5; if 32 > 5 then begin x := x + 1; x := x * x; y := 1.4 * 7; loona := 5 and false; end else x := 0; end. end.</pre>

Árvore Gerada:

- IMPRESSÃO: ÁRVORE SINTÁTICA ABSTRATA

Programa

	Corpo
	Declaracoes
	Declaracao de Variavel
	Lista de IDs
	loona
	Tipo Agregado
	1
	3
	Tipo Simples
	boolean
	Declaracao de Variavel
	Lista de IDs
	x

[illegible]


```

Variavel
| loona
Expressao
| Expressao Simples
| Termo
| Literal
| 5
| and
| Boolean Literal
| false
Atribuicao
| Variavel
| x
Expressao
| Expressao Simples
| Termo
| Literal
| 0

```

8. Análise de contexto

8.1. Descrição das estruturas de dados

Foi criada a classe `TabelaId`, que utiliza estrutura de dados `HashMap`, que irá salvar os identificadores na tabela hash implementada.

```
public class TabelaId {  
  
    public int erroV = 0;  
    HashMap table;  
  
    TabelaId() {  
        table = new HashMap();  
    }  
}
```

Figura 23: Classe `TabelaId` e seu construtor

Além disso, a classe implementa três métodos importantes:

- `enter()`, responsável por inserir o identificador na tabela hash caso ele não tenha sido declarado anteriormente (irá gerar um erro) .

```
public void enter(Token id, DeclaracaoDeVariavel declaration) throws IOException {  
    if (table.put(id.value, declaration) != null) {  
        Erros.error(211, erroV + 1);  
        System.out.println("    Identificador " + id.value + " ja declarado.");  
        System.out.println("    | Na linha:" + id.line);  
        //System.out.println("    | Coluna:" + id.col);  
        erroV++;  
        Erros.error(2, erroV);  
    }  
}
```

Figura 24: Método `enter()` da classe `TabelaId`

- retrieve() do tipo DeclaracaoDeVariavel, responsável por recuperar um identificador da tabela, caso não o encontre, gerará um erro.

```
public DeclaracaoDeVariavel retrieve(Token id) throws IOException {
    if (table.containsKey(id.value) == false) {
        Erros.error(212, erroV + 1);
        System.out.println("    Identificador '" + id.value + "' nao declarado.");
        System.out.println("    | Na linha:" + id.line);
        //System.out.println("    | Coluna:" + id.col);
        erroV++;
        Erros.error(2, erroV);
    } else {
        return (DeclaracaoDeVariavel) table.get(id.value);
    }
    return null;
}
```

Figura 25: Método retrieve() da classe Tabelald

- print(), que irá imprimir a tabela de identificadores

```
public String print() {

    StringBuilder bd = new StringBuilder();

    bd.append("\t\n - IMPRESSÃO: TABELA DE IDENTIFICADORES\n");
    bd.append("\n-----");

    bd.append("\n " + "VAR" + "\t|\t" + "\tADDR");
    bd.append("\n-----");
    bd.append("\n-----");

    table.keySet().forEach((name) -> {
        String key = name.toString();
        String value = table.get(name).toString();

        bd.append("\n ").append(key).append("\t|\t").append(value);
        bd.append("\n-----");
    });

    String tableprint = bd.toString();
    return (tableprint);
}
```

Figura 26: Método print() da classe Tabelald

8.2. Algoritmos utilizados na fase de identificação

Como um dos passos para verificar se as variáveis utilizadas no corpo do programa foram declaradas anteriormente, foi criada a classe DeclaracaoDeVariavel que possui dois atributos, o listOfIds, que referencia a classe ListaDeIds e o type, que referencia a classe abstrata Tipo.

```

public class DeclaracaoDeVariavel {
    public ListaDeIds listOfIds;
    public Tipo type;

    public void visit(Visitor v){
        v.visitDeclaracaoDeVariavel(this);
    }
}

```

Figura 27: Classe DeclaracaoDeVariavel

A classe ListaDeIds, na figura 28, por sua vez, cria o método visit() e dois atributos, um “next” com o tipo da própria classe, que será responsável por acessar os próximos ids da lista e o outro como “id” com o tipo da classe Token, como pode ser visto abaixo.

```

public class ListaDeIds {
    public Token id;
    public ListaDeIds next;

    public void visit(Visitor v){
        v.visitListaDeIds(this);
    }
}

```

Figura 28: Classe ListaDeIds

A classe responsável por realizar a identificação é a Checker, que assim como o Printer implementa o Visitor, garantindo que toda a árvore será percorrida. O método check() irá verificar se ocorreu algum erro na análise de contexto, em caso positivo, será enviado o código do primeiro erro ocorrido e a quantidade de erros, também temos o atributo table do tipo TabelalId, que trará consigo os métodos para inserir e recuperar valores da tabela hash.

```

public class Checker implements Visitor {

    TabelaId table;
    public int erroC = 0;

    Checker() {

        table = new TabelaId();

    }

    public void check(Programa program) throws IOException {

        System.out.println("\n>>> 2. ANALISE DE CONTEXTO <<<");
        program.visit(this);

        if (erroC != 0) {
            Erros.error(2, erroC);
        } else {
            System.out.println(" > Análise de Contexto concluída;\n > Total de Erros de Contexto: 0");
        }

    }

}

```

Figura 29: Classe Checker, seu construtor e o método check()

A classe Checker também é responsável por utilizar os métodos de visitação do Visitor, logo, também é um dos responsáveis pelos passos de impressão da AST. Segue abaixo um exemplo de método do Checker:

```

@Override
public void visitSeletor(Seletor selector) {
    Seletor aux = selector;
    while (aux != null) {
        aux.expression.visit(this);

        if (!aux.expression.type.equals("integer")) {
            try {
                Erros.error(225, erroC + 1);
            } catch (IOException ex) {
                Logger.getLogger(Checker.class.getName()).log(Level.SEVERE, null, ex);
            }
            System.out.println("      | Na linha: " + aux.expression.operator.line);
            erroC++;
        }

        aux = aux.next;
    }
}

```

Figura 30: Método visitSeletor() da classe Checker

Na figura 30, o método exposto tem como argumento um seletor do tipo Seletor, o mesmo será recebido internamente por uma variável auxiliar, também do tipo Seletor, e irá entrar em um condicional que ficará em operação enquanto houver seletores para ser iterados. Dentro do condicional será feita a verificação do tipo do seletor, caso ele seja diferente do tipo inteiro, o que não é permitido e será enviado para a classe Erro o código e a quantidade de erros.

Ao executar o código abaixo, poderemos verificar o funcionamento do compilador no que diz respeito a tabela de identificadores gerada.

Código Funcionando

```
program loona;
  var loona: array[1 ~ 3] of boolean;
  var x: real;
  var chuu: real;
  var y: integer;
  var isabella: integer;

begin

  loona[1] := false;
  isabella := 1;
  loona := 133 >= 5;
  if 32 > 5 then
    begin
      x := x + 1;
      x := x * x;
      y := 1.4 * 7;
      loona := 5 and false;
    end
  else
    x := 0;
  end.
end.
```

Saída

- IMPRESSÃO: TABELA DE IDENTIFICADORES

VAR	ADDR
isabella	AST.DeclaracaoDeVariavel@4820dd7e
x	AST.DeclaracaoDeVariavel@5a921b02
loona	AST.DeclaracaoDeVariavel@71de0693
y	AST.DeclaracaoDeVariavel@48c376d2
chuu	AST.DeclaracaoDeVariavel@6469999

Código com erros

```
program loona;  
  var loona: array[1 ~ 3] of boolean;  
  var x: real;  
  var chuu: real;  
  var y: integer;
```

```
begin
```

```
  loona[1] := false;  
  isabella := 1;  
  loona := 133 >= 5;  
  if 32 > 5 then  
    begin  
      x := x + 1;  
      x := x * x;  
      y := 1.4 * 7;  
      loona := 5 and false;
```

```
    end
```

```
  else
```

```
    x := 0;
```

```
end.
```

Saída

```
>>> 1. ANALISE SINTATICA <<<
> Análise Sintática concluída;
> Total de Erros Sintáticos: 0

>>> 2. ANALISE DE CONTEXTO <<<

[!][1] ERRO DE CONTEXTO (Erro 2.1.2)
  Identificador 'isabella' nao declarado.
  | Na linha:11
```

8.3. Descrição das estruturas de dados e algoritmos utilizados na fase de verificação de tipos; ilustração com exemplos

8.3.1. Verificação de tipos

A fase de verificação de tipos também é feita na classe Checker, responsável pela análise de contexto. Toda classe que terá seu tipo analisado, possui um atributo `tipo` em sua definição. Este atributo será do tipo “Tipo”, que é uma classe abstrata, para a classe `TipoAgregado` e do tipo `Token` para a classe `TipoSimples`. Ambas as classes estendem a classe `Tipo` e possuem métodos de visitação `Visitor` implementados. Além disso, `TipoAgregado`, que é utilizado para arrays possui dois atributos do tipo `Literal`, que são os índices do array.

```
public abstract class Tipo {

    public String type;

}
```

Figura 31: Classe abstrata Tipo

```
public class TipoSimples extends Tipo{
    public Token tipo;

    public void visit(Visitor v){
        v.visitTipoSimples(this);
    }
}
```

Figura 32: Classe TipoSimples

```

public class TipoAgregado extends Tipo {
    public Literal literal1, literal2;
    public Tipo typo;

    public void visit(Visitor v){
        v.visitTipoAgregado(this);
    }
}

```

Figura 33: Classe TipoAgregado

8.3.2. Verificação de tipo de variáveis.

```

@Override
public void visitTipoAgregado(TipoAgregado type) {

    //Visualiza se o Tipo incluído na declaração do Tipo-Agregado é Tipo-Agregado ou Tipo-Simples
    if (type.typo instanceof TipoAgregado) {
        ((TipoAgregado) type.typo).visit(this);
    } else {
        //Caso contrário, verifica-se e visita-se a instância do Tipo-Simples
        if (type.typo instanceof TipoSimples) {
            ((TipoSimples) type.typo).visit(this);
        }
    }
    type.type = type.typo.type;
}

@Override
public void visitTipoSimples(TipoSimples type) {
    type.type = type.typo.value;
}

```

Figura 34: Métodos visitTipoAgregado() e visitTipoSimples() da classe Checker

A verificação do tipo de variável é feita na figura 34, primariamente no método visitTipoAgregado, que irá verificar e visitar a instância do TipoAgregado caso o tipo seja do tipo TipoAgregado e, caso seja do tipo TipoSimples, visitar a instância do TipoSimples. Resumindo, caso a variável seja um array (tipo agregado), irá visitar o TipoAgregado, caso seja uma variável simples, irá visitar o TipoSimples.

8.3.3. Verificação de tipo da expressão

```
@Override
public void visitExpressao(Expression expression) {
    if (expression.simpleExpression != null) {
        expression.simpleExpression.visit(this);
        expression.type = expression.simpleExpression.type;
    }
    if (expression.simpleExpressionR != null) {
        expression.simpleExpressionR.visit(this);

        if ("real".equals(expression.simpleExpressionR.type)
            || expression.simpleExpressionR.type.equals("integer")) {
            expression.type = "boolean";
        } else {
            try {
                Erros.error(223, erroC + 1);
            } catch (IOException ex) {
                Logger.getLogger(Checker.class.getName()).log(Level.SEVERE, null, ex);
            }
            System.out.println("      | Na linha: " + expression.operator.line);
            erroC++;
        }
    }
}
```

Figura 35: Método visitExpressao() da classe Checker

O método visitExpressao() é explicado da seguinte maneira: Confere se a primeira expressão é vazia, caso não seja, o tipo da expressão é o mesmo tipo da primeira expressão, confere se a expressão seguinte, separada por um operador relativo, é vazia, caso não seja, o tipo da expressão à direita deve ser real ou inteiro, a fim de fazer a comparação relativa, a expressão então recebe o valor booleano, caso contrário, deve-se retornar mensagem de erro.

Para a verificação de tipo foi implementado o método visitExpressaoSimples() que será exposto abaixo.

```
@Override
public void visitExpressaoSimples(ExpressionSimples simpleExpression) {
    ExpressaoSimples aux = simpleExpression;
    String place = null;
    while (aux != null) {
        if (aux.term != null) {
            aux.term.visit(this);
            if (place == null) {
                place = aux.term.type;
            }
        }
        if (aux.operator != null) {

            switch (aux.operator.kind) {
                case Token.SUM:
                case Token.SUB:
                    switch (place) {

                        case "integer":
```

```

switch (aux.term.type) {
    case "integer":
        place = "integer";
        break;
    case "real":
        place = "real";
        break;
    default: {
        try {

            Erros.error(224, erroC + 1);
        } catch (IOException ex) {
            Logger.getLogger(Checker.class.getName()).
                log(Level.SEVERE, null, ex);
        }
        System.out.println("    | Na linha: " + aux.operator.line);
        erroC++;
    }
    break;
case "real":
    if (aux.term.type.equals("boolean")) {
        try {
            Erros.error(224, erroC + 1);
        } catch (IOException ex) {
            Logger.getLogger(Checker.class.getName()).
                log(Level.SEVERE, null, ex);
        }
        System.out.println("    | Na linha: " + aux.operator.line);
        erroC++;
    }
    place = "real";
    break;
default: {
    try {
        Erros.error(224, erroC + 1);
    } catch (IOException ex) {
        Logger.getLogger(Checker.class.getName()).
            log(Level.SEVERE, null, ex);
    }
}
System.out.println("    | Na linha: " + aux.operator.line);
erroC++;

}
break;
case Token.OR:
    if (!place.equals("boolean") || !aux.term.type.equals("boolean")) {
        try {
            Erros.error(224, erroC + 1);
        } catch (IOException ex) {
            Logger.getLogger(Checker.class.getName()).log(Level.SEVERE,
null, ex);
        }
        System.out.println("    | Na linha: " + aux.operator.line);
        erroC++;
    }

```

```
        }
        place = "boolean";
        break;
    }
}
}
aux = aux.next;
}

simpleExpression.type = place;

}
```

Obs: O código acima foi exposto dentro de uma tabela pela dificuldade em colocá-lo em uma única imagem de maneira legível ou em múltiplas imagens de forma a manter a indentação.

O código acima pode ser explicado da seguinte maneira: `ExpressãoSimples` recebe uma sequência de termos separados por um operador adicional (pode receber vazio), confere se o primeiro termo é vazio, `place` recebe o tipo do primeiro termo, confere se existem termos subsequentes, caso haja, vê se os operandos utilizados têm compatibilidade com os tipos dos termos. Caso não exista compatibilidade, deve-se retornar mensagens de erro, caso seja uma operação de soma ou subtração, o tipo do primeiro termo deve ser compatível com o tipo do segundo termo. Caso os tipos sejam incompatíveis, é retornada uma mensagem de erro

8.3.4. Verificação de tipo para comando iterativo e comando condicional

```
@Override
public void visitCondicional(Condicional conditional) {
    if (conditional.expression != null) {
        conditional.expression.visit(this);
        if (!conditional.expression.type.equals("boolean")) {
            try {
                Erros.error(222, erroC + 1);
            } catch (IOException ex) {
                Logger.getLogger(Checker.class.getName()).log(Level.SEVERE, null, ex);
            }
            System.out.println("    | Tipo retornado: " + conditional.expression.type);
            System.out.println("    | Na linha: " + conditional.expression.operator.line);
            erroC++;
        }
    }
}
```

Figura 36: Método visitCondicional() da classe Checker

```
public class Condicional extends Comando{
    public Expressao expression;
    public Comando command;
    public Comando commandElse;

    public void visit(Visitor v){
        v.visitCondicional(this);
    }
}
```

Figura 37: Classe Condicional

O método visitCondicional() verifica se a expressão do Condicional é inicialmente vazia, caso contrário, verifica se a mesma é do tipo “boolean”, se não for, exibe uma mensagem de erro.


```

@Override
public void visitIterativo(Iterativo iterativo) {
    if (iterativo.expression != null) {
        iterativo.expression.visit(this);
    }

    if (!iterativo.expression.type.equals("boolean")) {
        try {
            Erros.error(222, erroC + 1);
        } catch (IOException ex) {
            Logger.getLogger(Checker.class.getName()).log(Level.SEVERE, null, ex);
        }
        System.out.println("        | Tipo retornado: " + iterativo.expression.type);
        System.out.println("        | Na linha: " + iterativo.expression.operator.line);
        erroC++;
    }

    if (iterativo.command instanceof Atribuicao) {
        ((Atribuicao) iterativo.command).visit(this);
    } else if (iterativo.command instanceof ComandoComposto) {
        ((ComandoComposto) iterativo.command).visit(this);
    } else if (iterativo.command instanceof Iterativo) {
        ((Iterativo) iterativo.command).visit(this);
    } else if (iterativo.command instanceof Condicional) {
        ((Condicional) iterativo.command).visit(this);
    }
}
}

```

Figura 38: Método visitIterativo() da classe Checker

```

public class Iterativo extends Comando{
    public Expressao expression;
    public Comando command;

    public void visit(Visitor v){
        v.visitIterativo(this);
    }
}

```

Figura 39: Classe Iterativo

O método visitIterativo() confere se a expressão do iterativo é vazia, a expressão do iterativo deve retornar um tipo booleano (true ou false) e associar o comando do iterativo a atribuição, condicional, iterativo ou comando composto. Caso não retorne, deve-se exibir uma mensagem de erro.

8.3.5. Verificação de tipo para o comando de atribuição

```
@Override
public void visitAtribuicao(Atribuicao becomes) {
    becomes.variable.visit(this);
    becomes.expression.visit(this);

    //Verificacao de tipos:
    if (becomes.variable.type != null) {
        if (becomes.variable.type.equals(becomes.expression.type)) {
            becomes.type = becomes.variable.type;
        } else if (becomes.variable.type.equals("real") && becomes.expression.type.equals("integer")) {
            becomes.type = becomes.variable.type;
        } else {
            try {
                Erros.error(221, erroC + 1);
            } catch (IOException ex) {
                Logger.getLogger(Checker.class.getName()).log(Level.SEVERE, null, ex);
            }
            System.out.println("      | Na linha: " + becomes.variable.id.line);
            System.out.println("      | Coluna:" + becomes.variable.id.col);
            erroC++;
        }
    }
}
```

Figura 40: Método visitAtribuicao() da classe Checker

```
public class Atribuicao extends Comando{
    public Variavel variable;
    public Expressao expression;
    public String type;

    public void visit(Visitor v){
        v.visitAtribuicao(this);
    }
}
```

Figura 41: Classe Atribuicao

O método visitAtribuicao() confere se a atribuição não é vazia, caso o tipo da variável seja o mesmo tipo da expressão, a atribuição recebe a mesma tipagem, caso a variável tenha tipo real e a expressão tenha tipo inteiro, a atribuição é do tipo real, caso contrário, a atribuição tem valores incompatíveis e é gerada uma mensagem de erro.

8.4. Exemplos de entradas e saídas que comprovem a aceitação de cadeias bem formadas e a rejeição de cadeias mal formadas.

8.4.1. Cadeia bem formada

Código fonte.	Código objeto.
<pre> program loona; var loona: array[1 ~ 3] of boolean; var x: real; var chuu: real; var y: integer; var isabella: integer; begin loona[1] := false; isabella := 1; loona := 133 >= 5; if 32 > 5 then begin x := x + 1; x := x * x; y := 1.4 * 7; loona := 5 and false; end else x := 0; end end </pre>	<pre> 0x0: PUSH 5 0x1: LOADL false 0x2: STORE loona 0x3: LOADL 1 0x4: STORE isabella 0x5: LOADL 133 0x6: LOADL 5 0x7: CALL gtoe 0x8: STORE loona 0x9: LOADL 32 0x10: LOADL 5 0x11: CALL gt 0x12: JUMPIF(0) ELSE1 0x13: LOAD x 0x14: LOADL 1 0x15: CALL add 0x16: STORE x 0x17: LOAD x 0x18: LOAD x 0x19: CALL mult 0x20: STORE x 0x21: LOADL 1.4 0x22: LOADL 7 0x23: CALL mult 0x24: STORE y 0x25: LOADL 5 0x26: LOADL false 0x27: CALL and 0x28: STORE loona 0x29: JUMP ELSE2 ELSE1: 0x30: LOADL 0 0x31: STORE x ELSE2: 0x32: POP 5 0x33: HALT </pre>

8.4.2. Cadeia mal formada

```

program loona;
  var loona: array[1 ~ 3] of
boolean;
  var x: real;
  var chuu: real;
  var y: integer;
  var isabella: integer;

begin

  loona[1] := false;
  isabella := 1;
  loona := 133 >= 5;
  if 32 + 5 then
    begin
      x := x + 1;
      x := x * x;
      y := 1.4 * 7;
      loona := 5 and false;
    end
  else
    x := 0;
  end.

```

```

>>> 1. ANALISE SINTATICA <<<
> Análise Sintática concluída;
> Total de Erros Sintáticos: 0

```

```

>>> 2. ANALISE DE CONTEXTO
<<<

```

```

[!][1] ERRO DE CONTEXTO (Erro
2.2.2)

```

```

    Esperava-se expressao
booleana.

```

```

    | Tipo retornado: integer

```

9. Ambiente de execução

9.1. Tipo de máquina usada e expressões utilizadas

Abaixo seria o template sugerido no livro seria esse, mas optamos por utilizar diretamente a biblioteca de instruções.

Apresentaremos uma especificação de código para a tradução do Mini-Triangle para o código TAM.

As classes de expressões relevantes nesta linguagem são Programa, Comando, Expressão, Operador, V-nome e Declaração, primeiro introduzimos funções de código para estas classes:

RUM	Programa -> Instrução*
EXECUTE	Comando -> Instrução*
EVALUATE	Expressão -> Instrução*
FETCH	V-nome -> Instrução*
ASSIGN	V-nome -> Instrução*
ELABORATE	Declaração -> Instrução*

O código de objeto de cada expressão é uma sequência de instruções que serão processadas conforme mostrado na tabela do item 10 desta documentação.

9.2. Representação dos Dados

Logo abaixo temos a tabela de representação dos tipos de dados da linguagem fonte utilizada por este compilador.

Representação dos tipos de dados		
Integer	Utiliza 4 bytes	números inteiros entre 0 e 2147483647
Float	Utiliza 8 bytes	para reais entre 0 e 10^{4932}
Boolean	Utiliza 1 byte	para indicar true ou false

9.3. Formas de alocação de memória empregadas

Obs: A lógica de alocação seria essa, mas não foi tratada no código, chama apenas o endereço base.

Seria utilizada a alocação por linha, onde matrizes multidimensionais que pertencem a uma mesma linha são armazenados em sequência, por exemplo:

```
var loona: array[0 ~ 2] of array[1 ~ 3] of boolean;
```

Este seria armazenado da seguinte maneira.

v[0][1]
v[0][2]
v[0][3]
v[1][1]
v[1][2]
v[1][3]
v[2][1]
v[2][2]
v[2][3]

10. Linguagem-objeto

Obs: estamos considerando que todas as variáveis utilizam 1 byte para o push e o pop.

Mnemônico.	Sintaxe.	Semântica.
LOAD	LOAD(n) d[r]	Pega n bytes a partir do endereço (d + registrador r) e coloca no topo da pilha.
LOADL	LOADL d	Coloca o literal d no topo da pilha.
STORE	STORE (n) d[r]	Retira n bytes do topo da pilha e coloca no endereço (d + registrador r).
CALL	CALL label	Executa a sub rotina label.
PUSH	PUSH d	Aloca d bytes na memória.
POP	POP (n)	Desaloca n bytes do topo da pilha.
JUMP	JUMP d [r]	Salta até o endereço especificado (d + r)
JUMPIF	JUMPIF (n) d[r]	Retira um endereço do topo da pilha e realiza o salto para (d + registrador r).
HALT	HALT	Finaliza a execução o programa.

11. Geração de código

A função do gerador de código é realizar a tradução de uma linguagem fonte para linguagem objeto semanticamente equivalentes. Quando projetamos um gerador de código, portanto, devemos nos guiar pela semântica do código-fonte.

Para isso foi criada a classe `Coder`, que implementa o padrão de projeto visitor, a definição junto com seus atributos são mostrado logo abaixo.

```
public class Coder implements Visitor {

    int mem = 0;
    int ElseCont = 1;
    int varqtd = 0;
    int ElseAux;
    int WhileCont = 1, WhileAux = 0;
    //private static String message;
    StringBuilder bd = new StringBuilder();

    public String encode(Programa program) {

        System.out.println("\n>>> 3. GERACAO DE CODIGO <<<");
        program.visit(this);
        //bd.append("\n0x").append(mem).append(" ").append("END");
        if (ElseAux != 0)
            bd.append("\n\nELSE").append(ElseAux).append(": \n");
        else
            bd.append("\n\nEND").append(": \n");

        bd.append("\n0x").append(mem).append(": \t").append("POP \t").append(varqtd);
        mem++;
        bd.append("\n0x").append(mem).append(": \t").append("HALT");

        System.out.println(" > Geracao deCodigo concluida;");
        System.out.println(" \n-----\n");

        String codgen = bd.toString();

        return (codgen);
    }
}
```

Figura 42: Classe `Coder` e o método `encode()`

Na definição da classe `Coder` temos a declaração do `StringBuilder` que fará o papel de uma pilha, todas as inserções serão no próximo elemento vazio da pilha através do método “`append()`”, nativo do `StringBuilder`.

Além disso, temos seis atributos na classe, todos do tipo “int”, o “mem”, responsável por indicar a posição de memória atual, o “varqtd”, responsável por guardar a quantidade de variáveis utilizadas, por fim, “ElseCont”, “ElseAux”, “WhileCont” e “WhileAux” irão auxiliar operações na classe. O método encode() é o responsável por gerar todo o código que foi traduzido da linguagem-fonte para a linguagem-objeto nos métodos “visit”.

11.1. Funções de código implementados

Template Equivalente	Métodos responsáveis do coder.	Função.
Run P	visitPrograma	Responsável por iniciar o programa, encerrar e retornar o controle para o sistema operacional.
Execute C	visitIterativo visitCondicional	Executa o comando C. Realiza a montagem do template pré definido para cada tipo de comando.
Evaluate E	visitExpressao	Realiza o cálculo da expressão e coloca o valor resultante no topo da pilha.
Fetch	visitLiteral visitVariavel	Carrega o conteúdo de uma variável, ou um literal para o topo da pilha.
Elaborate	visitDeclaracaoDe Variavel	Aloca espaço na memória para as variáveis declaradas.
Assign	visitAtribuicao	Coloca o valor que está no topo da pilha na variável.

11.2. Descrição das estruturas de dados e algoritmos utilizados

11.2.1. Declaração de variável

```
@Override
public void visitDeclaracaoDeVariavel(DeclaracaoDeVariavel variableDeclaration) {

    ListaDeIds aux = variableDeclaration.listOfIds;
    while (aux != null) {
        aux = aux.next;
        varqtd++;
    }

    if (variableDeclaration.type instanceof TipoAgregado) {
        ((TipoAgregado) variableDeclaration.type).visit(this);
    } else {
        if (variableDeclaration.type instanceof TipoSimples) {
            ((TipoSimples) variableDeclaration.type).visit(this);
        }
    }
}
```

Figura 43: Método visitDeclaracaoDeVariavel() da classe Coder

Na figura 43 podemos ver o funcionamento do método de declaração de variáveis, que irá criar uma variável local “aux” do tipo ListaDeIds para receber a lista de ids recebida pelo método visitDeclaracaoDeVariavel(), logo após, será feita uma varredura em “aux” enquanto houver identificadores na mesma e será incrementada “varqtd”. O próximo passo é verificar se o tipo da variáveis declaradas são TipoSimples ou TipoAgregado e realizar a visitação.

```
@Override
public void visitVariavel(Variavel variable) {

    bd.append("\n0x").append(mem).append(": ").append("LOAD \t").append(variable.id.value);
    mem++;

    if (variable.selector != null) {
        variable.selector.visit(this);
    }
}
```

Figura 44: Método visitVariavel() da classe Coder

Na figura 44, o método visitVariavel() irá traduzir o código-fonte para o código objeto da respectiva variável, indicando a posição de memória, a instrução responsável por carregar a variável (LOAD) e o valor da variável, após isso, incrementa a posição de memória e salva tudo em “bd”, que é nossa pilha.

11.2.2. Comando Iterativo

```
@Override
public void visitIterativo(Iterativo iterativo) {

    bd.append("\n0x").append(mem).append(":t").append("JUMP\t").append("EVL").append(WhileCont);
    mem++;
    WhileAux = WhileCont; WhileCont++;
    bd.append("\n\nWHILE").append(WhileAux).append("\n");

    if (iterativo.command instanceof Atribuicao) {
        ((Atribuicao) iterativo.command).visit(this);
    } else if (iterativo.command instanceof ComandoComposto) {
        ((ComandoComposto) iterativo.command).visit(this);
    } else if (iterativo.command instanceof Iterativo) {
        ((Iterativo) iterativo.command).visit(this);
    } else if (iterativo.command instanceof Condicional) {
        ((Condicional) iterativo.command).visit(this);
    }

    bd.append("\n\nEVL").append(WhileAux).append("\n");

    if (iterativo.expression != null) {
        iterativo.expression.visit(this);
        mem++;
    }

    bd.append("\n0x").append(mem).append(":t").append("JUMPIF(1)\t").append("WHILE").append(WhileAux);
    mem++;
    WhileAux--;
}
```

Figura 45: Método visitIterativo() da classe Coder

O método visitIterativo() é responsável por verificar se o Iterativo é uma expressão e assim gerar o código correspondente (JUMPIF), caso contrário associá-lo a atribuição, condicional, iterativo ou comando composto e seu código correspondente (JUMP).

11.2.3. Comando condicional

```
@Override
public void visitCondicional(Condicional conditional) {
    if (conditional.expression != null) {
        conditional.expression.visit(this);
        bd.append("\n0x").append(mem).append(":\t").append("JUMPIF(0)\t").append("ELSE").append(ElseCont);
        mem++;
        ElseAux = ElseCont;
        ElseCont++;
    }

    if (conditional.command instanceof Atribuicao) {
        ((Atribuicao) conditional.command).visit(this);
    } else if (conditional.command instanceof ComandoComposto) {
        ((ComandoComposto) conditional.command).visit(this);
    } else if (conditional.command instanceof Iterativo) {
        ((Iterativo) conditional.command).visit(this);
    } else if (conditional.command instanceof Condicional) {
        ((Condicional) conditional.command).visit(this);
    }

    if (conditional.commandElse instanceof Atribuicao) {
        bd.append("\n0x").append(mem).append(":\t").append("JUMP\t").append("ELSE").append(ElseCont);
        mem++;
        bd.append("\n\nELSE").append(ElseCont-1).append(":\n");
        ElseAux++;
        ((Atribuicao) conditional.commandElse).visit(this);
    } else if (conditional.commandElse instanceof ComandoComposto) {
        bd.append("\n0x").append(mem).append(":\t").append("JUMP\t").append("ELSE").append(ElseCont);
        mem++;
        bd.append("\n\nELSE").append(ElseCont-1).append(":\n");
        ElseAux++;
        ((ComandoComposto) conditional.commandElse).visit(this);
    } else if (conditional.commandElse instanceof Iterativo) {
        bd.append("\n0x").append(mem).append(":\t").append("JUMP\t").append("ELSE").append(ElseCont);
        mem++;
        bd.append("\n\nELSE").append(ElseCont-1).append(":\n");
        ElseAux++;
        ((Iterativo) conditional.commandElse).visit(this);
    } else if (conditional.commandElse instanceof Condicional) {
        bd.append("\n0x").append(mem).append(":\t").append("JUMP\t").append("ELSE").append(ElseCont);
        mem++;
        bd.append("\n\nELSE").append(ElseCont-1).append(":\n");
        ElseAux++;
        ((Condicional) conditional.commandElse).visit(this);
    }
}
}
```

Figura 46: método visitCondicional() da classe Coder

O método visitCondicional() irá verificar se o seu argumento condicional é uma atribuição, comando composto ou condicional e realizar a respectiva tradução para seu código equivalente (JUMP/JUMPIF + ELSE).

11.2.4. Literal

```
@Override
public void visitLiteral(Literal literal) {
    bd.append("\n0x").append(mem).append(": ").append("LOADL \t").append(literal.name.value);
    mem++;
}
```

Figura 47: Método visitLiteral() da classe Coder

O método visitLiteral() realiza a tradução do código-fonte para o código-objeto (posição de memória + LOADL + valor) e salva tudo em “bd”.

11.2.5. Comando atribuição.

```
@Override
public void visitAtribuicao(Atribuicao becomes) {

    if (becomes.expression != null) {
        becomes.expression.visit(this);
    }

    if (becomes.variable != null) {
        bd.append("\n0x").append(mem).append(": ").append("STORE \t").append(becomes.variable.id.value);
        mem++;
    }

}
```

Figura 48: Método visitAtribuicao() da classe Coder

O método visitAtribuicao() fará a verificação do que está sendo atribuído, caso seja uma expressão, será feita a visitação na expressão, caso seja uma variável, será realizada a tradução do código-fonte para o código-objeto (posição de memória + STORE + valor).

11.2.6. Termo

```
@Override
public void visitTermo(Termo term) {
    Termo aux = term;

    while (aux != null) {
        if (aux.factor != null) { //Confere o primeiro fator

            if (aux.factor instanceof Variavel) {
                ((Variavel) aux.factor).visit(this);
                term.type = aux.factor.type;
            } else if (aux.factor instanceof Literal) {
                ((Literal) aux.factor).visit(this);
                term.type = aux.factor.type;
            } else if (aux.factor instanceof Expressao) {
                ((Expressao) aux.factor).visit(this);
                term.type = aux.factor.type;
            }

            if (aux.operator != null)
                switch (aux.operator.kind) {
                    case Token.DIV: //Divisão não pode conter operandos booleanos;
                        bd.append("\n0x").append(mem).append(":\\t").append("CALL\\tdiv");
                        mem++;
                        break;

                    case Token.MULT:
                        bd.append("\n0x").append(mem).append(":\\t").append("CALL\\tmult");
                        mem++;
                        break;

                    case Token.AND:
                        bd.append("\n0x").append(mem).append(":\\t").append("CALL\\tand");
                        mem++;
                        break;
                }
            aux = aux.next;
        }
    }
}
```

Figura 49: Método visitTermo() da classe Coder

O método visitTermo() irá realizar as devidas verificações (Variável, Expressão e Literal) e gerar o código correspondente aos operadores (div, mult e and).

11.3. Exemplos de tradução com trechos representativos do programas-fonte.

11.3.1. Comando condicional - IF.

Dado o código de testes abaixo, a expressão IF, será traduzida da seguinte forma:

Programa	Comando IF em TAM
<pre> program programa; var loona, edjair, mateus: integer; var kpop: array [1 ~ 4] of integer; begin loona := 0; edjair := loona + 1; mateus := 2; if ((mateus = 2) and (edjair < 2)) then begin loona := 1; kpop[1] := 3 * 5; end else kpop[1] := edjair; end. </pre>	<pre> 0x0: PUSH 4 0x1: LOADL 0 0x2: STORE loona 0x3: LOAD loona 0x4: LOADL 1 0x5: CALL add 0x6: STORE edjair 0x7: LOADL 2 0x8: STORE mateus 0x9: LOAD mateus 0x10: LOADL 2 0x11: CALL eq 0x12: LOAD edjair 0x13: LOADL 2 0x14: CALL lt 0x15: CALL and 0x16: JUMPIF(0) ELSE1 0x17: LOADL 1 0x18: STORE loona 0x19: LOADL 3 0x20: LOADL 5 0x21: CALL mult 0x22: STORE kpop 0x23: JUMP ELSE2 ELSE1: 0x24: LOAD edjair 0x25: STORE kpop ELSE2: 0x26: POP 4 0x27: HALT </pre>

11.3.2. Comando iterativo

Programa	Programa em TAM
<pre> program programa; var edjamigo : integer; begin edjamigo := 0; while (edjamigo < 2) do edjamigo := edjamigo + 1; end. </pre>	<pre> 0x0: PUSH 1 0x1: LOADL 0 0x2: STORE edjamigo 0x3: JUMP EVL1 WHILE1: 0x4: LOAD edjamigo 0x5: LOADL 1 </pre>

.	0x6: CALL add 0x7: STORE edjamigo EVL1: 0x8: LOAD edjamigo 0x9: LOADL 2 0x10: CALL lt 0x12: JUMPIF(1) WHILE1 END: 0x13: POP 1 0x14: HALT
---	---

11.3.3. Array multidimensional.

Programa	Programa em TAM
<pre> program programa; var marcus,aux : integer; var multarray : array [0 ~ 2] of array [1 ~ 3] of array [2 ~ 5] of integer; begin marcus := 0; aux := marcus; while (marcus <= 2) do begin multarray[marcus][2][2] := aux; marcus := marcus + 1; end; end. </pre>	0x0: PUSH 3 0x1: LOADL 0 0x2: STORE marcus 0x3: LOAD marcus 0x4: STORE aux 0x5: JUMP EVL1 WHILE1: 0x6: LOAD aux 0x7: STORE multarray 0x8: LOAD marcus 0x9: LOADL 1 0x10: CALL add 0x11: STORE marcus EVL1: 0x12: LOAD marcus 0x13: LOADL 2 0x14: CALL ltoe 0x16: JUMPIF(1) WHILE1 END: 0x17: POP 3 0x18: HALT

12. Manual de compilação

12.1. Instruções detalhadas sobre como obter uma versão executável do compilador a partir dos programas-fonte fornecidos.

O projeto foi desenvolvido utilizando a IDE NetBeans versão 8.0.1, para abrir o projeto pela IDE, deve-se selecionar Arquivo->Abrir Projeto, e selecionar o projeto.

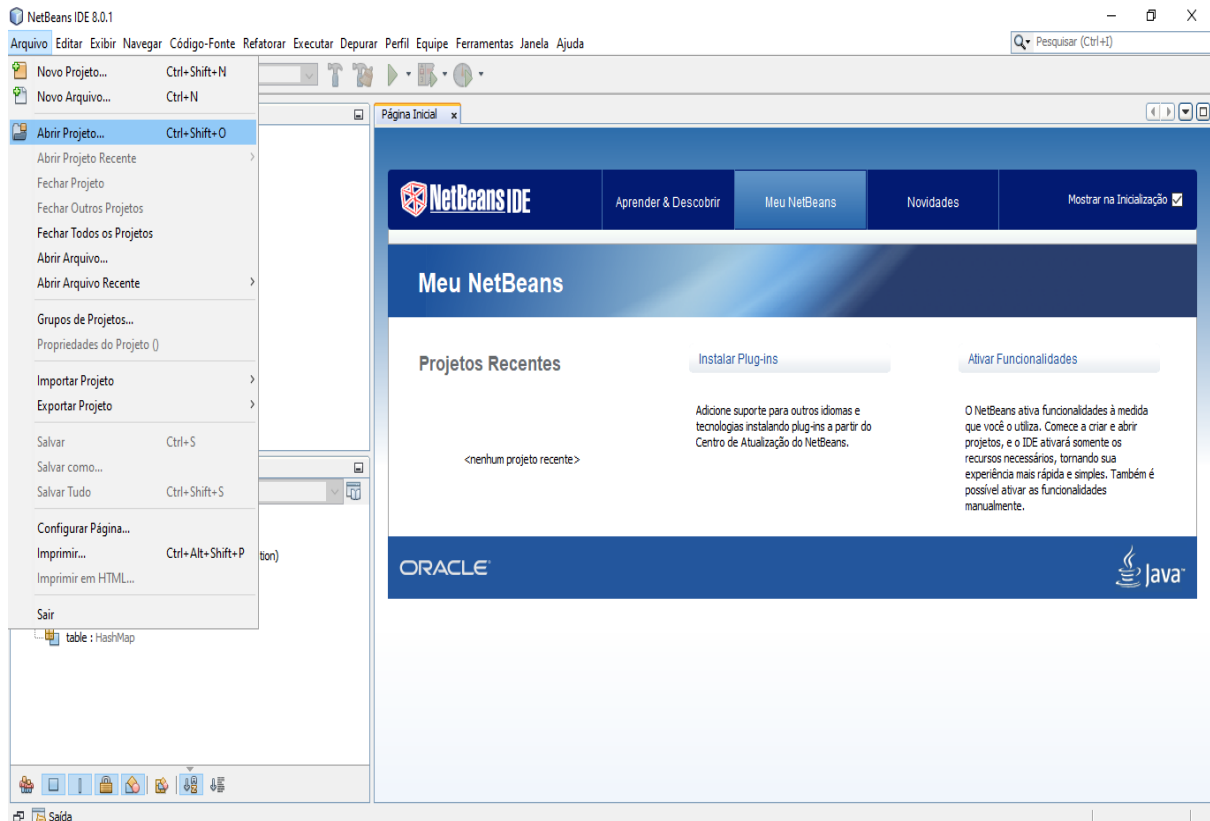


Figura 50: Método para abrir o projeto do compilador na IDE

Após aberto o compilador pode-se gerar um executável .jar, para isso clique com o botão direito em cima do projeto no canto esquerdo da IDE, logo após clique em Limpar e Construir.

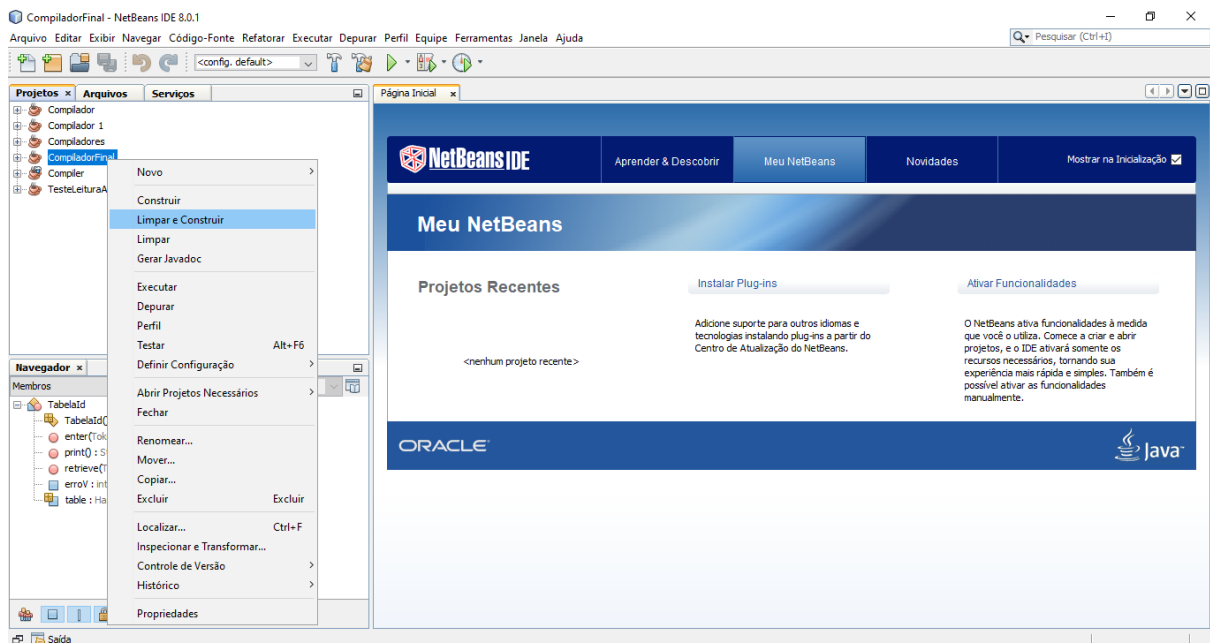


Figura 51: Exemplo de geração do .jar

O NetBeans irá criar o executável no diretório do projeto, na pasta “dist”.

12.2. Descrições completas sobre scripts, ambientes de desenvolvimento, versões, plataforma, sistema operacional, pacotes e outros recursos necessários para a correta obtenção da versão executável.

Uma versão executável pode ser obtida de maneira simples sem a utilização do terminal utilizando o NetBeans, testes foram realizados utilizando a versão 8.0.1 da IDE, para o sistema operacional Windows 10.

Foi necessário instalar dois pacotes Java antes do NetBeans. O primeiro pacote, chamado JRE (Java Runtime Environment), é responsável por permitir que aplicações escritas em Java sejam executadas. O segundo pacote, chamado JDK (Java Development Kit), é o pacote que contém toda a infraestrutura necessária para o desenvolvimento de aplicações.

Como nenhum instalador oficial é fornecido para o Netbeans, é necessário baixar os arquivos “Binaries” e descompactá-los. É conveniente colocar a pasta descompactada na pasta “Arquivos de Programas” do seu disco local.

12.3. Manual de instalação

O compilador pode ser usado de maneira portátil, sem necessidade de instalação, utilizando apenas o compilador gerado em formato .jar.

12.4. Manual de utilização

12.4.1. Compilação do mini pascal

O arquivo fonte pode ser escrito utilizando qualquer editor de texto, como o bloco de notas do Windows 10, Notepad ++, Visual Code, sublime, entre outros.

Para facilitar a utilização do usuário foi criada uma interface gráfica que tornará os comandos mais intuitivos. A interface será exibida ao executar o compilador e será possível carregar um arquivo de texto (.pas) e logo após carregado exibirá se houve sucesso ou não.

13. Instruções de execução

1. Na pasta “Compilador”, execute o arquivo “Compilador.bat” e aguarde o início da execução;
2. Não feche o terminal: os eventuais erros de compilação são mostrados através do prompt de comando;

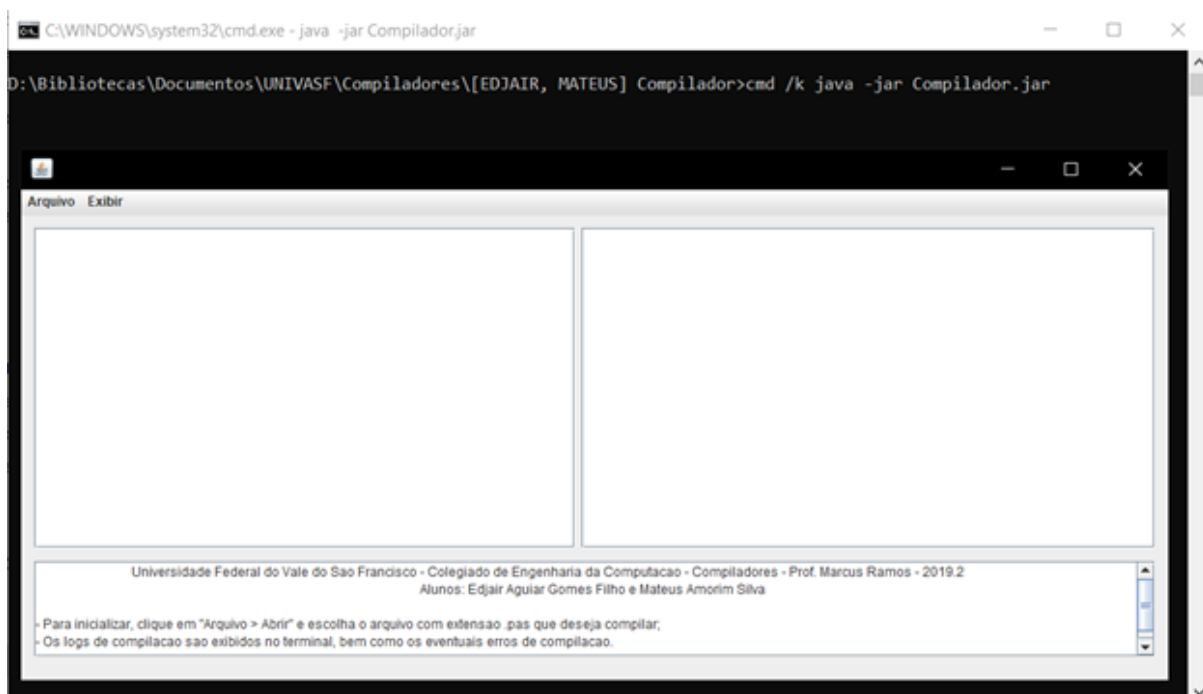


Figura 52: Tela inicial do compilador

3. Através da barra horizontal na parte superior da interface, clique em “Arquivo > Abrir” e selecione o arquivo de extensão “.pas” que deseja compilar;

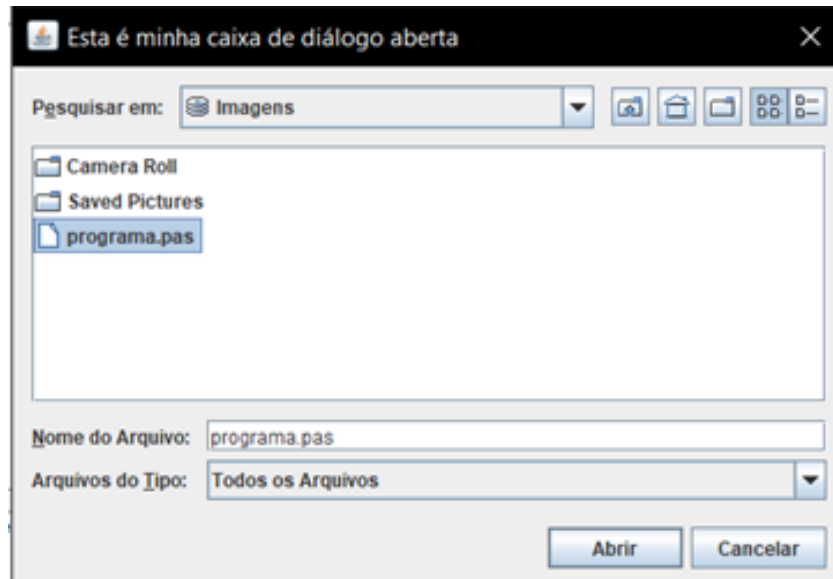


Figura 53: Selecionando o arquivo fonte para ser compilado

4. O corpo do programa-fonte selecionado deverá ser exibido na tela esquerda.

13.1 Compilação Bem-Sucedida:

1. Caso a compilação seja bem-sucedida, uma tela de confirmação da geração de código é exibida;

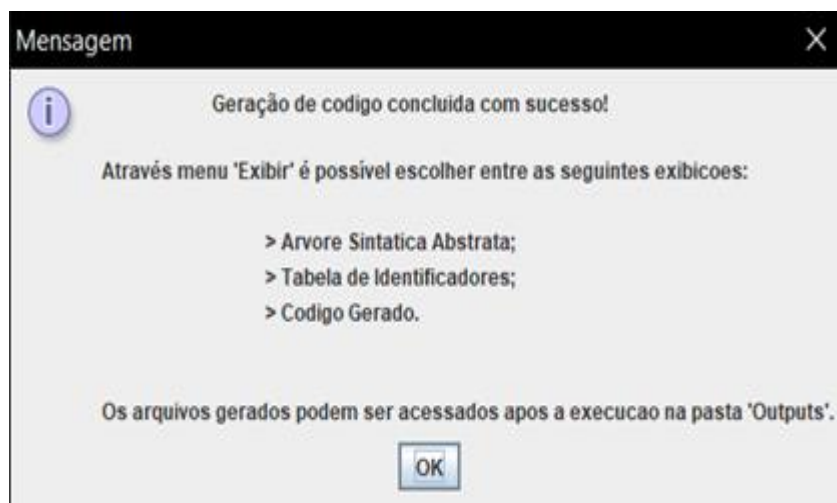


Figura 54: Abertura do arquivo .pas pelo compilador

2. Na tela à direita, será exibido o código gerado para a máquina TAM;

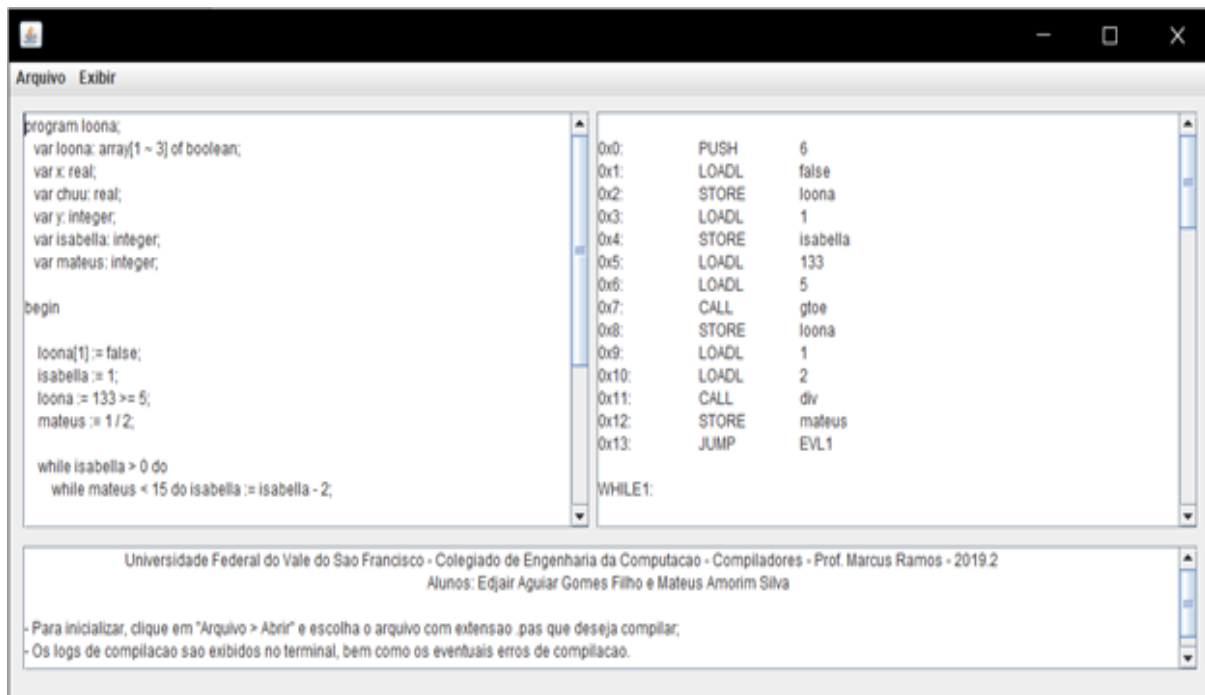


Figura 55: Exemplo de código gerado (código base -> código objeto).

3. Através da barra horizontal na parte superior, é possível selecionar a exibição da tela à direita. Ao clicar-se em “Exibir”, é possível alternar entre a exibição do código gerado, da árvore sintática abstrata ou tabela de identificadores;

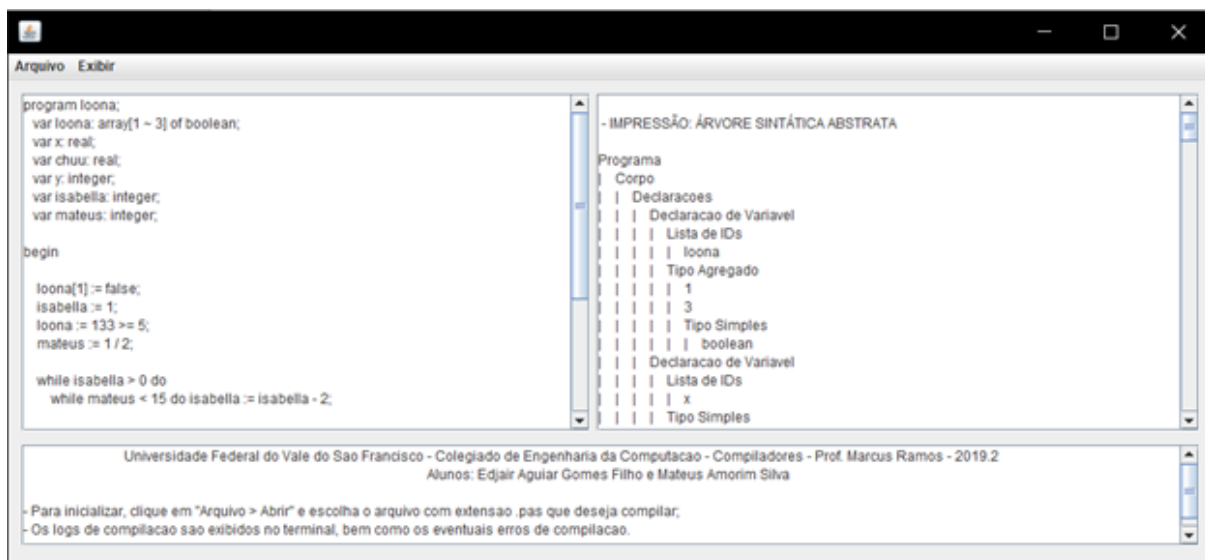


Figura 56: AST gerada do compilador

4. Os arquivos gerados também podem ser visualizados posteriormente no diretório “Outputs”, que se encontra na mesma pasta que o arquivo executável;

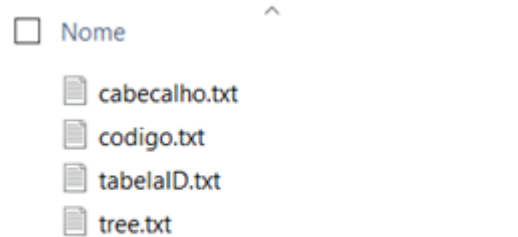


Figura 57: Arquivos gerados

13.2 Compilação Com Erros:

1. Caso o arquivo selecionado contenha erros, é exibida uma mensagem alertando o tipo de erro encontrado: erro sintático ou erro de contexto;

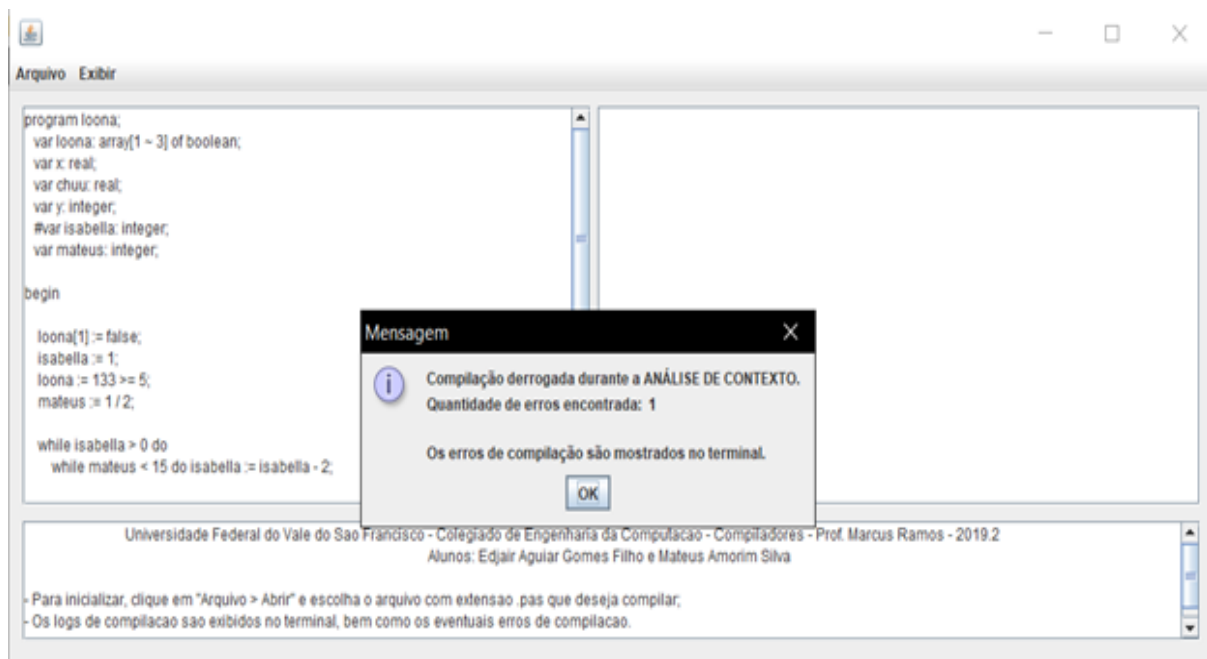
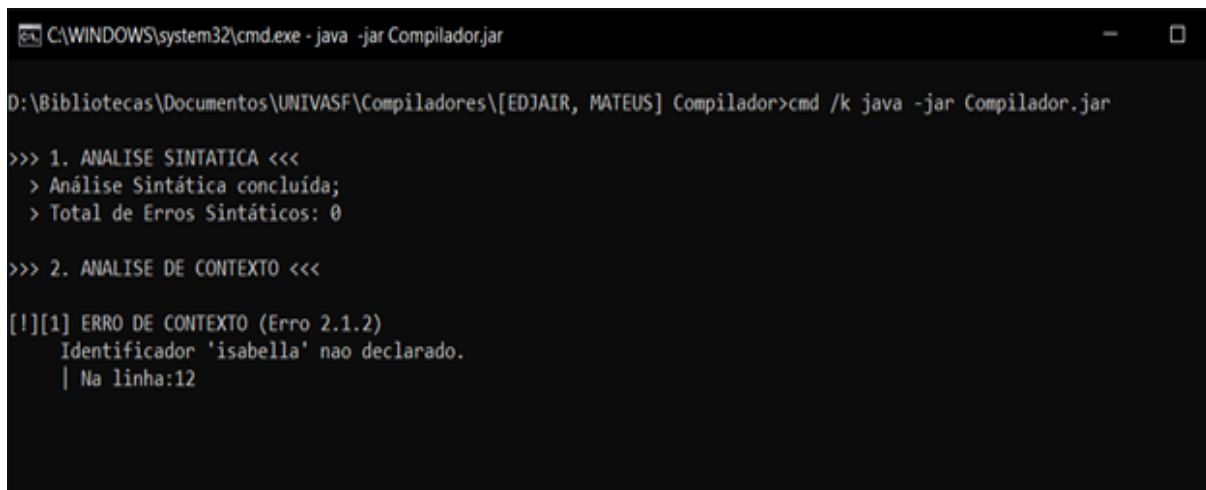


Figura 58: Tentativa de compilação com código com erros

2. A especificação detalhada do erro encontrado é mostrada no Prompt de Comando;



```
C:\WINDOWS\system32\cmd.exe - java -jar Compilador.jar

D:\Bibliotecas\Documentos\UNIVASF\Compiladores\[EDJAIR, MATEUS] Compilador>cmd /k java -jar Compilador.jar

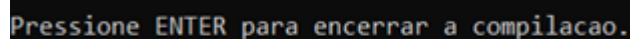
>>> 1. ANALISE SINTATICA <<<
> Análise Sintática concluída;
> Total de Erros Sintáticos: 0

>>> 2. ANALISE DE CONTEXTO <<<

[!][1] ERRO DE CONTEXTO (Erro 2.1.2)
      Identificador 'isabella' nao declarado.
      | Na linha:12
```

Figura 59: Exibição de erros no prompt.

3. Não são gerados arquivos de código, árvore sintática ou tabela de identificadores para códigos com erros de compilação;
4. Para encerrar a execução do compilador, basta o usuário pressionar ENTER no prompt de comando.



Pressione ENTER para encerrar a compilacao.

Figura 60: Instrução para encerrar o programa

Recomendamos a execução do compilar através do arquivo .bat que está junto ao .jar, pois o mesmo irá garantir que o prompt de comando esteja aberto enquanto o compilador está em execução, permitindo a aferição dos erros ocorridos no compilador.

14. Mensagens de erro

A implementação do tratamento de erros foi realizado criando uma classe denominado Error que extends a classe de erros do java RuntimeException.

Conforme dito anteriormente, foi criada uma classe chamada Erros, que irá verificar as condições, caso seja verdadeiras

Dessa forma, o compilador ao identificar algum erro em alguma etapa da compilação exibe o erro realizando um throw como pode ser observado abaixo.

14.1 Exibição dos erros.

Segue exemplos de possíveis erros identificados pelo compilador, de forma geral as mensagens.

14.1.1 Erro léxico.

Programa

```
program programa;  
  var cont, aux : integer;  
  var atribui : boolean;  
begin  
  atribui := ¬true;  
  aux := count;  
end.
```

>>> 1. ANALISE SINTATICA <<<

[!][1] ERRO SINTATICO (Erro 1.4)

Fator inválido.

> Fator espera receber ATRIBUIÇÃO (ID), LITERAL ou EXPRESSÃO ENTRE PARENTÊSES

| Na linha: 5

| Coluna: 17

Pressione ENTER no prompt para encerrar a compilação.

Para solucionar este erro, deve-se remover o caractere '¬', pois este não faz parte da linguagem.

14.1.2 Erro de sintaxe.

Programa
<pre>>>> 1. ANALISE SINTATICA <<< [!][1] ERRO SINTÁTICO (Erro 1.1) Token inesperado. Esperava encontrar: '~' Na linha: 3 Coluna: 21</pre>

14.1.3 Erro de contexto.

Programa
<pre>program programa; var count,aux : integer; var recebe : boolean; var ar : array [0 .. 2] of array [1 .. 3] of array [2 .. 5] of integer; begin recebe := 0; aux := count; while (count <= 2) do begin ar[count][2][2] := aux; count := count + 1; end; end .</pre>

Este erro ocorre quando está sendo atribuído a uma variável, uma expressão com um tipo diferente do declarado anteriormente.

Programa
<pre>program programa; var count,aux : integer; var ar : array [0 .. 2] of array [1 .. 3] of array [2 .. 5] of integer; begin aux := count; while (count + 2)</pre>

```
do
begin
ar[count][2][2] := aux;
count := count + 1;
end;
end
.
```

Para solucionar este erro deve colocar uma expressão booleana no comando iterativo.

Programa

```
program programa;
  var count,aux : integer;
  var ar : array [ 0 .. 2] of array [ 1 .. 3 ] of array [ 2 .. 5 ] of integer;
begin
  aux := count;
  if (count + 2)
  then
  begin
```

Para solucionar tal erro deve colocar uma expressão booleana no comando condicional.

Programa

```
program programa;
  var count,aux : integer;
  var ar : array [ 0 .. 2] of array [ 1 .. 3 ] of array [ 2 .. 5 ] of integer;
begin
  aux := count + true;
  if (count > 2)
  then
  begin
    ar[count][2][2] := aux;
  end;
end
.
```

O operador "+", não suporta tipos booleanos, para corrigir este erro deve-se

utilizar uma variável ou literal compatível com o operador e com a variável count.

Programa
<pre>program programa; var count : integer; begin aux := 1; end .</pre>

Neste caso foi utilizado uma variável que não foi declarada, para solucionar este erro deve-se utilizar uma variável declarada anteriormente.

15. Conclusões finais

15.1 Avaliação do projeto de criação de um compilador

Este projeto teve como principal função mostrar a grande utilidade da teoria de compiladores na vida de um profissional de computação, pois a partir destes conhecimentos o profissional terá plena consciência do que está fazendo ao trabalhar com qualquer linguagem, já que ele conhece o funcionamento básico da linguagem. A produção deste compilador foi dividido em quatro fases, análise léxica, análise sintática, análise de contexto e geração de código.

A análise léxica nos mostrou como feita a análise do código fonte propriamente dito, ou seja, a escrita do programa, logo vemos a importância de sabermos quais os símbolos da linguagem que estamos programando, e a importância de tornar os símbolos da linguagem determinísticos para que a linguagem seja eficiente.

Na análise sintática vemos como é feita a verificação das regras da linguagem, ou seja, a gramática da linguagem. Logo vemos a importância de entender as regras para a produção de uma linguagem, para que quando for necessário fazer uso da documentação de uma linguagem de programação, qualquer que seja ela, já teremos o conhecimento necessário para uma consulta mais eficiente.

A análise de contexto foi umas das fases mais importante para o aprendizado, pois ela esclareceu diversas dúvidas quanto ao contexto de comandos, utilização de variáveis, e como funciona a análise semântica de uma linguagem de programação. Também foi visto que como é realizado a análise de erros semânticos de uma linguagem de programação, que uma das coisas mais comum no meio da programação, já que os erros retornados aos usuários são de extrema importância para que o código seja funcional e coerente com as regras da linguagem.

Por fim temos a geração de código, outra fase bastante importante, pois é a partir dela que aprendemos como é feita a alocação de memória para as variáveis, e como é feita a manipulação dessa memória durante a chamada de comandos. Esta fase foi especialmente importante devido ao esclarecimento do funcionamento da alocação de memória de variáveis do tipo agregado, ou seja, array, já que é um tipo de dado muito utilizado.

A partir do aprendizado desta fase vimos é possível programar de forma mais eficiente sabendo como é realizada a tradução dos comando de alto nível para o baixo nível.

15.2 Descrição das principais dificuldades encontradas e encaminhamentos correspondentes

Entre as maiores dificuldades encontradas está o entendimento do padrão de projeto visitor, utilizado durante a impressão da árvore, fase de análise de contexto e geração de código, devido a forma como este percorre a árvore, visitando todos os nós transitando entre diversas classes durante a sua execução, mas após um estudo mais aprofundado e tirar dúvidas com o professor Marcus Ramos, a sua implementação foi realizada com êxito.

Outra dificuldade encontrada está relacionado com o endereçamento de variáveis na memória durante a fase de geração de código, mas especificamente para arrays com mais de uma dimensão, pois ao entendermos o processo de cálculo do endereço ao qual o usuário deseja acessar, houve uma dificuldade em como realizar este cálculo utilizando o TAM, visto que os valores das expressões necessários para o cálculo do endereço estarão na pilha, mas, após testes mesa e vários esboços de funções, foi chegado em um *template*, e o problema foi solucionado.

15.3 Críticas e sugestões

Acreditamos que o livro adotado não é intuitivo o suficiente para a matéria, devido ao principal fato de existir apenas uma edição disponível. Sugere-se a adoção de um livro com uma linguagem mais simples ou o desenvolvimento de um material didático para uso regular.