

Documentação do compilador

Aluno: Gustavo Souza Martins

Sumário

1. Descrição da linguagem.....	4
1.1 Relação e estrutura de separadores.....	4
1.2 Sintaxe livre de contexto.....	4
1.3 Sintaxe dependente de contexto.....	5
1.4 Semântica:.....	5
2. Descrição geral da arquitetura do compilador.....	5
3. Fundamentação teórica e técnicas empregadas na análise sintática.....	6
4. Análise Léxica.....	6
4.1 Técnicas utilizadas.....	6
4.2 Relação de Tokens.....	7
4.3 Gramática Léxica.....	7
5. Análise Sintática.....	8
5.1 Técnicas utilizadas.....	8
5.2 Gramática Sintática.....	9
6. Descrições das principais estruturas de dados utilizadas.....	10
7. Montagem e impressão da árvore de sintaxe abstrata.....	11
7.1 Descrição do processo de montagem da AST.....	11
7.2 Descrição do processo de impressão da AST.....	11
7.3 Exemplos de impressões na interface gráfica.....	12
8. Análise de contexto.....	14
8.1 Descrição da fase de identificação.....	14
8.2 Descrição da fase de verificação.....	14
8.3 Exemplos de entradas e saídas na análise de contexto.....	14
9. Geração de código.....	16
9.1 Funções de código.....	16
9.2 Descrição das estruturas de dados e algoritmos utilizados.....	18
9.3 Exemplos de traduções com trechos representativos do programa-fonte.....	18
10. Linguagem objeto.....	20
10.1 Relação de instruções.....	20

10.2 Sintaxe.....	20
10.3 Semântica.....	20
11. Ambiente de execução.....	21
11.1 Tipo de máquina usada.....	21
11.2 Avaliação de expressões.....	21
11.3 Formas de alocação de memória empregadas.....	21
12. Manual de compilação.....	21
12.1 Compilação e execução em sistemas Linux.....	21
12.2 Compilação e execução no Windows.....	22
12.3 Observações ao professor.....	22
13. Manual de utilização.....	22
13.1 Como utilizar o programa.....	23
13.2 Extensões dos arquivos de entrada e saída do compilador.....	24
13.3 Mensagens de erro do compilador.....	24
14. Conclusões finais.....	25

1. Descrição da linguagem

1.1 Relação e estrutura de separadores

Os separadores da linguagem são os símbolos que são ignorados pelo compilador no processo de análise. Esses símbolos são: *espaço*, e, '#', para dar início a um comentário em uma linha. Um comentário nessa linguagem funciona da seguinte forma, a partir do símbolo '#' todos os próximos caracteres serão ignorados pelo compilador, até que a próxima linha comece.

1.2 Sintaxe livre de contexto

A sintaxe livre de contexto se dá pelas regras a seguir. Observamos que partindo da cadeia de símbolos não terminais "<program>" podemos usar as demais regras para obter uma cadeia válida escrita na nossa linguagem.

```

<programa> ::= program <id> ; <corpo>
<id> ::= <letra> | <id> <letra> | <id> <digito>
<letra> ::= a | b | ... | z
<digito> ::= 0 | 1 | ... | 9
<corpo> ::= <declarações> <comando composto>
<declarações> ::= <declaração> ; | <declarações> <declaração> ; | <vazio>
<declaração> ::= var <id> : <tipo>
<vazio> ::= ε
<tipo> ::= <tipo-simples>
<tipo-simples> ::= integer | boolean
<comando-composto> ::= begin <lista-de-comandos> end
<lista-de-comandos> ::= <comando> ; | <lista-de-comandos> <comando> ; | <vazio>
<comando> ::= <atribuição> | <condicional> | <iterativo> | <comando-composto>
<atribuição> ::= <variável> := <expressão>
<condicional> ::= if <expressão> then <comando> ( else <comando> | <vazio> )
<iterativo> ::= while <expressão> do <comando>
<variável> ::= <id>
<expressão> ::= <expressão-simples> | <expressão-simples> <op-rel> <expressão-simples>
<expressão-simples> ::= <expressão-simples> <op-ad> <termo> | <termo>
<op-rel> ::= < > | =
<op-ad> ::= + | - | or
<termo> ::= <termo> <op-mul> <fator> | <fator>
<op-mul> ::= * | / | and
<fator> ::= <variável> | <literal> | ( <expressão> )
<literal> ::= <boolean-literal> | <integer-literal> | <float-literal>
<boolean-literal> ::= true | false
<integer-literal> ::= <digito> | <integer-literal> <digito>
<float-literal> ::= <integer-literal> . <interger-literal> | <integer-literal> . | . <interger-literal>

```

1.3 Sintaxe dependente de contexto

A sintaxe dependente de contexto se refere às regras de contexto presentes na linguagem fonte. Sendo elas:

1. Não possível declarar duas variáveis com o mesmo identificador, por mais que o tipo delas sejam diferentes.
2. Não é possível atribuir uma expressão cujo tipo seja diferente da variável que a está recebendo, ou seja, uma variável só pode armazenar um valor do mesmo tipo de sua declaração.
3. As expressões condicionais, aquelas que estão presentes nos comandos de *if* e *while* só podem ser do tipo *booleano*.
4. Não é possível realizar a chamada de uma variável que não foi, previamente, declarada.

1.4 Semântica:

A semântica da linguagem se refere ao significado dos símbolos presentes, sendo eles:

- **program** se refere ao começo do programa.
- **var** representa o começo da declaração de uma variável.
- **integer** e **boolean** se referem aos tipos atômicos definidos na linguagem, sendo eles, respectivamente inteiro e booleano.
- **“begin ... end”** representa um bloco de comandos sequenciais.
- Para realizar um comando de atribuição deve-se usar o identificador da variável seguido por “:” e uma expressão.
- Um comando condicional é definido por **“if ... then ... else”**, podendo ou não apresentar a palavra chave **else**. É lido na forma: *se (expressão) então (comando₁) senão (comando₂)*, ou seja, se a *expressão* for verdadeira é executado o *comando₁*, caso contrário, é executado o *comando₂*. Na ausência do símbolo **else**, nada é feito no caso da *expressão* ser falsa.
- Um comando iterativo é definido por **“while ... do ...”** Interpretamos ele como: *enquanto (expressão) faça (comando)*, ou seja, enquanto a *expressão* for verdadeira, execute *comando*.
- As expressões na linguagem fonte são definidas da forma: *(operante) (operador) (operante)*, ou seja, um operador sempre vai estar no meio de dois operantes. Esses operadores podem ser dos tipos relacional, aditivo ou multiplicativo.

2. Descrição geral da arquitetura do compilador

O compilador apresentado aqui está dividido em três passos/etapas. Partindo da Análise Sintática seguindo para a Análise de Contexto e por fim chegando na Geração de Código.

No primeiro passo o compilador analisa os símbolos que recebeu como entrada, e classifica se os mesmos pertencem à linguagem fonte ou não, chamamos essa etapa de análise léxica. O compilador analisa também a ordem que esses símbolos estão dispostos, se a mesma é válida, diante das regras de derivação da linguagem. Ao final dessa etapa é gerada uma árvore AST, que representa as derivações de regras utilizadas para poder chegar no código fonte. Essa árvore será, posteriormente, decorada com a análise de contexto, após as sub etapas de identificação e verificação.

Na Análise de Contexto é realizada as sub etapas Identificação e Verificação. Na primeira, é observada a declaração de variáveis, o bloco onde estão sendo declaradas e a associação com as chamadas das variáveis, ou seja, é analisado o escopo em que cada variável é declarada. Na segunda sub etapa é feita a verificação dos tipos de variáveis e também do tipo de retorno das expressões. É checado, por exemplo, se a chamada de uma identificação é válida baseada em seu uso. Também é estimado o valor resultante de expressões. Ao final da Análise de Contexto teremos uma AST decorada, com as informações necessárias para validar o código.

Por último, temos a fase da geração de código, após validar todo o programa fonte, podemos aplicar uma tradução direta para as instruções presentes no mesmo. Para isso, utilizamos de *Code functions* e *code templates*, que auxiliam na implementação em código dessa tradução. É nessa etapa que fazemos a alocação estática das variáveis declaradas pelo programa fonte.

3. Fundamentação teórica e técnicas empregadas na análise sintática

A análise sintática tem uma mini etapa que a antecede que é a análise léxica. Na análise léxica checamos os símbolos presentes no código de entrada, e se os mesmos pertencem a linguagem. Essa verificação é feita baseando-se numa relação de tokens, que é apresentada mais pra frente, juntamente com a gramática léxica, que dita como esses tokens são formados.

Após a análise léxica é feita a análise sintática, que vai ler os símbolos terminais e dizer se os mesmos estão colocados na ordem correta, seguindo derivações de regras da Sintaxe Livre de Contexto. A análise sintática ocorre seguindo o método recursivo descendente, ou seja, é montada a árvore de sintaxe de cima para baixo, que é uma forma determinística de analisar as regras de derivações. Esse determinismo é também garantido pelo fator LL(k). É necessário que a linguagem seja LL(1) para facilitar o processo de análise sintática. A determinação de uma linguagem, se ela é LL(1) ou não, é feita utilizando as funções auxiliares *first*, *lookahead* e *follow*.

4. Analise Léxica

4.1 Técnicas utilizadas

A implementação do analisador léxico se deu utilizando duas classes: *Scanner* e *Token*. A classe *Token* representa os símbolos da linguagem, é nela que estão definidas

as palavras-chave da linguagem fonte. Já a classe *Scanner* representa propriamente o processo de análise léxica, ao receber uma cadeia de entrada o *Scanner* analisa os símbolos nela presentes e os classifica entre Separador ou Token(palavra-chave).

Um exemplo do funcionamento do analisador léxico é: ao receber uma cadeia de entrada contendo o símbolo “begin”, esse símbolo será analisado e classificado como *Token.BEGIN*, apresentando também a linha e coluna em que está presente no arquivo fonte. Porém se receber um símbolo inválido na cadeia de entrada, como por exemplo “&”, o analisador léxico vai classificá-lo como *Token.ERROR*.

4.2 Relação de Tokens

Os tokens da nossa linguagem, ou também chamados símbolos terminais estão dispostos a seguir. Esses símbolos indicam palavras-chave da linguagem.

id	if	;
integer-literal	then	(
integer	else)
boolean	var	+
float	and	-
true	or	*
false	while	/
begin	do	>
end	:=	<
	:	=

4.3 Gramática Léxica

A gramática a seguir dita como são montados, inicialmente, os tokens da linguagem. Podemos explicar a expressão regular abaixo como: “Um *Token* está sempre separado de outro por um *Separador*. Um arquivo fonte contendo o código da linguagem pode começar com zero ou mais *Separadores*, e também pode terminar da mesma forma.”.

As regras representam a construção de tokens da linguagem, em ordem, dizemos: um identificador é formado de uma sequência e pelo menos uma letra seguida por letras e/ou números. Um inteiro na linguagem é formado por uma sequência de pelo menos um dígito seguido por outros números. Um *float* na linguagem é representado por zero ou mais números seguidos por um ponto que é seguido por zero ou mais dígitos. Por fim, as outras regras representam a construção de tokens com um único caractere, podemos chamar esses tokens de operadores.

$$((\text{Separator}^*) \text{Token})^* \text{Separator}$$

Token ::=

letra (letra | digito)^{*} | digito digito^{*} | (= | ε) | ; | (|) + | - | * | / | < | > | = |

Separador ::=

Gráfico^{*} eol | space | eol

5. Analise Sintática

5.1 Técnicas utilizadas

Para implementar o analisador sintático foi desenvolvida a classe *Parser*, que recebe os Tokens do analisador léxico e define se os mesmos estão postos na ordem correta, ou seja, se a cadeia de entrada é realmente uma cadeia oriunda de derivações das regras da linguagem fonte. Essa análise é feita utilizando o método recursivo descendente, ou seja, a árvore de sintaxe é montada de cima para baixo, utilizando derivações mais a esquerda.

Um exemplo do funcionamento do analisador sintático pode ser visto em:

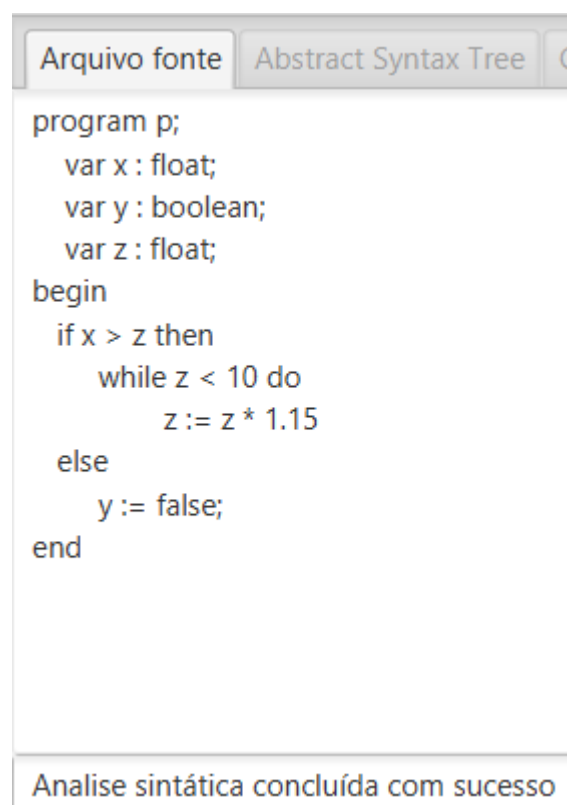


Imagem 01: execução com sucesso do analisador sintático

Onde observamos que, dada uma cadeia de entrada, o compilador analisa a ordem dos símbolos presentes, e se essa ordem for válida, diante as regras de derivação da linguagem, o analisador sintático termina sua análise sem apresentar erros. Porém, se

houver algum erro na cadeia de entrada, o analisador sintático terminará a análise pontuando o erro localizado.

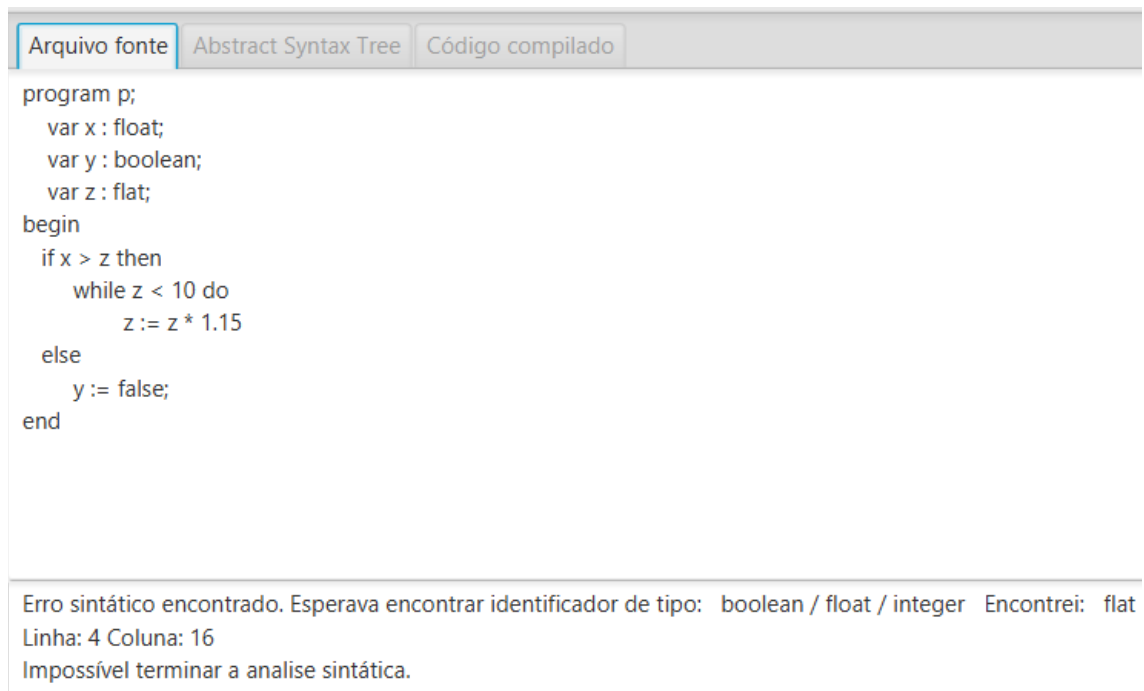


Imagem 02: execução sem sucesso do analisador sintático

5.2 Gramática Sintática

Buscando transformar a linguagem mostrada no tópico 1.2 em LL(1), visto que a anterior não era, principalmente por apresentar recursões à esquerda. O processo de transformação em LL(1) foi dado por exclusão das recursões à esquerda, fatorações de símbolos à esquerda e substituições. Foram realizadas algumas substituições nas regras buscando simplificar a implementação do analisador sintático em código. Por fim, a linguagem obtida, LL(1), é dada pelas regras:

```

<programa> ::= program <id> ; <corpo>
<corpo> ::= <declaração> <comando-composto>
<declaração> ::= var <id> : <tipo> ; ( var <id> : <tipo> ; )* | ε
<tipo> ::= integer | boolean
<comando-composto> ::= begin <lista-de-comandos> end
<lista-de-comandos> ::= <comando>; ( <comando>; )* | ε
<comando> ::= <id> := <expressão>
               | if <expressão> then <comando> ( else <comando> | ε )
               | while <expressão> do <comando>
               | <comando-composto>
<expressão> ::= <expressão-simples> ( <op-rel> <expressão-simples> )*
<expressão-simples> ::= <termo> ( <op-ad> <termo> )*
<op-rel> ::= < > | =

```

$\langle \text{op-ad} \rangle ::= + \mid - \mid \text{or}$
 $\langle \text{termo} \rangle ::= \langle \text{fator} \rangle (\langle \text{op-mul} \rangle \langle \text{fator} \rangle)^*$
 $\langle \text{op-mul} \rangle ::= * \mid / \mid \text{and}$
 $\langle \text{fator} \rangle ::= \langle \text{id} \rangle \mid \langle \text{literal} \rangle \mid (\langle \text{expressão} \rangle)$
 $\langle \text{literal} \rangle ::= \langle \text{boolean-literal} \rangle \mid \langle \text{integer-literal} \rangle \mid \langle \text{float-literal} \rangle$
 $\langle \text{boolean-literal} \rangle ::= \text{true} \mid \text{false}$

6.Descrições das principais estruturas de dados utilizadas

Para implementar o compilador foram construídas algumas estruturas de dados em etapas importantes do processo de compilação.

Presente no início do processo de compilação temos a presença dos Tokens, que é uma estrutura construída com o objetivo de representar os símbolos(palavras-chave) da linguagem fonte, essa estrutura de dados é composta por um tipo enumerado para representar os diversos Tokens da linguagem fonte.

```
public final static byte IDENTIFIER = 0, INTEGER_LITERAL = 1, OPERATOR_SUM = 2, OPERATOR_SUB = 3, 9 usages
    OPERATOR_MULT = 4, OPERATOR_DIV = 5, OPERATOR_LESSTHAN = 6, OPERATOR_GREATERTHAN = 7, OPERATOR_EQUALS = 8,
    BEGIN = 9, PROGRAM = 10, OPERATOR_AND = 11, OPERATOR_OR = 12, DO = 13, ELSE = 14, END = 15, IF = 16, 6 usages
    INTEGER_TYPE = 17, BOOLEAN_TYPE = 18, THEN = 19, VAR = 20, BOOLEAN_TRUE = 21, BOOLEAN_FALSE = 22, 2 usages
    WHILE = 23, SEMICOLON = 24, COLON = 25, BECOMES = 26, LPAREN = 27, RPAREN = 28, ERROR = 29, EOT = 30; 4 usages
```

Imagem 03: enumeração dos Tokens da linguagem fonte

Logo após o processo de análise sintática é gerada uma Árvore de Sintaxe Abstrata(AST) que representa uma construção válida do programa fonte, ou seja, as derivações das regras apresentadas anteriormente, em forma de nós de uma árvore. Essa árvore pode variar entre os códigos fontes recebidos na entrada do compilador, porém na interface gráfica do compilador é mostrada ao usuário a AST que corresponde ao código fonte requisitado.

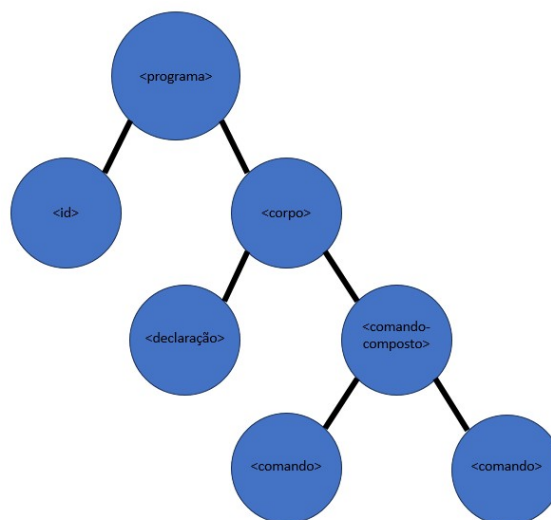


Imagem 04: exemplo de uma Árvore de Sintaxe Abstrata

É implementada também uma tabela de identificação que representa as declarações de variáveis realizadas no código fonte. Essa tabela é utilizada, principalmente para realizar análise de contexto. Essa tabela também é responsável por associar uma chamada de um identificador à sua declaração.

```
public class IdentificationTable {
    List<String> ids;
    List<Type> tipos;

    public IdentificationTable() {...}

    public void enter(String id, String tipo){...}

    public Type retrieve(String id) {...}
}
```

Imagem 05: classe que representa a tabela de identificação em código

7.Montagem e impressão da árvore de sintaxe abstrata

7.1 Descrição do processo de montagem da AST

A árvore de sintaxe abstrata é montada utilizando o método recursivo descendente, ou seja, é montada de cima pra baixo, da raiz até as últimas folhas. Esse método é facilitado pelo motivo de termos uma gramática LL(1), o que simplifica e possibilita uma escolha rápida e fácil da regra de derivação a ser aplicada. A classe que realiza essa montagem é chamada de Parser, ela recebe de entrada os Tokens presentes no código fonte já classificados e categorizados e analisa a ordem em que estão postos e vai criando a AST a partir dessa análise.

7.2 Descrição do processo de impressão da AST

A impressão da árvore de sintaxe abstrata é feita utilizando o padrão de projeto Visitor, que visita todos os nós da AST e imprime seus valores na interface gráfica. É válido citar que o Visitor será utilizado nas etapas de análise de contexto e geração de código também, processos que serão detalhados posteriormente nesse texto. Já na interface gráfica a AST é mostrada utilizando uma estrutura presente na biblioteca gráfica JavaFX, chamada de TreeView. A classe que implementa a interface do Visitor é chamada de Printer, é nela que realmente ocorre a impressão da AST.

O funcionamento do Visitor é simples, um nó chama os próximos e em cada nó é verificado se o mesmo é nulo ou não. A ordem em que são chamados os nós se baseia nas regras de derivação da linguagem, ou seja, por exemplo, o nó <programa> chama os nós <id> e <corpo>, <corpo> por sua vez chama os nós <declaração> e <comando-composto> e o processo é seguido até que não hajam mais nós para serem percorridos.

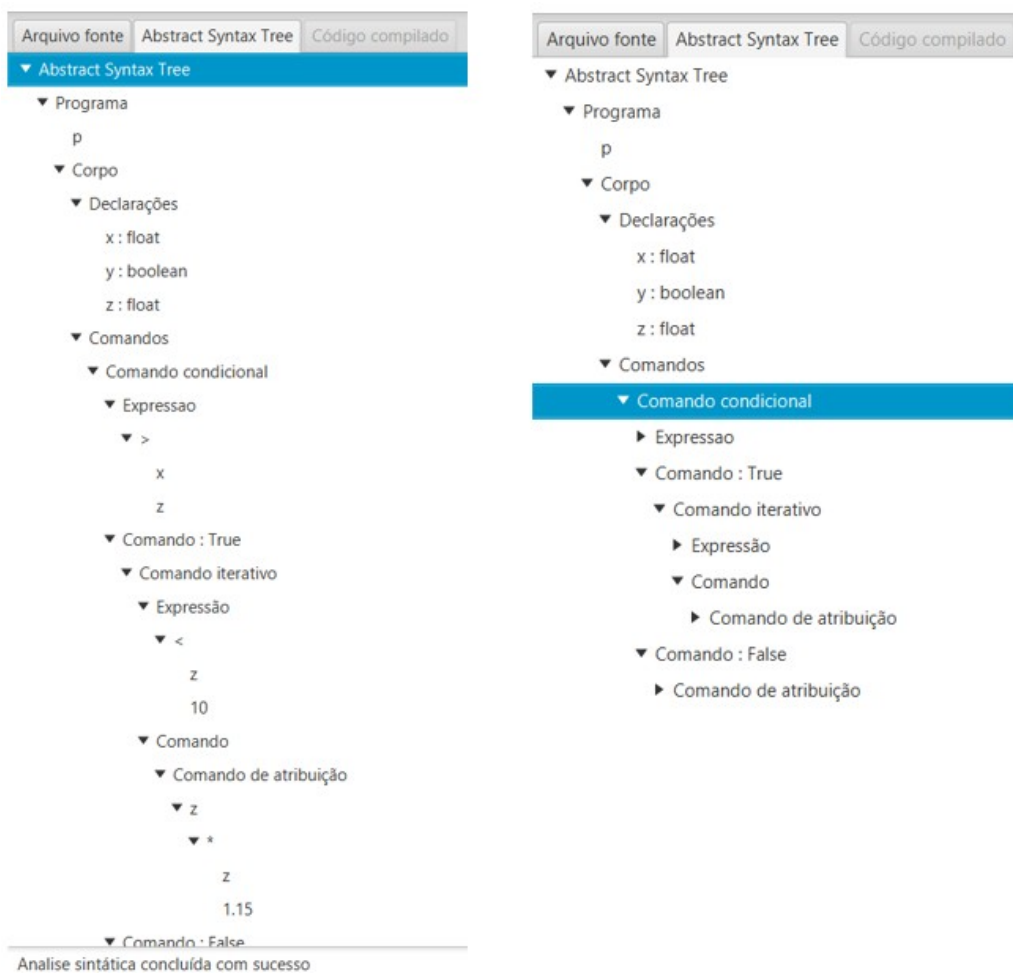
7.3 Exemplos de impressões na interface gráfica

Nas imagens a seguir, estarão presentes duas entradas de código fonte no compilador e como as AST's dessas entradas são mostradas pela interface gráfica.

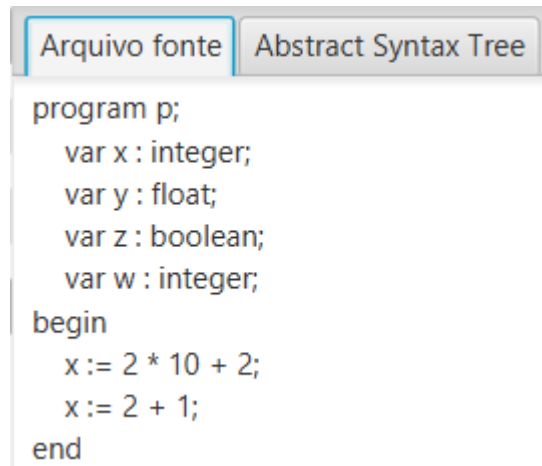
```

program p;
var x : float;
var y : boolean;
var z : float;
begin
  if x > z then
    while z < 10 do
      z := z * 1.15
    end
  else
    y := false;
  end
end
  
```

Imagem 06: entrada de um código fonte no compilador



Imagens 07 e 08: árvore de sintaxe abstrata do código fonte anterior mostrada graficamente no programa, na primeira imagem a AST está totalmente expandida e na segunda apenas alguns nós estão expandidos



```

program p;
  var x : integer;
  var y : float;
  var z : boolean;
  var w : integer;
begin
  x := 2 * 10 + 2;
  x := 2 + 1;
end

```

Imagem 09: um outro código fonte servindo como entrada do compilador

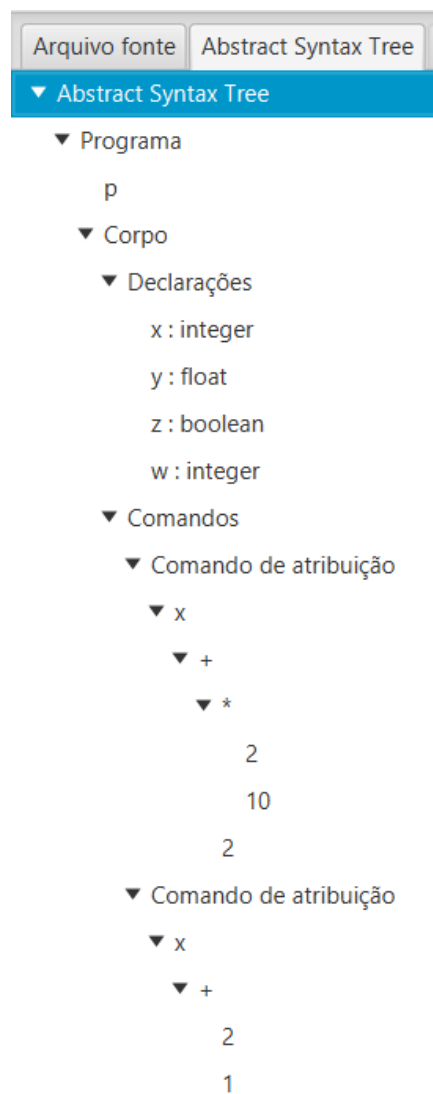


Imagem 10: AST que representa o código fonte anterior impressa na interface gráfica

8. Análise de contexto

Como mencionado na seção 6 desse texto, descrições das principais estruturas de dados utilizadas, a estrutura de dados que será utilizada durante a análise de contexto é tabela de identificação. Participando de ambas as partes da análise de contexto, identificação e verificação. Na tabela de identificação podemos inserir valores (declarações) utilizando o método *enter(String id, String tipo)* e recuperar valores utilizando o método *retrieve(String id)*.

A análise de contexto utiliza do padrão de projeto implementado, Visitor, para visitar todos os nós da árvore de sintaxe abstrata do programa fonte, o que possibilita realizar as etapas de identificação e verificação. A classe que implementa a interface Visitor é chamada de Checker, é ela que inicia o processo de análise de contexto.

8.1 Descrição da fase de identificação

Durante a fase de identificação as declarações de variáveis que estão presentes no código fonte são postas na tabela utilizando o método *enter(...)*. Antes de serem inseridas devidamente na tabela é checado se aquela declaração já não existia na tabela, pois é uma regra de contexto da linguagem fonte que cada identificador de variável seja declarado apenas uma vez. Sendo válida a inserção da declaração, é inserido então o identificador da variável e também seu tipo, *integer* ou *boolean*.

8.2 Descrição da fase de verificação

A fase de verificação ocorre sempre que uma variável é chamada em algum comando ao longo do corpo do código fonte. Essa fase checa se o tipo da expressão presente em algum comando é válido para aquele comando específico. Ou seja, em exemplo, se estamos atribuindo uma expressão a uma variável em um comando de atribuição, aquela expressão deve resultar em um valor do tipo daquela variável. Citando outro exemplo, observamos um comando condicional, a expressão nele presente deve ser do tipo booleano, logo a fase de verificação checará se a expressão resulta em um tipo booleano válido. Nessa fase, é utilizado o método *retrieve(...)* que recupera o tipo de uma variável declarada com base no seu identificador.

8.3 Exemplos de entradas e saídas na análise de contexto

Nas imagens a seguir estarão presentes exemplos de entradas e saídas do analisador de contexto.

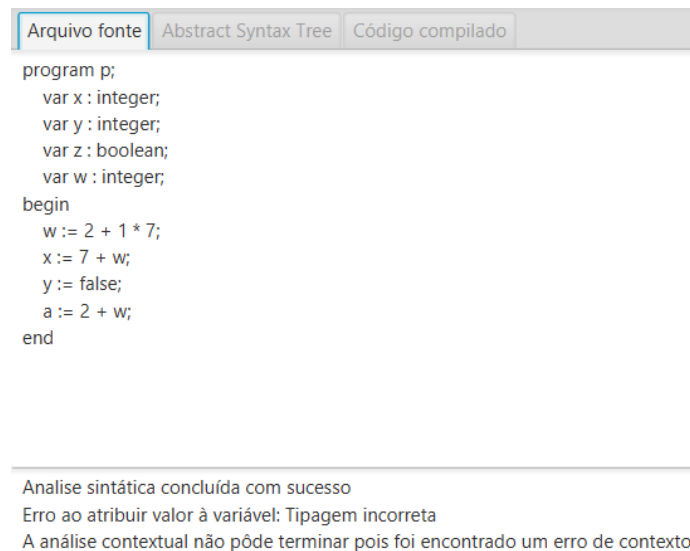


Imagem 11: Exemplo de entrada no compilador onde há a presença de um erro de contexto

Na imagem 11 observamos que o analisador de contexto observou a atribuição de uma expressão cujo tipo de retorno não corresponde ao tipo de declaração da variável: A variável *y* foi declarada para armazenar um valor inteiro, porém vemos que é realizada a atribuição de um valor booleano a ela.

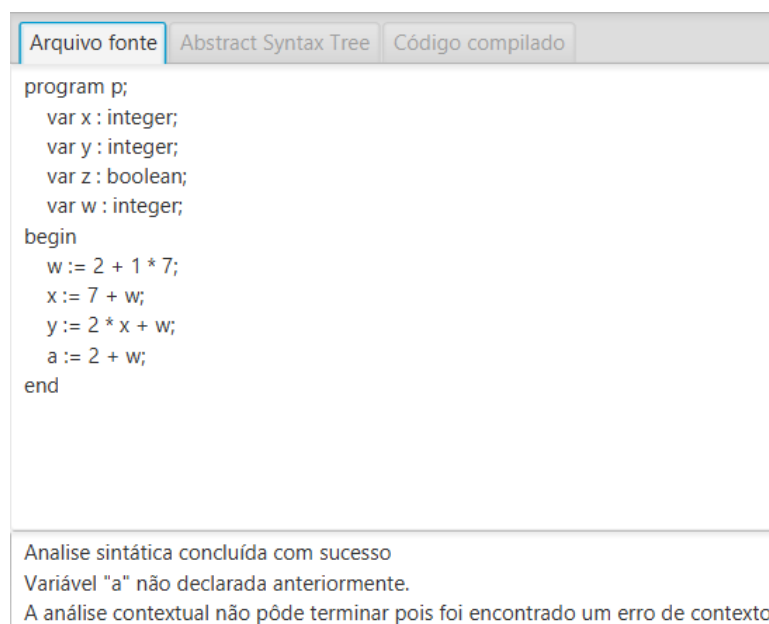


Imagem 12: Exemplo de entrada de um outro código fonte onde há a presença de outro erro de contexto

Já na imagem 12 observamos que o erro de contexto presente se refere à chamada de uma variável que não foi declarada previamente.

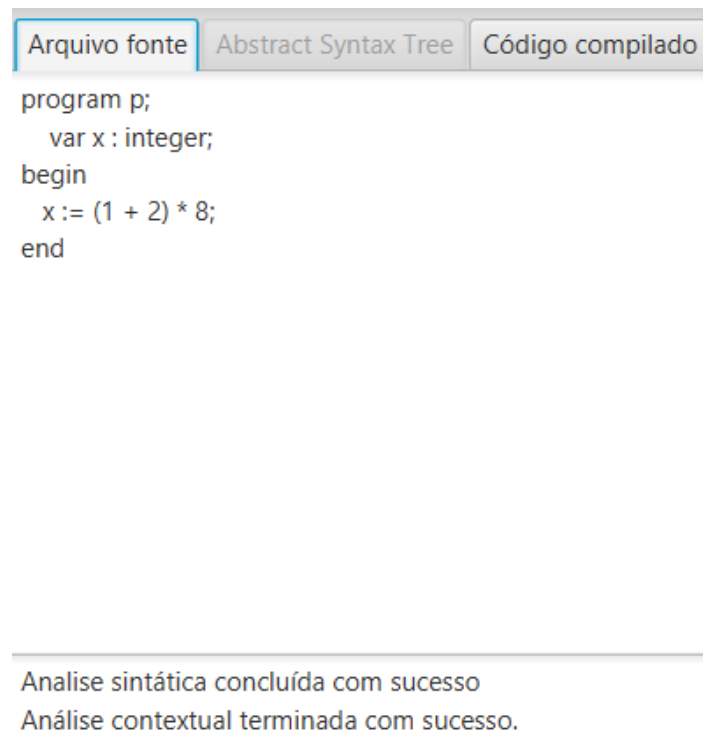


Imagem 13: Exemplo de entrada de um outro código fonte onde não há erros de contexto presentes

Na imagem 13 observamos que não há erro de contexto presente no código fonte do programa, resultando no término com sucesso da análise de contexto.

9. Geração de código

9.1 Funções de código

As funções de códigos utilizadas para auxiliar a tradução do programa fonte foram as seguintes:

Para o nó programa da AST:

```

run[P] =
  execute P
  HALT

```

Para os nós de comando sequencial, comando de atribuição, comando condicional incompleto e completo e comando iterativo, temos, respectivamente:

```

execute[C1; C2] =
  execute C1
  execute C2

```

```

execute[I := E] =
  evaluate E
  assign I

```



```

execute[if E then C] =
    evaluate E
    JUMPIF(0) g
    execute C
g:

```

```

execute[if E then C1 else C2] =
    evaluate E
    JUMPIF(0) g
    execute C1
    JUMP h
g: execute C2
h:

```

```

execute[while E do C] =
    g: evaluate E
        JUMPIF(0) h
    execute C
h:

```

Para uma expressão binária, temos a seguinte função de código, sendo p uma operação primitiva, como soma(add), multiplicação(mult), entre outras.

```

execute[E1 O E2] =
    evaluate E1
    evaluate E2
    CALL p

```

Para recuperar e guardar o valor de uma variável, temos, respectivamente:

```

fetch[I] =
    LOAD d[SB]

```

```

assign[I] =
    STORE d[SB]

```

Onde $d[SB]$ é o endereço relativo ao stack base da alocação estática realizada pelo programa.

Por fim, para as declarações de variáveis e declarações sequenciais temos, respectivamente:

```

elaborate[var I : T] =
    PUSH size

```

```

elaborate[D1, D2] =
    elaborate D1
    elaborate D2

```

Onde size é o tamanho do tipo daquela variável. Em nossa linguagem fonte podemos declarar uma variável para guardar um tipo inteiro e um tipo booleano, ambos com size de uma palavra.

9.2 Descrição das estruturas de dados e algoritmos utilizados

Durante a geração de código a única estrutura de dados que participa do processo é a tabela de identificação, pois é nela que é guardada as declarações de variáveis e, conseqüentemente o endereço das mesmas. Quando ocorre a chamada de uma variável em alguma expressão, por exemplo, chamamos a instrução LOAD d[SB] que carrega a variável colocada no endereço d em relação ao stack base no topo da pilha.

Essa estrutura de dados já foi explicada anteriormente nesse texto porém, de forma resumida, ela representa uma tabela das declarações das variáveis, contendo seu identificador, tipo e endereço. É possível realizar duas operações sobre essa tabela, inserir uma declaração e recuperar uma declaração.

9.3 Exemplos de traduções com trechos representativos do programa-fonte

Alguns exemplos de entradas de programas fontes válidos e suas respectivas saídas, traduzidas, seguem a seguir:

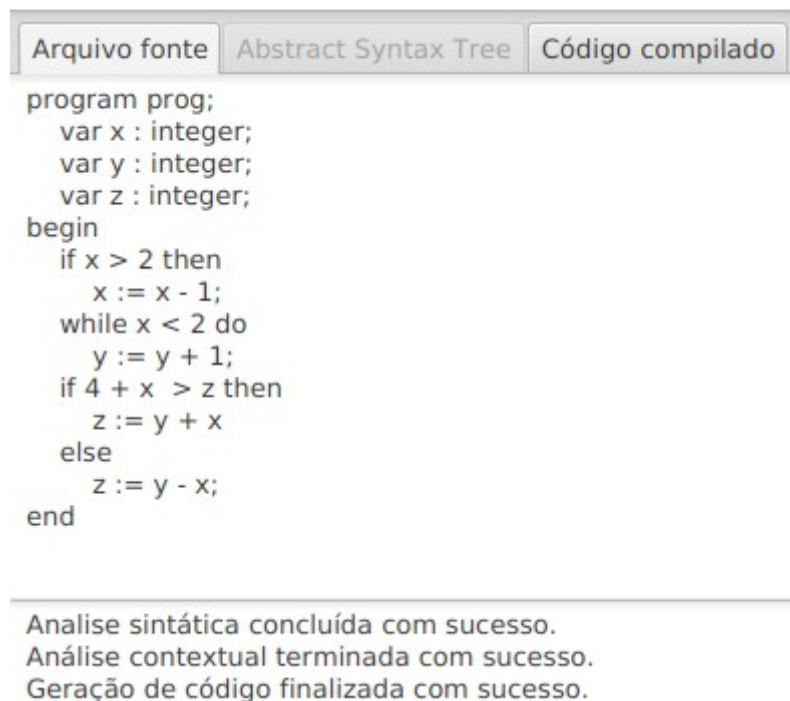


Imagem 14: Um programa fonte válido recebido como entrada do compilador

Arquivo fonte	Abstract Syntax Tree	Código compilado
<pre> L0001: PUSH 1[SB] L0002: PUSH 1[SB] L0003: PUSH 1[SB] L0004: LOAD 0[SB] L0005: LOADL 2 L0006: CALL gt L0008: JUMPIF(0) L0007 L0009: LOAD 0[SB] L0010: LOADL 1 L0011: CALL sub L0012: STORE 0[SB] L0007: L0013: L0015: LOAD 0[SB] L0016: LOADL 2 L0017: CALL lt L0018: JUMPIF(0) L0014 L0019: LOAD 1[SB] L0020: LOADL 1 L0021: CALL add L0022: STORE 1[SB] L0023: JUMP L0013 L0014: L0024: LOADL 4 L0025: LOAD 0[SB] L0026: CALL add L0027: LOAD 2[SB] L0028: CALL gt L0031: JUMPIF(0) L0029 L0032: LOAD 1[SB] L0033: LOAD 0[SB] L0034: CALL add L0035: STORE 2[SB] L0036: JUMP L0030 L0029: L0037: LOAD 1[SB] L0038: LOAD 0[SB] L0039: CALL sub L0040: STORE 2[SB] L0030: L0041: HALT </pre>		

Imagem 15: Programa objeto que representa a saída do programa fonte da Imagem 14

Arquivo fonte	Abstract Syntax Tree	Código compilado
<pre> program p; var x : integer; var y : integer; var z : boolean; var w : integer; begin x := y + 2; end </pre>		

Análise sintática concluída com sucesso.
 Análise contextual terminada com sucesso.
 Geração de código finalizada com sucesso.

Imagem 16: Um outro exemplo de programa fonte válido sendo recebido na entrada do compilador

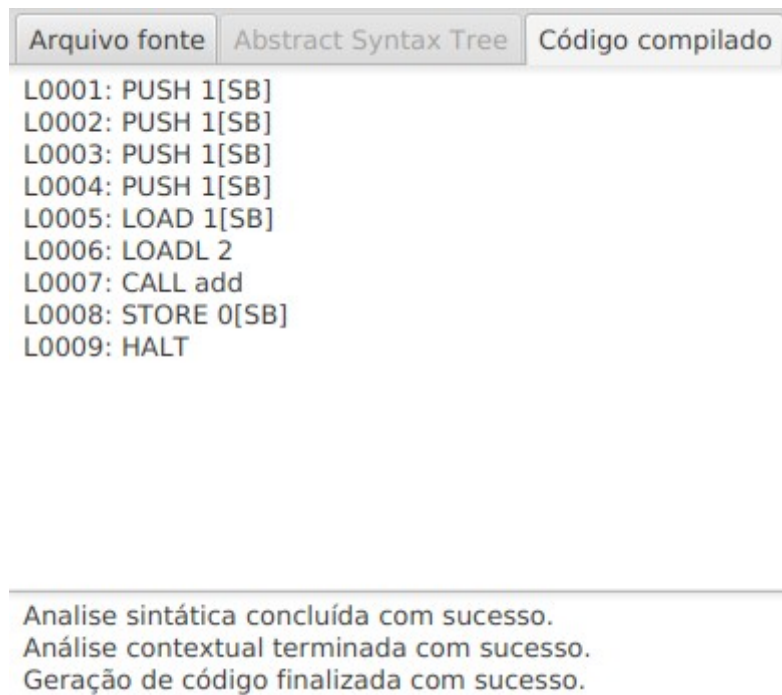


Imagem 17: Saída do programa fonte recebido na imagem 16

10. Linguagem objeto

10.1 Relação de instruções

A linguagem objeto utilizada no processo de tradução do programa fonte possui as seguintes instruções:

HALT, PUSH, LOAD, LOADL, CALL, STORE, JUMPIF, JUMP.

10.2 Sintaxe

A sintaxe da linguagem objeto é representada pelas funções e pelos modelos de código mencionados no tópico 9.1 anteriormente nesse texto. Seguindo também os nós da AST criada após a etapa de análise sintática. Ou seja, a tradução do programa fonte é feita seguindo a ordem dos nós: programa > declarações > comandos.

10.3 Semântica

A semântica das instruções listadas no tópico 10.1 são as seguintes, para cada instrução:

HALT: Encerra a execução do programa.

PUSH d[SB]: Aloca uma variável de tamanho d no stack base.

LOAD d[SB]: Carrega no topo da pilha de execução o valor da variável presente no endereço d em relação ao stack base.

LOADL l: Carrega um literal inteiro no topo da pilha de execução.

CALL p: Chama uma operação primitiva p, que irá consumir valores da pilha de execução.

STORE d[SB]: Consome um valor presente no topo da pilha de execução e o guarda no endereço d em relação ao stack base.

JUMPIF(v) g: Representa um pulo condicional, ou seja, se o valor presente no topo da pilha de execução for o valor v, pular para a instrução presente no rótulo g do código objeto.

JUMP g: Representa um pulo incondicional, ou seja, pule para a instrução presente no rótulo g independente de qualquer condição.

11. Ambiente de execução

11.1 Tipo de máquina usada

A máquina virtual JVM utilizada para compilar e executar o programa em sua versão final foi a versão 17.

11.2 Avaliação de expressões

A avaliação de expressões presentes no programa fonte foi deixada para ser realizada na execução do programa objeto. Ou seja, todas as expressões presentes no programa fonte são traduzidas diretamente para o programa objeto.

11.3 Formas de alocação de memória empregadas

Seguindo a linguagem fonte que é válida como entrada do compilador apresentado no texto, o único tipo de alocação de memória realizada é a alocação estática, para variáveis disponíveis globalmente no programa, ou seja, durante toda sua execução. Esse tipo de alocação são feitas no Stack Base durante a compilação do programa fonte.

12. Manual de compilação

O código fonte do compilador foi fornecido pelo aluno ao professor através da entrega de um pendrive.

12.1 Compilação e execução em sistemas Linux

Para compilar o código fonte em uma versão executável é necessário ter instalado na máquina a ferramenta Maven, como também a sua devida configuração para utilizar a versão 17 do Java.

Para obter o executável do compilador basta executar o seguinte comando no diretório do código fonte:

```
mvn clean package
```

Após a execução com sucesso do comando acima, será criada no diretório a pasta *target* que conterá o arquivo *.jar* correspondendo ao compilador, pronto para ser executado. O arquivo terá o nome de *Compilador-1.0.jar*.

Para executar o arquivo *.jar* basta executar a linha de comando no diretório *.../target/*:

```
java -jar Compilador-1.0.jar
```

Observando a necessidade de executar o comando acima com a versão 17 ou superior do Java. Para consultar a versão do Java instalada e selecionada na máquina basta executar o comando:

```
java -version
```

O código fonte foi desenvolvido utilizando a IDE IntelliJ IDEA Community, logo, é recomendada a utilização da mesma para análise e modificação do código fonte.

12.2 Compilação e execução no Windows

No Windows, para obter o executável do compilador basta rodar o seguinte programa no diretório do código fonte:

```
./mvnw.cmd clean package
```

Observando que é necessário ter configurada no sistema a variável *JAVA_HOME* com uma JDK na versão 17 ou maior do Java.

Para a execução do programa fonte compilado basta executar o comando mencionado anteriormente, no diretório *.../target/*:

```
java -jar Compilador-1.0.jar
```

12.3 Observações ao professor

Os arquivos executáveis estão disponibilizados no pendrive fornecido ao professor. No diretório chamado *executáveis*. Nesse mesmo diretório estão presentes alguns arquivos de teste, na pasta *arquivos de teste*.

Os arquivos das classes desenvolvidas para o compilador estão presentes no seguinte diretório e suas subpastas, a partir do diretório base do pendrive de entrega:

```
/Compilador/src/main/java/main/compilador
```

13. Manual de utilização

Para executar devidamente o executável do compilador é preciso ter no sistema no mínimo a versão 17 da JDK do Java. É também de muita importância observar que, a versão compilada, seguindo os passos do tópico anterior, é apenas executável no mesmo

sistema em que foi compilada, ou seja, se o código fonte foi compilado em um sistema linux ele é executável apenas em outros sistemas linux.

É utilizada, para auxiliar na interface gráfica do programa, a biblioteca JavaFX, logo, para realizar mudanças gráficas no programa se faz necessário a obtenção da versão 21 da SDK do JavaFX.

13.1 Como utilizar o programa

Após executar o programa seguindo as instruções do tópico 12, a tela inicial deve ser a mesma ou semelhante à mostrada a seguir:

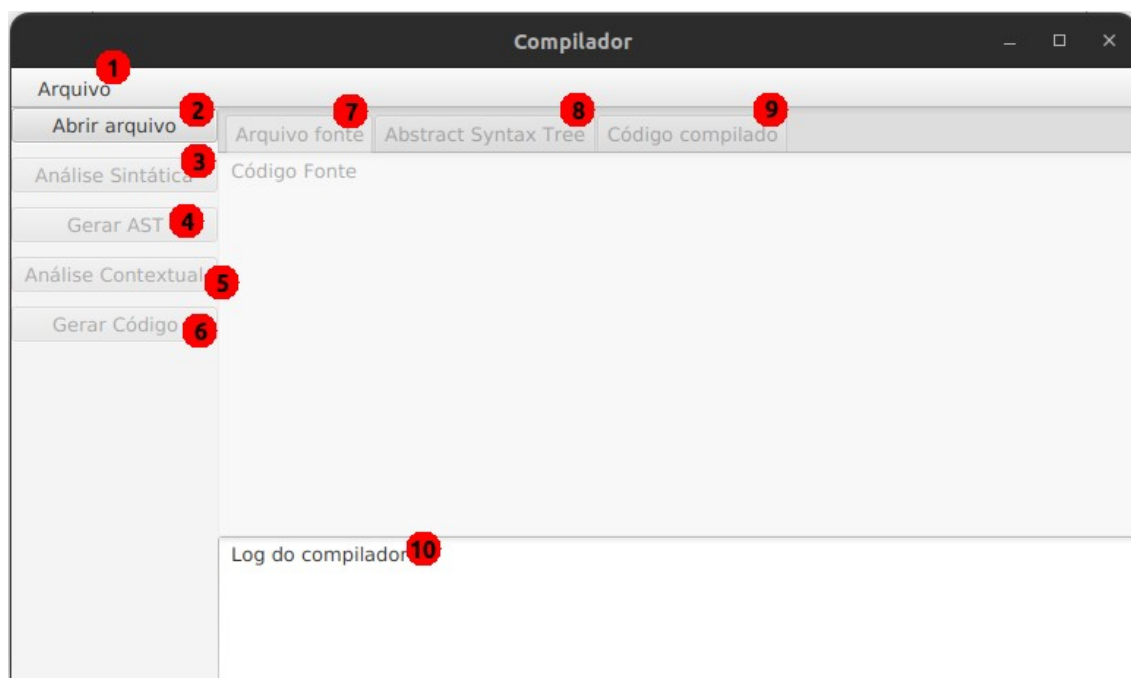


Imagem 18: Tela inicial do compilador com numerações

A imagem 18 possui numerações de algumas áreas que são ou botões com ações ou áreas de informação, que serão explicadas a seguir.

01: O botão “Arquivo” abre um menu onde é possível abrir um novo programa fonte, ao selecionar a opção “Abrir” ou salvar o programa objeto obtido na saída, selecionando a opção “Salvar”.

02: O botão “Abrir arquivo” abre um novo arquivo. Ao abrir um novo arquivo, o programa fonte é exibido na guia “Arquivo fonte” representada na figura pela numeração 7.

03: O botão “Análise Sintática” realiza a análise sintática do programa fonte, se o mesmo foi carregado corretamente. Ficará desabilitado até que um arquivo fonte seja aberto, e após acionado também ficará desativado. O resultado da análise sintática será mostrado no “Log do compilador” representado pela numeração 10 na figura. Ou seja, quaisquer erros ou uma mensagem de sucesso será mostrada nessa área.

04: O botão “Gerar AST” imprime a árvore de sintaxe abstrata que representa o programa fonte. A AST ficará disponível na janela “Abstract Syntax Tree” representada pela numeração 8 na imagem 18. Esse botão só ficará disponível para ser ativado após ser realizada a análise sintática.

05: O botão “Análise Contextual” realiza o processo de análise de contexto sobre o programa fonte. Quaisquer erros encontrados durante a análise de contexto serão mostrados na área 10 da figura. Esse botão só ficará disponível para ser ativado após ser realizada a análise sintática.

06: O botão “Gerar código” realiza a tradução da linguagem fonte para a linguagem objeto. Esse botão só fica disponível para ser clicado após término com sucesso da análise contextual. O código traduzido ficará disponível na janela “Código compilado” representada pelo número 9 na figura.

07: Janela onde é mostrado o programa fonte após ser devidamente aberto utilizando algum dos botões 1 ou 2.

08: Janela onde é mostrada a árvore de sintaxe abstrata após o botão 4 ser clicado.

09: Janela onde é mostrado o código fonte traduzido para a linguagem objeto. Ao clicar no menu 1 e na opção “Salvar” o arquivo salvo terá o código mostrado nessa janela.

10: Log do compilador. Aqui são mostradas as mensagens de erros e também de sucesso após realizar as operações listadas a cima.

13.2 Extensões dos arquivos de entrada e saída do compilador

Os programa fonte abertos para serem usados como entrada do compilador devem estar na extensão `.txt` e podem estar em qualquer diretório na máquina, visto que, ao clicar em algum dos botões de abrir arquivo um prompt de abrir arquivo é aberto, basta navegar para o diretório do arquivo fonte. O código objeto é salvo também com a extensão `.txt` e pode ser salvo também em qualquer diretório da máquina.

13.3 Mensagens de erro do compilador

As mensagens de erro acerca dos processos de análise sintática e de contexto e geração de código são exibidas no log do compilador, como mencionado anteriormente no tópico 13.1.

Para a análise sintática as mensagens de erro dizem a respeito da utilização das palavras chaves e sua ordem. A mensagem de erro mostra o que há de errado no código fonte e mostra possíveis correções válidas para o erro.

Já as mensagens de erro resultantes do processo de análise de contexto informam apenas o tipo de erro que está presente no código fonte, detalhando o erro a um nível em que seja possível o usuário entender e conseguir corrigir o erro. Sendo válido mencionar que apenas o primeiro erro de contexto é informado.

Por fim, para o processo de geração de código, visto que todas as checagens já foram realizadas anteriormente e essa última etapa do processo de compilação é apenas uma tradução direta do código fonte, não há erros para serem mostrados no log do compilador.

14. Conclusões finais

Em conclusão final o aluno considerou o projeto desenvolvido ao longo do semestre como fundamental para a fixação do conteúdo passado em sala pelo professor. Considerando a própria implementação do compilador e também sua documentação como ferramentas de estudo para as provas da matéria e também para o aprendizado do conteúdo.

Acredita-se que a maior dificuldade do aluno durante a elaboração desse trabalho estando na disponibilização de tempo para o projeto, visto que, o projeto em si não possui um nível muito complexo de implementação em código, porém, demandou bastante tempo do aluno.