

UNIVERSIDADE ESTADUAL PAULISTA
FACULDADE DE CIÊNCIAS E TECNOLOGIA

PROJETO COMPILADOR – ANALISADOR SINTÁTICO PARA LALG
RELATÓRIO – PARTE 3

COMPILADORES
PROF. DR. CELSO OLIVETE JÚNIOR

BRUNO SANTOS DE LIMA
LEANDRO UNGARI CAYRES

PRESIDENTE PRUDENTE
JANEIRO - 2018

BRUNO SANTOS DE LIMA
LEANDRO UNGARI CAYRES

PROJETO COMPILADOR – ANALISADOR SEMÂNTICO, GERAÇÃO DE
CÓDIGO E INTERPRETAÇÃO PARA LALG
RELATÓRIO
PARTE 3

Relatório do projeto prático - parte III da disciplina de
Compiladores, lecionada pelo docente Dr. Celso
Olivete Júnior, no curso Bacharelado em Ciência da
Computação – Departamento de Matemática e
Computação da Faculdade de Ciências e Tecnologia
(FCT Unesp – Presidente Prudente).

PRESIDENTE PRUDENTE

JANEIRO – 2018

SUMÁRIO

1 INTRODUÇÃO	4
2 FUNCIONAMENTO DO ANALISADOR SINTÁTICO CONSTRUÍDO	5
2.1 Descrição teórica do programa	5
2.1.1 JavaCC	5
2.2 Descrição da estrutura e organização do programa	6
2.4 Processo de recuperação de erros	7
2.5 Testes realizados na ferramenta	7
2.5.1 Teste 1: Programa correto1	7
2.5.4 Teste 4: Programa incorreto	8
3 FUNCIONAMENTO DO ANALISADOR SEMÂNTICO	9
3.1 Detalhes inerentes a implementação da análise semântica	10
3.2 Casos de teste	11
3.2.1 Teste 1: Programa sem erro semântico	11
3.2.1 Teste 2: Programa com erro semântico	12
4 GERAÇÃO E INTERPRETAÇÃO DE CÓDIGO	14
4.1 Linguagem <i>BYTECODE</i>	14
4.2 Utilização na aplicação	16

1 INTRODUÇÃO

Este trabalho da disciplina de Compiladores tem como objetivo a construção de uma aplicação, que realize a análise sintática, semântica e a geração de código para programas que são escritos na linguagem de programação LALG.

O analisador sintático é o processo que analisa uma dada sequência de entrada e determina sua estrutura gramatical seguindo uma determinada gramática formal. Basicamente verifica se a sequência de entrada pertence a gramática de uma determinada linguagem. Em caso negativo o analisador sintático informa um erro ou um conjunto de erros.

O analisador semântico visa verificar o uso adequado acerca dos seguintes pontos: análise contextual, checagem de tipos, unicidade. A análise semântica pode identificar os seguintes erros: variável ou procedimento não declarado ou declarado mais de uma vez, incompatibilidade de parâmetros, uso de variáveis em escopo inadequado, variável declarada, mas nunca utilizada, entre outros.

Este trabalho está organizado do seguinte modo: na Seção 2 é relatado o funcionamento do analisador sintático construído contendo a descrição teórica do programa e sua descrição de estrutura e funcionamento. Além disso é apresentado testes de programas e os resultados da análise, na Seção 3 é apresentado o funcionamento da análise semântica e exemplos de testes realizados na mesma, por fim a Seção 4 descreve o processo de geração de código intermediário e interpretação de código gerando a execução do programa escrito na linguagem LALG.

2 FUNCIONAMENTO DO ANALISADOR SINTÁTICO CONSTRUÍDO

2.1 Descrição teórica do programa

A aplicação foi desenvolvida na linguagem de programação orientada a objetos Java utilizando a IDE NetBeans 8.2. Para a interface gráfica foi utilizado o pacote Swing do Java, além disso foi utilizado uma Biblioteca de Gerador denominada JavaCC. A aplicação encontra-se no repositório: <https://github.com/brunoslima/ProjetoCompilador>. Na Figura 1 é ilustrado a interface da ferramenta em seu estado atual.

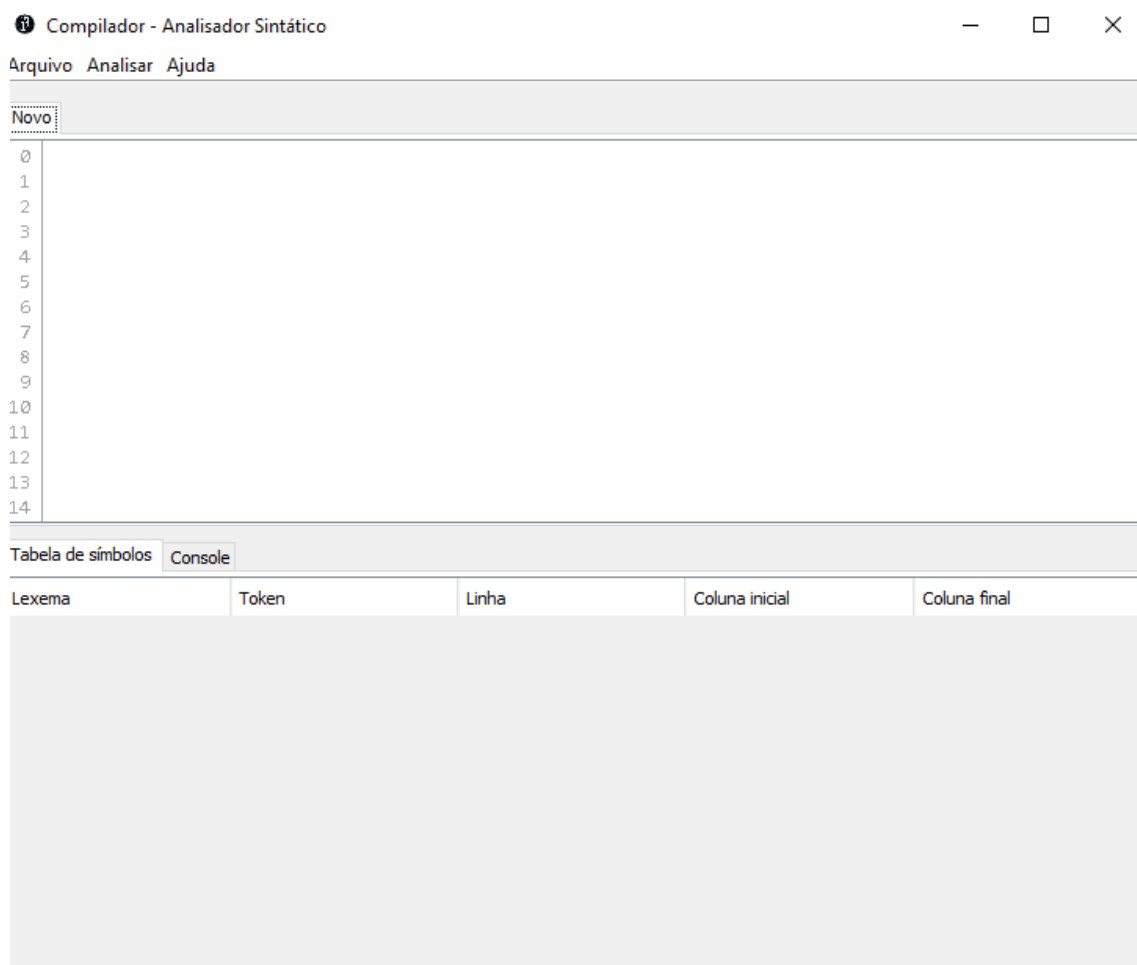


Figura 1 - Instantâneo da interface atual da ferramenta

2.1.1 JavaCC

O JavaCC é um conjunto de Bibliotecas que podem ser adicionadas a um projeto com o objetivo de facilitar a análise sintática. O JavaCC é um gerador de analisador

sintático aberto da linguagem Java. O JavaCC permite a elaboração de um analisador sintático através de uma gramática fornecida em E-BNF. O analisador sintático gerado por ele é descendente, o qual utiliza classes gramaticais LL(1), o que exclui recursividade a esquerda, sendo necessário transformar a gramática original da LALG em LL(1). Para maiores informações consulte: <https://javacc.org/>

2.2 Descrição da estrutura e organização do programa

Neste trabalho, o JavaCC foi utilizado da seguinte forma: primeiramente foram importadas as bibliotecas do JavaCC para o projeto, em seguida foi construído um arquivo chamado Grammar.jj que contém métodos para realizar a análise. Esse arquivo quando compilado gera automaticamente um conjunto de classes, com seus respectivos métodos implementados, que guiados pelas descrições do Grammar.jj realizam a análise sintática sobre um conjunto de entrada.

A aplicação contém um conjunto de classes organizadas em pacotes. A classe Arquivo.java é responsável por abrir e ler um arquivo texto que contém um código fonte que sofrerá uma análise sintática.

Com o código-fonte lido através de um arquivo ou simplesmente digitado na área de texto pode-se executar a análise sintática e como resultado é informado se a análise foi concluída com sucesso ou se contém erros, caso contenha erros são apresentados uma listagem dos mesmos.

As demais classes e pacotes são responsáveis pela construção da interface gráfica da aplicação.

2.3 Descrição do funcionamento do programa

O funcionamento do programa ocorre do seguinte modo: dada uma sequência de entrada a mesma é analisada seguindo as diretrizes do arquivo Grammar.jj. Este arquivo contém a gramática da linguagem LALG, bem como seus terminais e não terminais. Logo no início do arquivo contém alguns métodos responsáveis por realizar a comparação de tokens, recuperar e salvar os possíveis erros gerados.

O arquivo é formado por um conjunto de procedimentos recursivos, cada um desses procedimentos está relacionado com um não terminal, assim a cada token lido da sequência de entrada é verificado se ele se enquadra na gramática descrita, caso se

enquadre este realiza chamadas nos procedimentos da regra na qual ele se enquadra, caso não se enquadre em nenhuma regra ou mesmo quebre alguma regra um método é chamado para armazenar o erro encontrado e assim através do modo pânico a análise sintática continua a acontecer.

2.4 Processo de recuperação de erros

Nem sempre a análise sintática vai se deparar com um programa que não tenha nenhum erro gramatical, sendo assim foi implementado um mecanismo de recuperação de erros para guardar todos os erros encontrados pela análise.

Existe uma classe chamada `RecuperacaoErros` que é responsável por manter a lista de todos os erros encontrados durante a análise sintática.

Como dito anteriormente a recuperação de erros ocorre utilizando o modo pânico, assim quando um token é lido e identificado que quebra a regra gramatical os demais tokens vindos na sequência são descartados até que uma nova regra possa ser formada, assim continuar a análise sintática.

Ao ler um token é observado a regra atual em que seguindo a gramática e também é espiado os tokens seguintes, utilizando a função *LOOKAHEAD* no arquivo `Grammar.jj`, caso exista um erro é visto com base na gramática o token que era esperado e é chamada uma função que armazena os erros na lista de erros da classe `RecuperacaoErros` e os tokens seguintes são descartados utilizando a ideia do modo pânico.

2.5 Testes realizados na ferramenta

A seguir são descritos alguns testes aplicados sobre a ferramenta. Os testes têm como objetivo mostrar o bom funcionamento da ferramenta e da sua capacidade em realizar a análise sintática com tratamento de erros.

2.5.1 Teste 1: Programa correto1

O primeiro teste foi aplicado testando um programa previamente conhecido e correto no qual atribui a divisão do número 10 por 2. Note o resultado da saída como análise sintática concluída e sem erros.

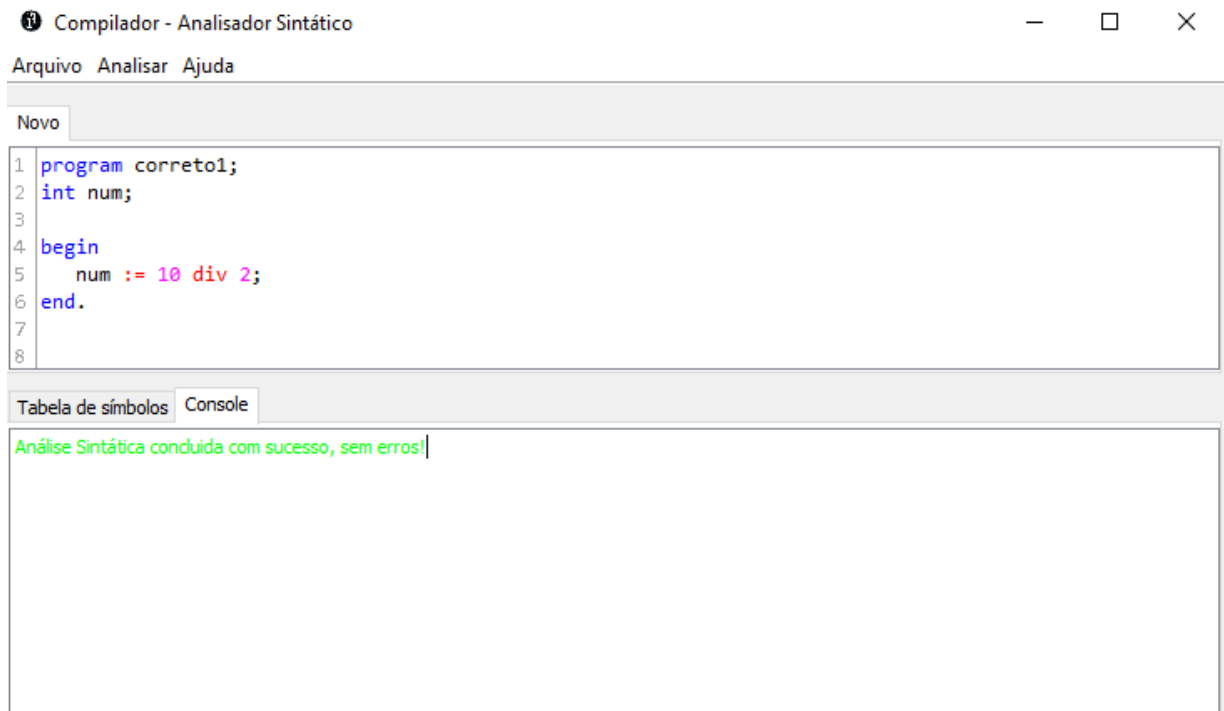


Figura 2 - Teste 1: Programa correto1

2.5.4 Teste 4: Programa incorreto

O último teste que esse relatório apresenta consiste em um programa incorreto mais complexo contendo uma maior quantidade de erros. A seguir é descrito o programa incorreto2.

```
program correto;
int a, b, c;
boolean d, e, f;

procedure proc(var a1 : int);
int a, b, c;
boolean d, e, f;
begin
    a:=1;
    if (a<1)
        a:=12
end;

begin
    a:=2;
    b:=10;
    c:=11;
    a:=b+c;
    d:=true;
    e:=false;
    f:=true;
    read(a);
    write(b);
    if (d)
```



```

begin
    a:=20;
    b:=10*;
    c:=a div b
end;
while (a>1)
begin
    if (b>10)
        b:=2;
    a:=a-1
end
end.

```

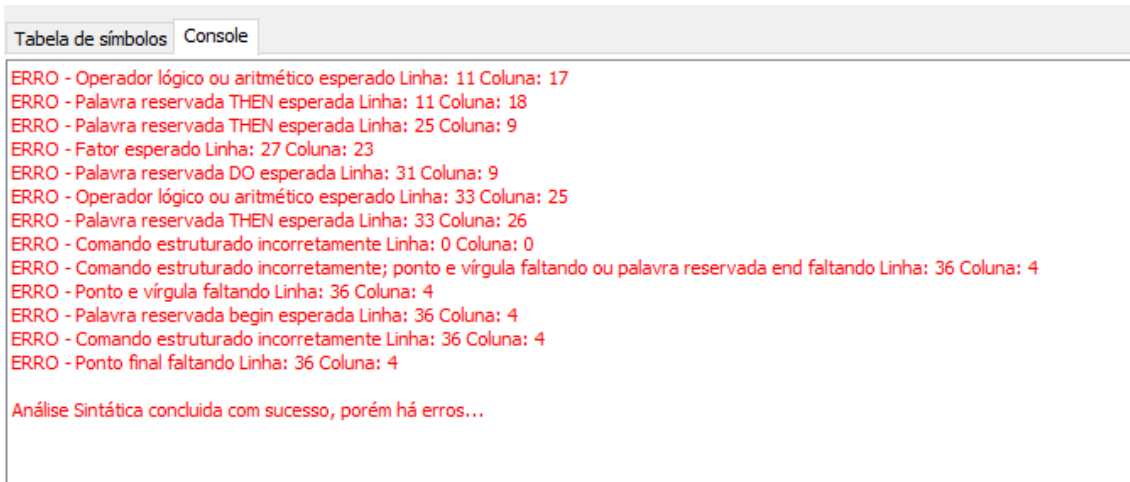


Figura 3 – Teste 5 - Erros encontrados no programa incorreto2

3 FUNCIONAMENTO DO ANALISADOR SEMÂNTICO

O analisador semântico é executado somente após a análise sintática ter sido concluída. Nesta fase é verificado se existem erros semânticos, em caso negativo então a geração de código intermediário pode ter seu início. Em caso positivo, os possíveis erros que podem ocorrer são: variável ou procedimento não declarado ou declarado mais de uma vez, incompatibilidade de parâmetros, uso de variáveis em escopo inadequado, variável declarada, mas nunca utilizada, entre outros.

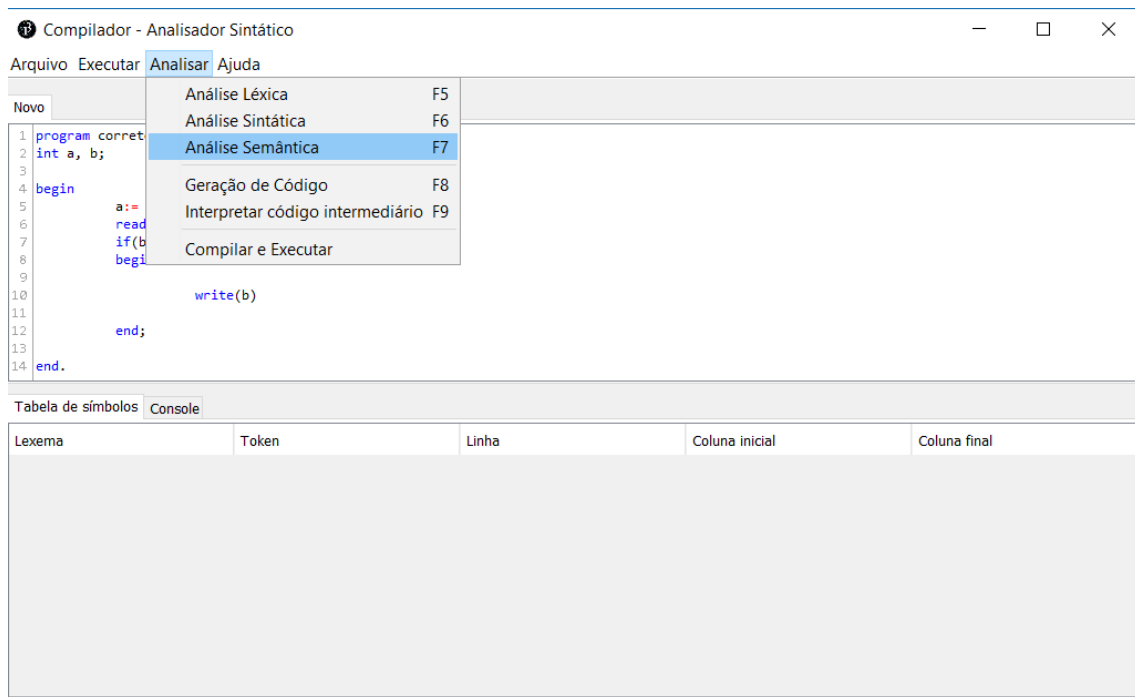


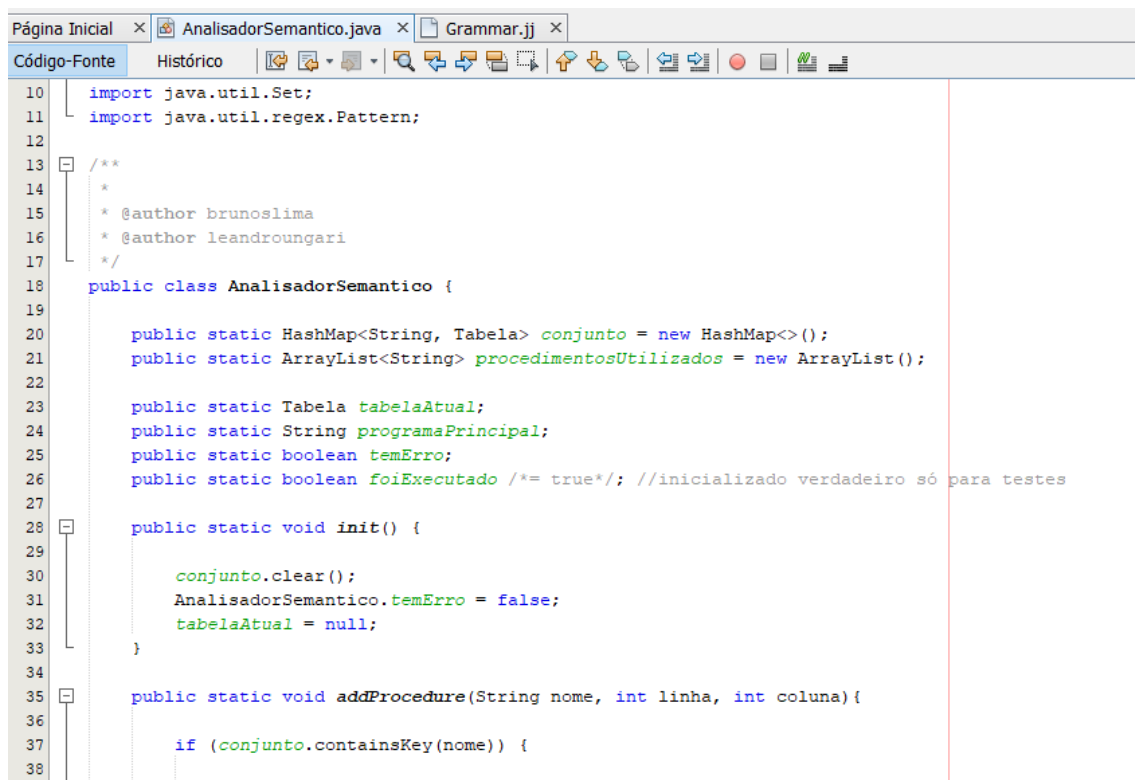
Figura 4 - Instantâneo da interface da aplicação ilustrando o início da análise semântica

Na aplicação para iniciar a análise semântica primeiramente deve-se ter um código escrito utilizando a linguagem LALG, após isso basta clicar em **Analisar -> Análise Semântica** ou simplesmente utilizar o atalho **F7**. Após isso o resultado da análise semântica é apresentado no Console.

3.1 Detalhes inerentes a implementação da análise semântica

O processo de implementação da análise semântica de modo análogo a análise sintática também utiliza o gerador implementado com a utilização do Javacc, mais especificamente utilizando o arquivo Grammar.jj.

Foi implementada um conjunto de classes que trabalham em conjunto para a realização da análise semântica. Dentre elas a principal é a `AnalisadorSemantico.java` ela é a única que trabalha em conjunto com o `Grammar.jj`, sendo ainda a classe que gerencia todas as demais.



```
10 import java.util.Set;
11 import java.util.regex.Pattern;
12
13 /**
14  *
15  * @author brunoslima
16  * @author leandroungari
17  */
18 public class AnalisadorSemantico {
19
20     public static HashMap<String, Tabela> conjunto = new HashMap<>();
21     public static ArrayList<String> procedimentosUtilizados = new ArrayList();
22
23     public static Tabela tabelaAtual;
24     public static String programaPrincipal;
25     public static boolean temErro;
26     public static boolean foiExecutado /*= true*/; //inicializado verdadeiro só para testes
27
28     public static void init() {
29
30         conjunto.clear();
31         AnalisadorSemantico.temErro = false;
32         tabelaAtual = null;
33     }
34
35     public static void addProcedure(String nome, int linha, int coluna){
36
37         if (conjunto.containsKey(nome)) {
38
```

Figura 5 - Parte da classe AnalisadorSemantico.java

Além disso, existe segunda classe importante denominada TabelaErrosSemantico.java, responsável por abrigar em uma lista todos os erros encontrados pelo analisador semântico.

3.2 Casos de teste

A seguir, são descritos dois casos de testes que ilustram o funcionamento do analisador semântico. No primeiro teste temos o exemplo de um programa que não apresenta nenhum erro semântico, o segundo caso de teste ilustra o exemplo de um programa com uma sequência de erros semânticos.

3.2.1 Teste 1: Programa sem erro semântico

Neste primeiro caso de teste, tem-se o programa semantico1.txt que não apresenta nenhum erro semântico, consequentemente também não apresenta erro sintático. Após a

execução da análise semântica nenhum erro foi encontrado e uma mensagem informa que a análise semântica foi concluída com sucesso, sem apresentar erros.

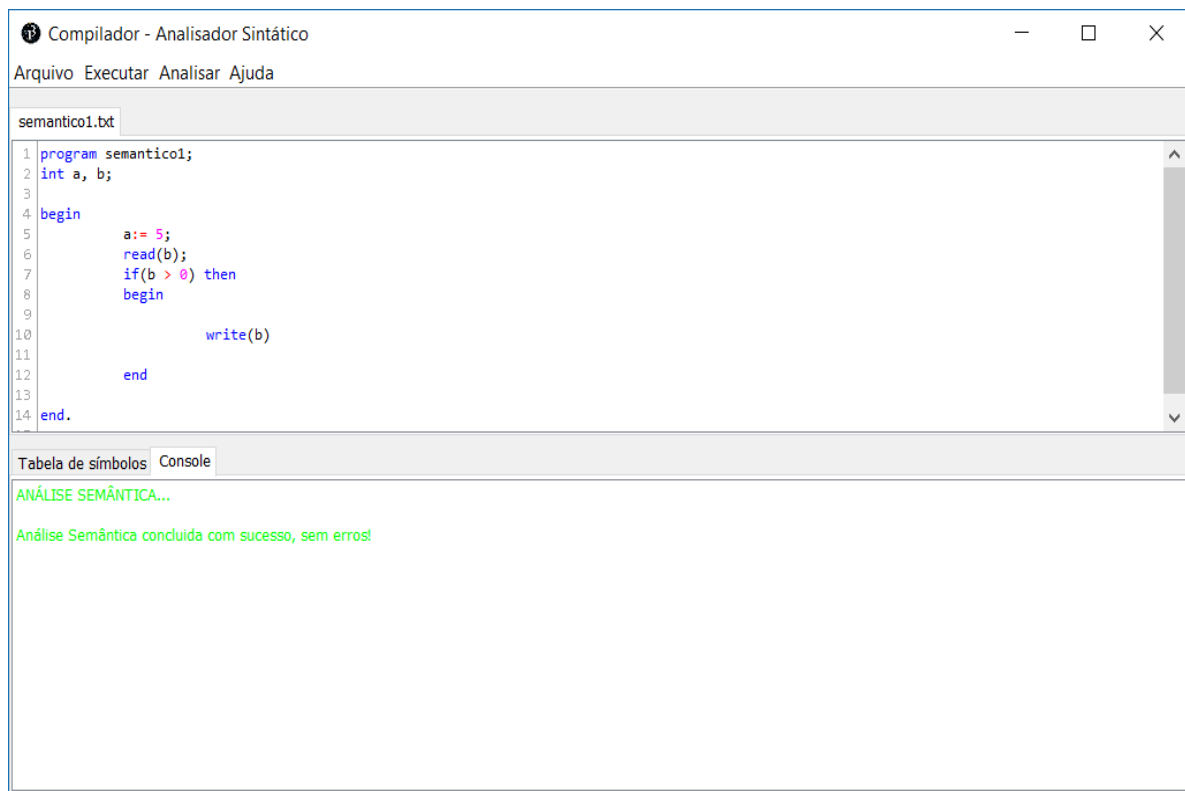


Figura 6 - Teste1: Programa sem erros semânticos

3.2.1 Teste 2: Programa com erro semântico

O segundo teste foi aplicado utilizando o programa semantico2.txt descrito a seguir. Esse programa contém uma sequência de erros semânticos que abrangem praticamente todos os erros semânticos que a aplicação consegue identificar. A seguir é ilustrado o programa semantico2.txt e na Figura 7 é ilustrado os erros semânticos apresentados na execução da aplicação.

```
program semantico2;  
int a, b, c, z, g;  
boolean d, e, f;  
  
procedure proc(var a1, a2 : int);  
int a, b, c, z, zz;
```

```

boolean d, e, f;
begin
    a:=zz;
    if (a<=1) then
        a:=12
    end;

procedure proc1(var a1, a2 : int);
int a, b;
begin

    a := a + 1
end;

begin
    a:=2;
    b:=10;
    c:=z;
    a:=b+c;
    proc(a);
    proc(a,d);
    proc2(b);
    d:=true;
    e:=false;
    f:=true;
    read(a,d);
    write(b);
    read(d,e);
    write(f,d,e,b);
    write(a,b,g);
    if (d) then
        begin
            a:=20;
            b:=10*c;
            c:=a div b
        end;
    while (a>1) do
        begin
            if (b>=10) then
                b:=2;
            a:=a-1
        end
    end.

```

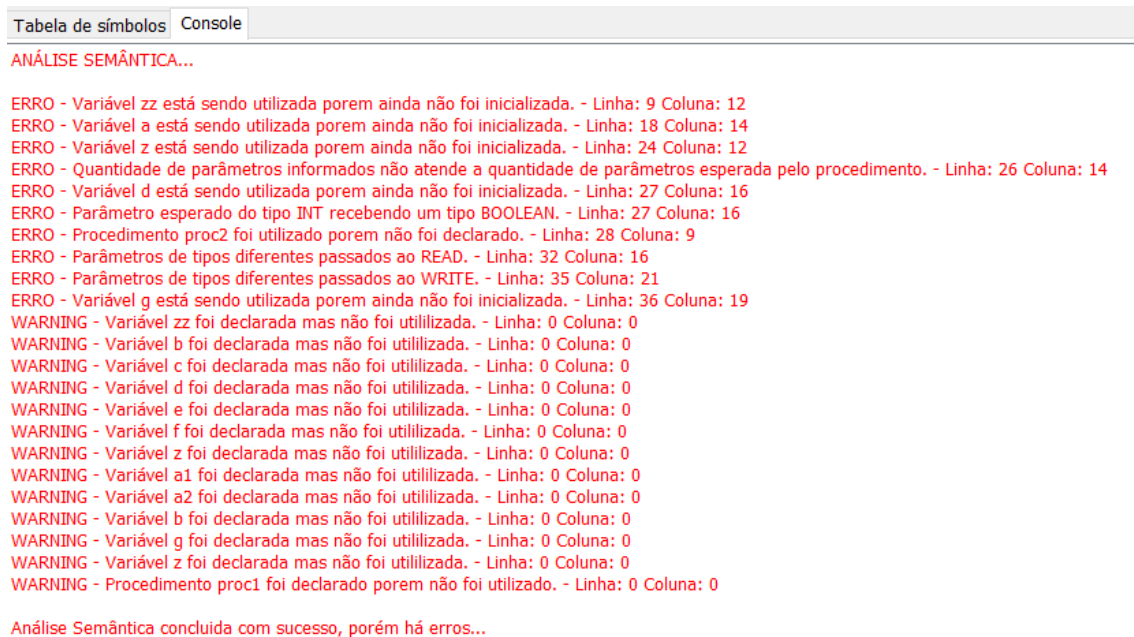


Figura 7 - Instantâneo do Console da ferramenta após a execução do programa semantico2.txt

4 GERAÇÃO E INTERPRETAÇÃO DE CÓDIGO

Após a verificação de erros por parte dos analisadores léxico, sintático e semântico, inicia-se a fase de Síntese de Código, a qual compõe o *back-end* do projeto de um compilador. Neste projeto em específico, foi proposta a geração de um código intermediário, o qual é interpretado pela Máquina de Execução para Pascal (MEPA).

Essa máquina de interpretação utiliza um mecanismo de pilha, a qual contém uma área exclusiva para os dados, em que as variáveis declaradas recebam uma respectiva posição de memória, assim como o processamento de expressões aritméticas ou booleanas; e outra área para o armazenamento das instruções do programa. Para o gerenciamento dessas áreas de memória existe um contador de topo de pilha e um contador de programa, respectivamente.

4.1 Linguagem *BYTECODE*

A linguagem de código intermediário, ou *bytecode*, define as seguintes instruções apresentadas na tabela abaixo:

INSTRUÇÃO	AÇÃO
CRCT c	Carrega o valor da constante c no topo da pilha
CRVL n	Carrega o valor da variável da posição n no topo da pilha
ARMZ n	Armazena o valor do topo da pilha na variável de posição n
SOMA	Realiza a adição dos dois elementos mais ao topo da pilha e empilha novamente
SUBT	Realiza subtração dos dois elementos mais ao topo da pilha e empilha novamente
MULT	Realiza multiplicação dos dois elementos mais ao topo da pilha e empilha novamente
DIVI	Realiza a divisão dos dois elementos mais ao topo da pilha e empilha novamente
MODI	Realiza o módulo dos dois elementos mais ao topo da pilha e empilha novamente
INVR	Realiza a inversão de sinal numérico
CONJ	Realiza a operação binária AND
DISJ	Realiza a operação binária OR
NEGA	Realiza a operação unária NOT
CMME	Comparação se é menor
CMMA	Comparação se é maior
CMIG	Comparação se é igual
CMDG	Comparação se é diferente
CMAG	Comparação se é maior ou igual
CMEG	Comparação se é menor ou igual
DSVS p	Desvio incondicional para o endereço p
DSVF p	Desvio se falso para o endereço p

NADA	Nenhum ação
LEIT	Leitura de inteiro da entrada
LEICH	Leitura de caracter da entrada
IMPR	Impressão de inteiro na saída
IMPC	Impressão de caracter na saída
IMPE	Impressão de uma nova linha
INPP	Início de programa
AMEM n	Alocação de n posições de memória
DMEM n	Desalocação de n posições de memória
PARA	Finalização de programa

4.2 Utilização na aplicação

Para executar a geração do código intermediário, é requerida a execução prévia dos analisadores léxico, sintático e semântico, de forma a garantir a inexistência de erros; sendo localizada no menu Analisar, no item Geração de Código, como apresentado na figura abaixo.

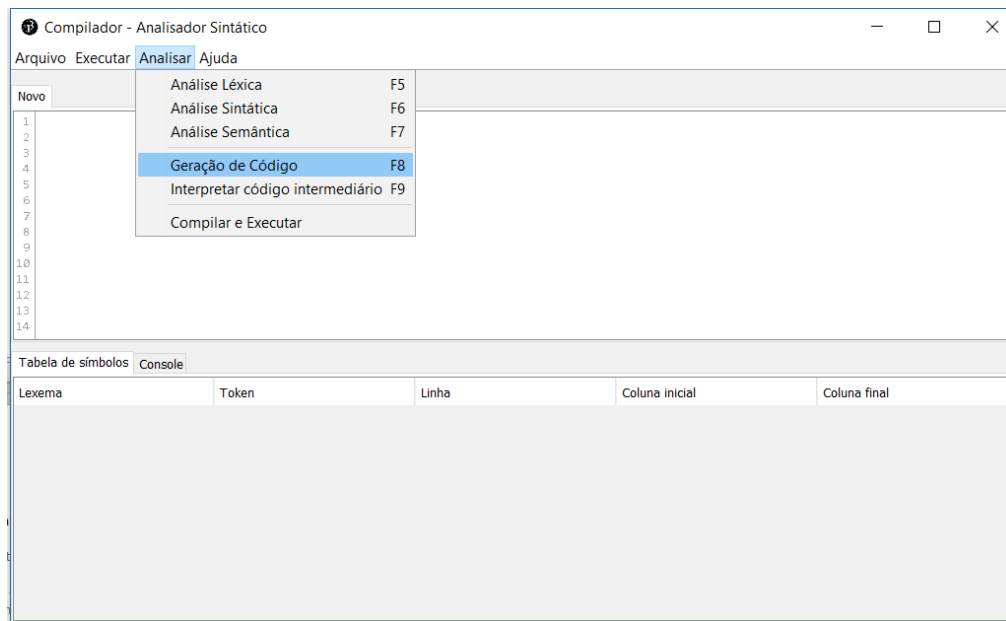


Figura 8 - Instantâneo da aplicação com início da geração de código

Como exemplo de geração de código, foi utilizado o seguinte código fonte:

```

program correto;
int a, b, c;

begin
    a:=2;
    b:=10;
    c:=11;
    a:=b+c;

    write(a);
    if (c >= 11) then
        begin
            b:=10*c;
            write(b);
            c:=a div b;
            write(c)
        end
    else write(c);
    while (a>1) do
        begin
            if (b>=10) then
                b:=2;
            write(a);
            a:=a-1
        end
    end
end

```

end

end.

A partir do código anterior, o seguinte *bytecode* foi gerado:

INPP	ARMZ 1	CRVL 0
AMEM 1	CRVL 1	CRCT 1
AMEM 1	IMPR	SUBT
AMEM 1	CRVL 0	ARMZ 0
CRCT 2	CRVL 1	DSVS 35
ARMZ 0	DIVI	PARA
CRCT 10	ARMZ 2	
ARMZ 1	CRVL 2	
CRCT 11	IMPR	
ARMZ 2	DSVS 35	
CRVL 1	CRVL 2	
CRVL 2	IMPR	
SOMA	CRVL 0	
ARMZ 0	CRCT 1	
CRVL 0	CMMA	
IMPR	DSVF 52	
CRVL 2	CRVL 1	
CRCT 11	CRCT 10	
CMAG	CMAG	
DSVF 33	DSVF 45	
CRCT 10	CRCT 2	
CRVL 2	ARMZ 1	
MULT	CRVL 0	
	IMPR	

Após a geração, para executá-lo basta acionar a opção Analisar -> Interpretar código intermediário.

Adicionalmente a ferramenta fornece a opção Compilar e Executar no menu Analisar, a qual executa todas as etapas de análise e geração de código de forma consecutiva, para melhor usabilidade do usuário. Também é possível importar um *bytecode* externo através da opção Arquivo -> Abrir ... -> Código intermediário, e executá-lo através do item Executar -> Interpretar código intermediário externo.