

**Anhanguera Educacional
FacNet - Faculdade de Negócios e Tecnologia da Informação
Sistemas de Informação**

Brazuka Script

**0991026477 - ALBERTO ARAÚJO COSTA
0991026645 - CYNTIA CRISTINE CAMPOS DIAS
1158366269 - KLEBER BATISTA SOARES DE OLIVEIRA
0991027009 - JAQUELINE OLIVEIRA DE LIMA
0984000089 - LUIZ FERNANDO PERES DE OLIVEIRA**

**Taguatinga
2012**

0991026477 - ALBERTO ARAÚJO COSTA
0991026645 - CYNTIA CRISTINE CAMPOS DIAS
1158366269 - KLEBER BATISTA SOARES DE OLIVEIRA
0991027009 - JAQUELINE OLIVEIRA DE LIMA
0984000089 - LUIZ FERNANDO PERES DE OLIVEIRA

Brazuka Script

Monografia apresentada como exigência
para obtenção do grau de Bacharelado
em Sistemas de Informação da
Anhanguera Educacional.

Orientador: Guilherme Parente Costa

Taguatinga
2012

Dedicamos essa obra a todos os estudantes dos primeiros semestres dos cursos da área de tecnologia da informação.

AGRADECIMENTOS

Agradecemos primeiramente a Deus, por ter nos dado força para superar os obstáculos, depois aos nossos familiares e orientador (Guilherme Parente Costa), que tanto nos apoiaram nos momentos de tomada de decisão, bem como nos deram uma direção para a formação da ideia. Por último, gostaríamos de agradecer a instituição Anhanguera e a Coordenadora (Cíntia Simões de Oliveira) que sempre fez de tudo para que o nosso curso tivesse a melhor qualidade possível.

"UBUNTU - Eu sou, porque nós somos". Antiga palavra africana

RESUMO

Brazuka Script é uma pseudolinguagem de programação que tem o objetivo de facilitar a aprendizagem de lógica computacional para estudantes dos primeiros semestres dos cursos de tecnologia da informação.

Brazuka Script utiliza alguns conceitos famosos para analisar o código e verificar a sua autenticidade em relação à estrutura do algoritmo, como por exemplo, sintaxe e semântica, que foram criadas propositalmente parecidas com a sintaxe e semântica da linguagem C, tendo como objetivo facilitar a migração dos alunos para linguagens como Java e C# posteriormente.

Brazuka Script possui um plugin amigável para a interface de desenvolvimento Eclipse, e após o código brazuka ser escrito e analisado, ocorre a tradução e execução em nível de processamento.

Brazuka Script está inserido dentro do conceito da solução Brazuka, tendo a possibilidade de sincronizar algoritmos em mais de um ambiente de desenvolvimento e plataforma, além de possuir uma rede social de compartilhamento de códigos e experiências.

Obs.: Caso o leitor se interesse em desenvolver a sua própria linguagem baseada na solução, basta ir até o site na aba de "Downloads" e baixar a documentação e o código da linguagem

Palavras-chave: Linguagem de Programação, Analisador, Brazuka Script

ABSTRACT

Brazuka Script is a pseudo programming language that aims to facilitate the learning of computational logic for beginning students of courses in information technology.

Brazuka Script uses some famous concepts to analyze the code and verify the authenticity in relation to the structure of the algorithm, such as syntax and semantics, which were created purposely similar to the syntax and semantics of the language C, aiming to facilitate migration of students to languages like Java and C # later.

Brazuka Script has a friendly plugin interface for the Eclipse environment, and after the code Brazuka be written and analyzed, occurs translation and execution in level of processing.

Brazuka Script is within the solution Brazuka, with the possibility of synchronizing algorithms in more than one development environment and platform, as well as having a social network to share codes and experiences.

Keywords: Programming Language, Analyzer, Brazuka Script

LISTA DE FIGURAS

Figura 1 - Processo de análise de tokens.....	12
---	----

LISTA DE TABELAS

Tabela 1 - Tabela de Léxicos.....	14
-----------------------------------	----

SUMÁRIO

1 INTRODUÇÃO.....	10
2 COMPILADORES.....	11
3 ANALISADOR DE TOKENS.....	12
4 ANALISADOR LÉXICO.....	14
5 ANALISADOR SINTÁTICO.....	19
6 ANALISADOR SEMÂNTICO, GERADOR DE CÓDIGO E EXECUÇÃO.....	23
7 CONCLUSÃO.....	25
REFERÊNCIAS.....	26

1 INTRODUÇÃO

O objetivo dessa obra é de que o leitor seja capaz de entender o funcionamento da teoria dos compiladores (que também podem ser aplicados a tradutores e interpretadores) e do "motor" relacionado à linguagem Brazuka Script. Seremos capazes de distinguir tanto o processo de análise de geração de código-objeto, quanto identificar os passos relacionados à estrutura léxica, sintática e semântica de uma linguagem.

É importante mencionar que Brazuka Script não foi criada com o intuito de ser a mais rápida, mais inteligente e melhor linguagem de programação, mas sim com o intuito de promover o compartilhamento de ideias relacionadas a algoritmos.

A ideia inicial é de que qualquer pessoa possa alterar o "motor" da linguagem criando sua própria solução, bastando ter um conhecimento acerca de estrutura de dados e uma boa lógica de programação, sendo desejável que se conheça a linguagem Java.

Caso o leitor não esteja interessado em criar sua própria solução ou entender da teoria dos compiladores, é sugerido que dê uma olhada diretamente no manual, onde podemos encontrar o guia de utilização da linguagem e do plugin para o ambiente de desenvolvimento Eclipse.

2 COMPILADORES

"Compilador é um programa que traduz um programa-fonte para um programa-objeto", segundo Aho *et al.* (2008) . Dessa forma, um compilador recebe como entrada de dados um programa-fonte em uma linguagem previamente determinada e produz como saída o programa-objeto, ocorrendo um processo de tradução para comandos de máquina. É importante saber que o programa-objeto só é criado caso não exista erros durante o processo de análise e tradução.

Brazuka Script é dividido em algumas fases de análise que ocorrem antes do programa-objeto ser executado, sendo elas:

- Analisador de Tokens;
- Analisador Léxico;
- Analisador Sintático;
- Analisador Semântico, Gerador de Código e Execução.

Geralmente, como foi proposto por Loudon (2004), as fases para tradução de programa-fonte para programa-objeto são: analisador léxico, analisador sintático, analisador semântico e gerador de código.

Na linguagem Brazuka Script, foi separado a identificação de tokens do analisador léxico, e o analisador semântico foi acoplado ao gerador de código, o que será explicado nos próximos capítulos.

3 ANALISADOR DE TOKENS

Antes de entendermos como o analisador de tokens funciona, primeiro precisamos entender que tokens são símbolos representados por padrões definidos em determinada linguagem. Ou seja, através de uma *máquina de estados*, verificamos qual o próximo caractere válido e o guardamos numa lista encadeada de tokens.

A validação da *máquina de estados* do analisador de tokens é feita caractere a caractere e ocorre da seguinte maneira:

Se o caractere "lookahead" (o próximo caractere) fizer parte do padrão existente, então o analisador adiciona o caractere ao token e incrementa a posição do lookahead. Caso o lookahead seja vazio (espaços, tabulações e etc) ou não fizer parte do padrão esperado, então o token é adicionado na lista de tokens e o analisador reinicia o conteúdo do token esperando novos padrões, como mostra a figura:

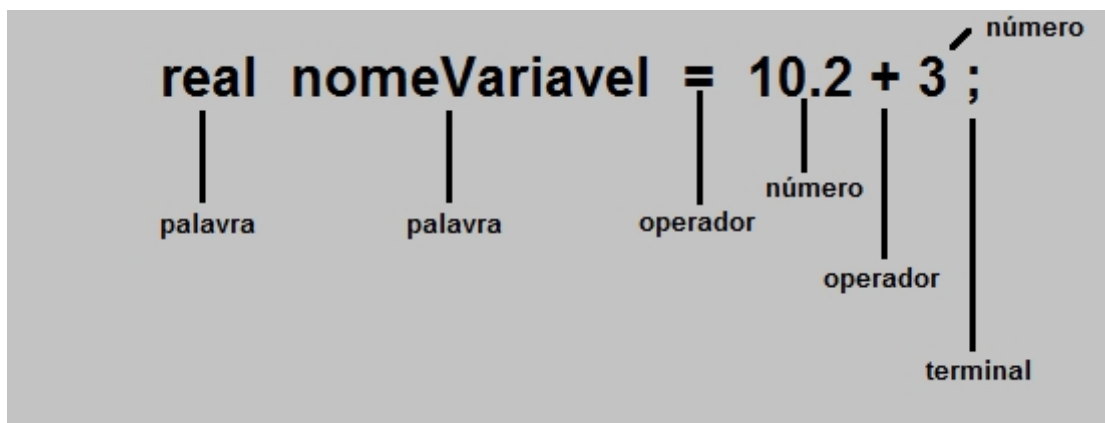


Figura 1 - Processo de análise de tokens

Geralmente, um token pode guardar ou não informações a respeito de suas características.

Em Brazuka Script, um token guarda informações internas, e elas são: gênero do token, tipo do token, valor do token e número da linha.

- **Gênero do Token:** É setado quando conhecemos o token pela primeira vez. O gênero do token pode ser GT_PALAVRA, GT_NUMERO, GT_OPERADOR ou GT_CHARACTER_TERMINAL.

- **Tipo do Token:** É setado no analisador léxico. Representa o valor do token sintaticamente falando, por exemplo TT_FIM_COMANDO, que é setado toda vez que encontramos (na análise léxica) a ocorrência do caractere ponto-e-vírgula.

- **Valor do Token:** O valor literal do token.

- **Número da Linha:** O número da linha em que o token se encontra.

Dessa forma, ao fim da execução do analisador de tokens na Figura 1, a lista de tokens terá os tokens:

- Token { GT_PALAVRA, null, "real", 1 },
- Token { GT_PALAVRA, null, "nomeDaVariavel", 1},
- Token { GT_OPERADOR, null, "=", 1},
- Token { GT_NUMERO, null, "10.2", 1},
- Token { GT_OPERADOR, null, "+", 1},
- Token { GT_NUMERO, null, "3", 1},
- Token { GT_CHARACTER_TERMINAL, null, ";", 1}

Obs.: O primeiro parâmetro é o gênero do token, o segundo parâmetro é o tipo do token, o terceiro parâmetro é o valor do token e o último parâmetro é o número da linha que se encontra o token.

As informações citadas acima são guardadas nos tokens para que os analisadores posteriores possam agrupá-los de forma a gerar valor às sentenças do programa-fonte, ou seja, até agora a nossa lista de tokens não passa de uma lista cheia de caracteres separados por padrões, mas que não geram nenhum valor.

4 ANALISADOR LÉXICO

O objetivo principal do analidior léxico é manipular a lista de tokens que veio do analisador de tokens e acrescentar valor para enviar ao analisador sintático.

Brazuka Script possui uma tabela de léxicos, e é a partir dela que o analisador léxico acrescenta valor. Os léxicos funcionam como palavras reservadas da linguagem, ou seja, toda vez que o analisador léxico verifica a ocorrência de uma palavra ou símbolo que ele conhece, ele preenche o token com a informação do tipo do mesmo (vide Analisador de Tokens).

Tabela 1 - Tabela de Léxicos

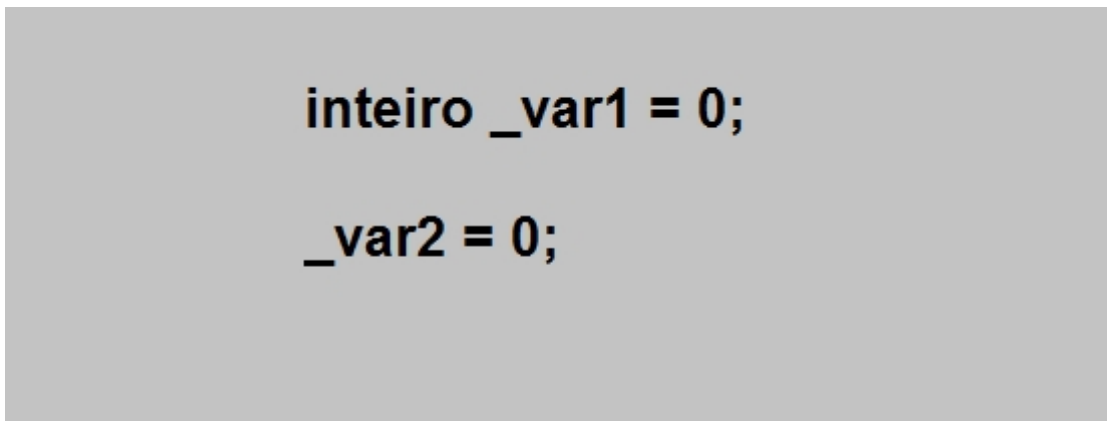
#inclusao	TT_INCLUSAO
#definir	TT_CONSTANTE
funcao	TT_FUNCAO
procedimento	TT_PROCEDIMENTO
vazio	TT_TIPO_DE_DADOS
inteiro	TT_TIPO_DE_DADOS
real	TT_TIPO_DE_DADOS
caracter	TT_TIPO_DE_DADOS
texto	TT_TIPO_DE_DADOS
logico	TT_TIPO_DE_DADOS
retorna	TT_RETORNO
se	TT_CONDICAO
senao	TT_CONDICAO_SENAO
enquanto	TT_ENQUANTO
faca	TT_FACA_ENQUANTO
para	TT_PARA
(TT_ABRE_PARENTESE
)	TT_FECHA_PARENTESE
{	TT_ABRE_CHAVE
}	TT_FECHA_CHAVE
[TT_ABRE_COLCHETE
]	TT_FECHA_COLCHETE
;	TT_FIM_COMANDO
,	TT_SEPARA_VARIAVEL
<	TT_OPERADOR_RELACIONAL
>	TT_OPERADOR_RELACIONAL

&	TT_ENDERECO_VARIAVEIS
+	TT_OPERADOR_MATEMATICO_1
-	TT_OPERADOR_MATEMATICO_1
*	TT_OPERADOR_MATEMATICO_2
/	TT_OPERADOR_MATEMATICO_2
%	TT_OPERADOR_MATEMATICO_2
leia	TT_LEIA
escreva	TT_ESCREVA
verdadeiro	TT_LOGICO
falso	TT_LOGICO
"	TT_ASPAS_DUPLAS
'	TT_ASPAS_SIMPLES
++	TT_INCREMENTO
--	TT_INCREMENTO
+=	TT_CALCULO_E_ATRIBUICAO
-=	TT_CALCULO_E_ATRIBUICAO
*=	TT_CALCULO_E_ATRIBUICAO
/=	TT_CALCULO_E_ATRIBUICAO
%=	TT_CALCULO_E_ATRIBUICAO
=	TT_ATRIBUICAO
==	TT_OPERADOR_RELACIONAL
>=	TT_OPERADOR_RELACIONAL
<=	TT_OPERADOR_RELACIONAL
e	TT_OPERADOR_LOGICO
ou	TT_OPERADOR_LOGICO
!	TT_OPERADOR_LOGICO_NEGACAO
!=	TT_OPERADOR_RELACIONAL
.	TT_EXTENSAO_BIBLIOTECA
?	TT_TERNARIO
:	TT_DOIS_PONTOS
principal	TT_FUNCAO_PRINCIPAL
escolha	TT_ESCOLHA
caso	TT_CASO
padrao	TT_PADRAO
pula	TT_PULA
continua	TT_CONTINUA
aleatorio	TT_ALEATORIO
limpatela	TT_LIMPA_TELA
Ocorre em nível de código	TT_DECLARACAO_FUNCAO
Ocorre em nível	TT_DECLARACAO_VARIAVEL

de código	
Ocorre em nível de código	TT_NUMERO_CONSTANTE
Ocorre em nível de código	TT_PALAVRA_CONSTANTE
Ocorre em nível de código	TT_CHARACTER_CONSTANTE
Ocorre em nível de código	TT_CHAMA_FUNCAO
Ocorre em nível de código	TT_CHAMA_VARIAVEL

Como dito anteriormente, léxicos são os tokens conhecidos da linguagem, porém nem todo token é um léxico, já que um token pode conter um valor desconhecido.

Um token desconhecido pela linguagem pode virar um léxico da linguagem caso ele seja declarado em nível de código, como você deve ter percebido na tabela de léxicos, ou seja, na verdade os tokens desconhecidos podem obter valor caso o seu conteúdo seja explicitamente informado pelo usuário no programa-fonte, como nome de variáveis, funções e constantes. Veja a imagem a seguir:



```
inteiro _var1 = 0;
_var2 = 0;
```

Figura 2 - Validação Analisador Léxico

Ao fim da execução do analisador léxico na Figura 2, teremos os tokens da linha 1 com os tipos dos tokens preenchidos:

- Token { GT_PALAVRA, TT_TIPO_DE_DADOS, "inteiro", 1 },
- Token { GT_PALAVRA, TT_DECLARACAO_VARIAVEL, "_var1", 1 },
- Token { GT_OPERADOR, TT_ATRIBUICAO, "=", 1 },
- Token { GT_NUMERO, TT_NUMERO_CONSTANTE, "0", 1 },

Porém, como você deve ter percebido, a linha 2 da Figura 2 no token "_var2" irá gerar uma exceção de erro léxico, já que a variável não foi declarada pelo usuário, ou seja, a linguagem não conhece a palavra.

Por último, devemos compreender as competências do analisador léxico, que são:

- Verificar léxicos da linguagem.
- Verificar novos léxicos que o usuário criou.
- Verificar se já existe variável declarada com o mesmo nome no bloco de código corrente.
- Verificar se já existe função declarada com o mesmo nome da variável criada pelo usuário.

Obs.: Para ser enviado ao analisador sintático, **TODOS** os tokens da lista de tokens devem possuir um léxico válido. Caso algum token não seja preenchido, então o analisador léxico irá gerar uma exceção na linha do token que provocou o erro.

5 ANALISADOR SINTÁTICO

O objetivo do analisador sintático é verificar se uma expressão do programa-fonte obedece às regras de formação de uma dada gramática. Desta forma, o analisador sintático cuida exclusivamente da forma das sentenças da linguagem, baseando-se na gramática que define a linguagem.

O analisador sintático geralmente transforma as expressões da linguagem em uma árvore sintática (como Fabiano Baldo sugere (2012)), que é uma representação gráfica da derivação de sentenças gramaticais.

Ex: (num + num) * num

<i>expr</i>	\longrightarrow	<i>expr op expr</i>
<i>expr</i>	\longrightarrow	'(' <i>expr</i> ')'
<i>expr</i>	\longrightarrow	<i>id</i>
<i>expr</i>	\longrightarrow	<i>num</i>
<i>op</i>	\longrightarrow	+ - * /

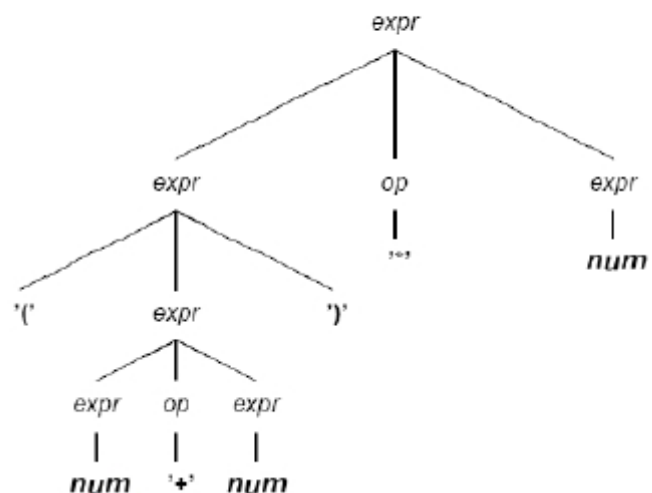


Figura 3 - Árvore de Análise Sintática

Segundo Baldo *et al.* (2012), o princípio da tradução dirigida por sintaxe estabelece que o significado da sentença está diretamente relacionado a sua estrutura sintática representada na árvore. Assim, a posição de cada token na árvore indica sua relação com os outros tokens reconhecidos. Caso a árvore não obedeça ao padrão sintático ao fim da próxima expressão, então será gerado um erro sintático.

Analizador Sintático Brazuka Script

Em Brazuka Script, o analisador sintático não utiliza uma árvore sintática, como a maior parte das literaturas sugerem. Em vez disso, o analisador sintático reaproveita a lista encadeada de tokens e utiliza uma máquina de estados para saber qual o próximo estado válido de acordo com uma regra pré-definida da linguagem. Esse processo é muito parecido com a estrutura de árvore sintática e a simula perfeitamente. Veja a figura a seguir:

```
inteiro x = 2;  
se(x > 0)  
{  
    escreva("x maior que zero.");  
}
```

Figura 4 - Analizador Sintático Brazuka

Ao fim do processo de análise de tokens e análise léxica, teremos a lista de tokens que estarão preparados para serem analisados sintaticamente.

// Linha 1

- Token { GT_PALAVRA, TT_TIPO_DE_DADOS, "inteiro", 1 },
- Token { GT_PALAVRA, TT_DECLARACAO_VARIAVEL, "x", 1 },
- Token { GT_OPERADOR, TT_ATRIBUICAO, "=", 1 },
- Token { GT_NUMERO, TT_NUMERO_CONSTANTE, "2", 1 },
- Token { GT_CHARACTER_TERMINAL, TT_FIM_COMANDO, ";", 1 },

// Linha 2

- Token { GT_PALAVRA, TT_CONDICAO, "se", 2 },
- Token { GT_CHARACTER_TERMINAL, TT_ABRE_PARENTESE, "(", 2 },
- Token { GT_PALAVRA, TT_CHAMA_VARIAVEL, "x", 2 },

- Token { GT_OPERADOR, TT_OPERADOR_RELACIONAL, ">", 2},
- Token { GT_NUMERO, TT_NUMERO_CONSTANTE, "0", 2},
- Token { GT_CHARACTER_TERMINAL, TT_FECHA_PARENTESE, ")", 2},

// Linha 3

- Token { GT_CHARACTER_TERMINAL, TT_ABRE_CHAVE, "{", 3},

// Linha 4

- Token { GT_PALAVRA, TT_ESCREVA, "escreva", 4},
- Token { GT_CHARACTER_TERMINAL, TT_ABRE_PARENTESE, "(", 4},
- Token { GT_PALAVRA, TT_PALAVRA_CONSTANTE, "x maior que zero", 4},
- Token { GT_CHARACTER_TERMINAL, TT_FECHA_PARENTESE, ")", 4},
- Token { GT_CHARACTER_TERMINAL, TT_FIM_COMANDO, ";", 4},

// Linha 5

- Token { GT_CHARACTER_TERMINAL, TT_FECHA_CHAVE, "}", 5},

A análise e validação sintática dos léxicos, ocorre utilizando uma *máquina de estados*.

Máquina de Estados

Uma máquina de estados finitos (FSM - do inglês Finite State Machine) ou autômato finito é um modelo matemático usado para representar programas de computadores ou circuitos lógicos. O conceito é concebido como uma máquina abstrata que deve estar em um de seus finitos estados. A máquina está em apenas um estado por vez, este estado é chamado de estado atual. Um estado armazena informações sobre o passado, isto é, ele reflete as mudanças desde a entrada num estado, no início do sistema, até o momento presente. Uma transição indica uma mudança de estado e é descrita por uma condição que precisa ser realizada para que a transição ocorra. Uma ação é a descrição de uma atividade que deve ser realizada num determinado momento. Black, Paul E. (2008).

Exemplo - Analisador Sintático Brazuka Script e Máquina de Estados

Utilizando o primeiro token da primeira linha da lista de tokens da Figura 4, temos a máquina de estados no primeiro estado ("estado padrão" ou "estado inicial"). Então, o analisador sintático verifica quais os próximos estados válidos após um TT_TIPO_DE_DADOS. Se o próximo estado estiver dentro das possibilidades do estado anterior, então o analisador sintático continua mudando o estado da máquina. Caso o estado não esteja dentro das definições do estado anterior, então o sistema gera um erro sintático.

Nesse exemplo, a máquina de estados no primeiro momento espera por um léxico do tipo TT_DECLARACAO_VARIAVEL obrigatoriamente, já que depois de um TT_TIPO_DE_DADOS (quando a máquina de estados estiver no estado inicial) deve vir um léxico referente a uma declaração de variável. Após isso, a máquina de estados irá procurar por um padrão de léxicos que esteja entre TT_ATRIBUICAO, TT_FIM_COMANDO, TT_ABRE_COLCHETE e TT_SEPARA_VARIAVEL. Como o próximo léxico é um TT_ATRIBUICAO, então a máquina de estados recebe o próximo estado, em que ela aceita uma expressão qualquer, exemplo: "2 + 3 * (5 - 4)". Na primeira linha, após o TT_ATRIBUICAO, o próximo léxico é equivalente ao número 2, e então a máquina de estados irá procurar um operador matemático, relacional, incremento ou um TT_FIM_COMANDO. Como o último léxico da expressão é o TT_FIM_COMANDO, então a máquina de estados é reiniciada para começar outra validação sintática.

Obs.: Toda vez que o analisador sintático verifica um caractere terminal, o ponto-e-virgula por exemplo, ele reinicia a máquina de estados, ou seja, ele faz com que a máquina de estados volte ao estado inicial.

Obs².: A lista de tokens só é enviado para o analisador semântico e gerador de código caso não haja nenhum erro de token, léxico ou sintático.

6 ANALISADOR SEMÂNTICO, GERADOR DE CÓDIGO E EXECUÇÃO

Analizador Semântico

A análise semântica é responsável por validar os tipos e valores das variáveis, por exemplo, uma cadeia de caracteres não pode ser atribuída a um inteiro.

Também é papel do analisador semântico a verificação dos tipos passados nos parâmetros das funções e outros, ou seja, o analisador semântico se preocupa com os valores da linguagem e não mais com o formato em que a linguagem está escrita (analisador sintático).

Em Brazuka Script, não é feita explicitamente a análise semântica, porque como ocorre a tradução do algoritmo para java, não se teve a preocupação de fazer o esse analisador, já que bastou-se validar o analisador semântico da própria linguagem java. Ou seja, após a geração de código, o próprio javac irá informar se ocorreu erro semântico ou não.

Gerador de Código

A parte de geração de código é a parte mais simples da linguagem, já que garantimos que a linguagem está bem escrita por ter passado pelos analisadores anteriores. Como dito acima, a análise semântica só é realizada depois de que o programa-fonte já foi traduzido para linguagem Java, porém, antes disso, os tokens são transformados em tokens compatíveis com Java. Veja a imagem a seguir:

```
#definir CONSTN 1
static final CONSTN = (float) 1;

inteiro x = n3 + n5++;
int x = n3 + n5++;
```

A imagem mostra um exemplo de como um token de definição de constante em uma linguagem é traduzido para o código Java. O token original é '#definir CONSTN 1'. A tradução para Java é 'static final CONSTN = (float) 1;'. Abaixo, há um exemplo de uso da constante: 'inteiro x = n3 + n5++;' e sua tradução para Java: 'int x = n3 + n5++;'. O texto está em um fundo cinza escuro com cores de destaque: o token original em branco, a tradução em verde e o exemplo de uso em branco.

Figura 5 - Geração de Código

Execução

Após ser gerado o arquivo Java, é preciso agora compilar o mesmo e rodá-lo.

O "motor" da linguagem tenta compilar o arquivo, caso não consiga, então é considerado um erro semântico e é passado à linguagem Brazuka para que a mesma trate o erro e gere a exceção para o usuário. Caso ocorra tudo certo, então o "motor" executa a classe criada via Reflection.

7 CONCLUSÃO

Ao fim dessa obra, percebemos que para desenvolver uma linguagem de programação, não é necessário que se siga a teoria tão a risca quanto vemos nas gramáticas de referência, mas é claro que se queremos uma linguagem mais poderosa, é bom começar lendo muito a respeito do tema e processos de desenvolvimento de compiladores, além de entender melhores práticas para ganhar performance.

Devemos lembrar também que Brazuka Script é uma linguagem que foi criada para ser o mais didática possível, não pensando em nível de melhores métodos para poupar processamento. E descobrimos também que Brazuka Script na realidade utiliza as três formas de manipulação de linguagens (tradução, compilação e interpretação, respectivamente) e conseguimos com esse documento entender o processo de desenvolvimento de compiladores e da linguagem tratada no tema.

Com isso, podemos perceber que Brazuka Script pode com certeza contribuir com o aprendizado de alunos dos primeiros semestres, já que a solução engloba também sincronismo de arquivos para o usuário, além de disponibilizar uma rede social para compartilhamento de ideias e experiências.

REFERÊNCIAS

(20/05/2002), J. W. C. T. E. A. F. S. **VAMOS CONSTRUIR UM COMPILADOR - Análise Léxica**. <http://compiladores.osdevbrasil.net/tutor07.htm>: OS DEV BRASIL, 2002. 1p.

., **Análise Léxica**.

https://www.google.com.br/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=0CD AQFjAB&url=https%3A%2F%2Fcompilador-pam.googlecode.com%2Ffiles%2F2-Analise_lexica.pdf&ei=ilyrUPK_IoTe8ASCglGgDw&usg=AFQjCNEJbmqqys1PWakezEg xzgmxndKr6Q&cad=rja: , 2012. 45p.

., **Plugin Eclipse**. Code Google: Velo Eclipse, 2009. 1p. Disponível em:

<<http://code.google.com/p/veloeclipse/source/browse/trunk/veloeclipse/src/com/googlecode/veloeclipse/vaulttec/ui/editor/text/DirectiveDetector.java?spec=svn07b5392c9f8c99b6fe12e3ed34dd35b91c27bf5f&r=07b5392c9f8c99b6fe12e3ed34dd35b91c27bf5f>> Acesso em: 1 jan. 0001

., **Geração de Código Intermediário - Análise Semântica**.

<http://www.inf.ufrgs.br/~nicolas/pdf/Compiladores14b-tac1.pdf>: INF UFRGS, 2011. 6p.

AIKEN, P. A. **Análise Sintática**.

<http://www.youtube.com/watch?v=A0s-FuZne7o&feature=relmfu>: Vídeo, 2012. 0p.

BALDO, F. **Compiladores - Análise Sintática**. 2012. 32f. Monografia (Bacharelado em Ciências da Computação) - UDESC, http://www2.joinville.udesc.br/~coxa/cursos/compiladores/2008-2/material/Analise_sintatica.pdf, 2012.

BECKER, J. **Ícones plugin Eclipse**. <http://tech.joelbecker.net/>: Artigos Tech Joel Becker, 2010. 1p. Disponível em:

<<http://tech.joelbecker.net/articles/resources/5-eclipseicons>> Acesso em: 29 out. 2012

BLACK, P. E. **Finite State Machine**. Dictionary of Algorithms and Data Structures.: U.S. National Institute of Standards and Technology., 1999. 21p. Disponível em:

<<http://xlinux.nist.gov/dads/HTML/finiteStateMachine.html>> Acesso em: 1 jan. 0001

BLASS, S. **Plugin Eclipse**. Network World: Network World, 2007. 2p. Disponível em: <<http://www.networkworld.com/columnists/2007/082907-dr-internet.html>> Acesso em: 26 out. 2012

C., D. **Análise Semântica.**

<http://www.inf.ufrgs.br/gppd/disc/cmp135/trabs/martinotto/trabII/semantica.htm>: INF UFRGS, 2012. 1p.

COIMBRA, B. B. **Introdução aos compiladores.**

<http://blog.bbcoimbra.com/2011/09/18/introducao-aos-compiladores.html>: FATEC-SP, 2011. 60p. Disponível em:

<<http://blog.bbcoimbra.com/static/coimbra-compilers-2011.pdf>> Acesso em: 1 jan. 0001

COURSERA, **Class Coursera Compilers.** <https://class.coursera.org/compilers>: Coursera, 2012. 50p.

CRENSHAW, J. W. **Compiladores.** <http://compiladores.osdevbrasil.net/>: OS DEV BRASIL, 1995. 20p.

DEVMEDIA, **Introdução ao Eclipse RCP.** DevMedia: DevMedia, 2011. 2p.

Disponível em: <<http://www.devmedia.com.br/introducao-ao-eclipse-rcp/4352>> Acesso em: 16 nov. 2012

DEVMEDIA, **Java Compiler Compiler - JavaCC.** <https://javacc.dev.java.net/>:

DevMedia, 2010. 6p. Disponível em:

<<http://www.devmedia.com.br/conheca-agora-e-baixe-o-plugin-do-javacc-para-eclipse/2311>> Acesso em: 3 out. 2012

ECHEVARRIA, **Eclipse com JavaCC.** BLOG DO ECHEVARRIA: E CHEVARRIA, 2008. 5p. Disponível em:

<<http://blogdoechevarria.blogspot.com.br/2008/05/eclipse-com-javacc.html>> Acesso em: 3 out. 2012

ERECHIM, **Apostila de Compiladores.**

<http://www.ebah.com.br/content/ABAAAAdnMAK/apostila-compiladores>: Ebah, 2001. 41p.

G.R, P. **Ícones Plugin Eclipse.** Blog Cypal Solutions: Eclipse Tips, 2008. 1p.

Disponível em: <<http://blog.cypal-solutions.com/2008/02/eclipse-icons.html>> Acesso em: 31 out. 2012

GALLARDO, D. **Desenvolvendo Plugin Eclipse.** .: DeveloperWorks Brasil, 2002.

9p. Disponível em: <<http://www.ibm.com/developerworks/br/library/os-ecplug/>>
Acesso em: 21 nov. 2012

MINOCHA, S. **Eclipse RCP**. IBM: DeveloperWorks Brasil, 2006. 2p. Disponível em:
<<http://www.ibm.com/developerworks/br/library/os-ecl-rcpapp/index.html>> Acesso
em: 16 nov. 2012

RAMALHO, J. C. L. **Compiladores Incrementais**.
http://www3.di.uminho.pt/~jcr/COMMS/sngen/sngen_acet.html: Di UMINHO, 2001. 6p.

RICARTE, I. L. M. **Análise Léxica**.
<http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node45.html>: DCA
UNICAMP, 2003. 4p.

RICARTE, I. L. M. **Análise Semântica**.
<http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node1.html>: DCA
UNICAMP, 2003. 60p.

ROSSINI., C. L. G. L. E. S. **Implementação da análise semântica**.
<http://www.google.com.br/url?sa=t&rct=j&q=codigo%20analizador%20sintatico&source=web&cd=4&ved=0CGwQFjAD&url=http%3A%2F%2Fcompiladormarvel.googlecode.com%2Ffiles%2FCompiladores%2520pt3.doc&ei=xjO9T5eYCMbZgQffmf35Dg&usg=AFQjCNG-mQIUZhVdpLcgariy-ZWRELbd2g&cad=rja>: UFJF, 2007. 9p.

SA, T. **Plugin Eclipse - How To Implement**. ,: Talend Inc, 2007. 1p. Disponível em:
<http://www.talendforge.org/svn/tos/tags/release-2_2_3/org.talend.commons/src/main/java/org/talend/commons/ui/swt/colorstyledtext/scanner/ColoringScanner.java>
Acesso em: 19 nov. 2012

SORANZ, F. **Tutorial Compiladores**.
<http://tutorialcompiladores.pbworks.com/w/page/22541625/Introdu%C3%A7%C3%A3o>: PB Works, 2011. 16p.

TULER, G. M. A. C. T. P. E. D. **Lua Eclipse - Plugin Eclipse**. LuaForge: LuaForge
Net, 2012. 29p. Disponível em: <<http://luaeclipse.luaforge.net/br/manual.html>>
Acesso em: 19 nov. 2012

UFCG, **AUTOMATO FINITO E ANALISE LEXICA**. 2005. 24f. Monografia
(Bacharelado em Ciências da Computação) - UFCG,
<http://www.dsc.ufcg.edu.br/~peter/cursos/cc/material/p2-gram&lex-2p.pdf>, 2005.

UFPE, C. **COMPILADORES**. 2001. 60f. Monografia (Bacharelado em Ciências da Computação) - URICER – Universidade Regional Integrada do Alto Uruguai, <http://www.cin.ufpe.br/~pftbm/apostila-LFeC-II.pdf>, 2001.

ULLMAN, R. S. J. D. **Compiladores - Princípios, Técnicos e Ferramentas** . . : Prentice Hall, 2008. 648p.