

# Analizador Gramatical Genérico - Analyzer

Raimundo Botelho, Israel Monteiro e Michael Salzer

Centro Universitário do Pará (CESUPA)

Av. Gov. José Malcher, 1963 – 66.060-230 – Belém – PA – Brasil

raimundo.botelho@gmail.com, israelwb@msn.com, salzerbcc@yahoo.com.br

**Abstract.** *This article presents the architecture and use of a Generic Grammatical Analyzer. It is a framework in Java that is designed to allow users to build, quickly and easily, compilers and interpreters without the need to implement all the complex steps of a translator. The article begins speaking of the purpose of the tool, the architecture and finishes, showing a practical example of how to implement an arithmetic expressions interpreter with the framework.*

**Resumo.** *Este artigo apresenta a arquitetura e a utilização de um Analisador Gramatical Genérico. Trata-se de um framework em Java que foi desenvolvido para permitir que usuários construam, de maneira rápida e fácil, compiladores e interpretadores sem a necessidade de implementar todas as etapas complexas de um tradutor. O artigo começa falando do objetivo da ferramenta, da arquitetura e finaliza, mostrando um exemplo prático de como implementar um interpretador de expressões aritméticas com o framework.*

## 1. Introdução

A disciplina de Compiladores dentro da grade curricular do curso de Ciência da Computação, objetiva prover o bacharelado de conhecimentos para construção de compiladores, assim como a percepção das diversas etapas envolvidas no processo de compilação. Porém, o domínio das técnicas para construção de compiladores e interpretadores exige o estudo e conhecimento de outras disciplinas afins, tais como Linguagens Formais e Autômatos, Teoria da Computabilidade além de conceitos de análise de fluxo de dados e gerência de memória. Tanta exigência torna a disciplina complexa, causando muitas vezes desinteresse pelo seu completo aprendizado e deixando lacunas no currículo do estudante. Porém quando dominado esses princípios e técnicas, são tão penetrantes, que serão usadas muitas vezes na carreira de um cientista da computação.

Saindo do plano acadêmico em direção ao plano profissional, o tema construção de compiladores pode parecer distante da realidade dos desenvolvedores de sistemas informatizados. No entanto, frequentemente profissionais enfrentam dificuldades para implementar algum interpretador de expressões ou um tradutor para uma pequena linguagem. Mesmo que tenham proficiência na construção dos mesmos, é sempre muito complicado e demorado desenvolver um analisador gramatical para determinada tarefa, pois sempre causará impacto na produtividade, fator de exigência preponderante dentro das empresas hoje em dia.

Diante dessas circunstâncias, foi apresentada em um Trabalho de Curso de Bacharelado de Ciência da Computação do Centro Universitário do Pará - CESUPA, uma ferramenta batizada como Analyzer. Trata-se de um analisador gramatical genérico. O Analyzer tem a pretensão não só de auxiliar os alunos de Ciência da Computação no aprendizado da disciplina de Compiladores, mas também, de acompanhá-los em suas vidas profissionais quando necessitarem de uma ferramenta de fácil utilização para a construção de analisadores gramaticais como tradutores, compiladores, interpretadores, sites de buscas, algoritmos para mineração de texto e etc.

## 2. Apresentação da ferramenta

O Analyzer é um framework com a funcionalidade de um analisador gramatical LL(1) genérico - desenvolvido em Java - que pode ser utilizado para implementar compiladores e interpretadores de pequeno e médio porte de maneira simples e rápida.

Internamente o Analyzer possui um analisador léxico, um analisador sintático preditivo tabular e uma API para dar suporte a implementação da análise semântica, geração de código intermediário e geração de código objeto.

Um dos benefícios que o Analyzer busca oferecer aos usuários é a facilidade de se desenvolver um analisador gramatical completo com o mínimo de esforço possível.

Para a análise léxica e sintática é necessário apenas configurar um arquivo XML com as principais características da linguagem da qual se estiver fazendo uso sem precisar inserir uma única linha de código em Java.

Nas demais etapas: análise semântica, geração de código intermediário e geração de código objeto - denominada pelo framework de computação, o usuário poderá contar com a flexibilidade e a facilidade de utilização da API do Analyzer para dar suporte ao processamento do resultado final da análise.

## 3. Arquitetura

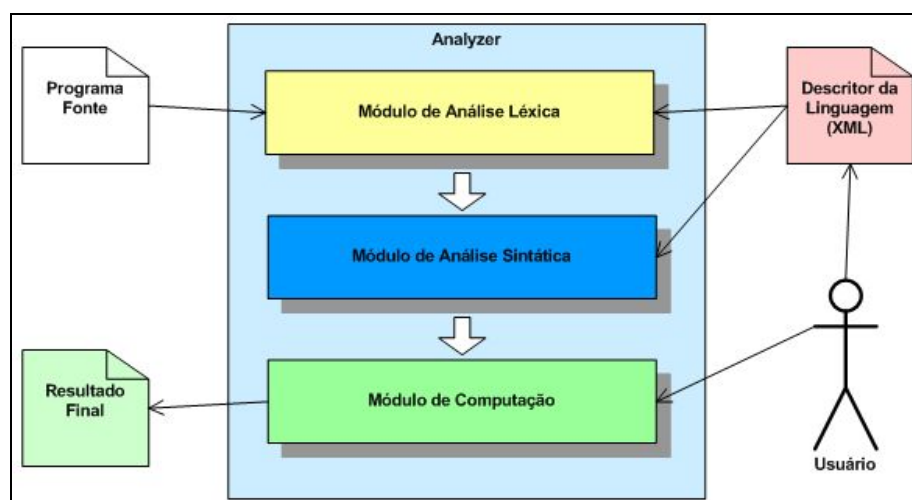


Figura 1. Arquitetura do Analyzer

A arquitetura do Analyzer possui três módulos básicos (Figura 1) que são responsáveis pelas etapas de análise e síntese do framework, apoiados pelo Descritor da Linguagem e pela API de Suporte a Computação. Esses módulos se comportam de acordo com as regras da linguagem que são definidas pelo usuário em um arquivo XML no descritor da linguagem.

### 3.1. Módulo de Análise Léxica:

É o módulo responsável pelo reconhecimento das menores unidades de um código fonte que são os tokens.

No framework existem vários tokens pré-definidos para número inteiro, real, lógico e string, porém se o usuário preferir poderá definir seus próprios tokens através de expressões regulares no Descritor da Linguagem.

### 3.2. Módulo de Análise Sintática:

Esse módulo realiza a análise estrutural do programa de acordo com a Gramática Livre de Contexto (GLC) que também é definida pelo usuário no Descritor da Linguagem.

### 3.3. Módulo de Computação:

Responsável pela fase de síntese do framework. Esse módulo gera um resultado significativo para o programa de acordo com a implementação do usuário através da API de Suporte a Computação do Analyzer.

No módulo de computação pode ser feita a análise semântica, geração de código intermediário, geração de código objeto e outras opções que o usuário achar necessário para a geração do resultado.

### 3.4. Descritor da Linguagem:

Trata-se de um arquivo XML (Figura 2) onde o usuário definirá os tokens, as regras de produções e o mapeamento dessas regras para a API de Suporte a Computação.

O Descritor da Linguagem é dividido em duas sessões: Sessão de Análise Léxica e Sessão de Análise Sintática.

```
<?xml version="1.0" encoding="UTF-8"?>
<analyzer>
  <language-name>Expression</language-name>
  <lexical>
    <tokens>
      <real/>
      :
    </tokens>
  </lexical>
  <syntactic>
    <context-free-grammar>
      <not-terminal name="expression" method="logicalOperation">
        <productions>
          :
        </productions>
      </not-terminal>
    </context-free-grammar>
    :
  </syntactic>
</analyzer>
```

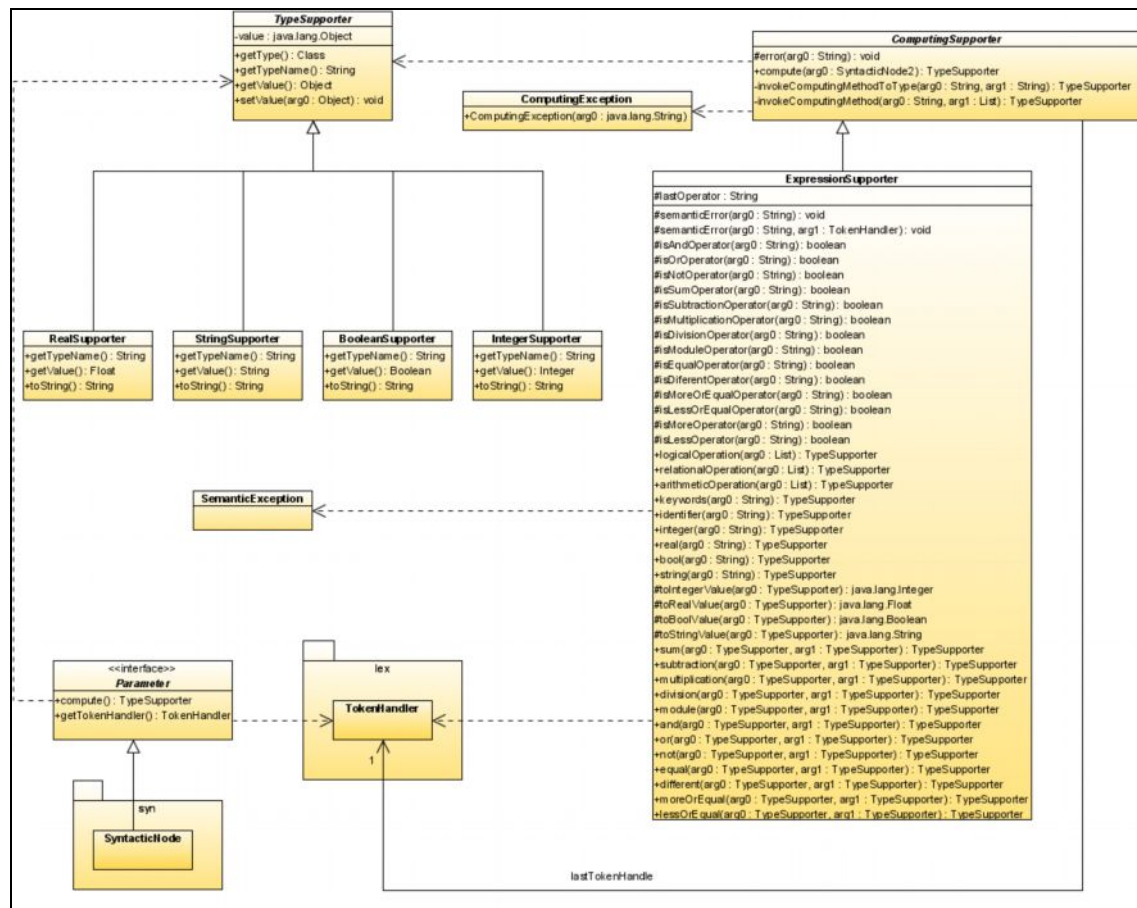
Figura 2. Descritor da linguagem

### 3.5. API de Suporte a Computação:

Disponibiliza para o usuário um conjunto de classes e interfaces (Figura 3) para dar suporte na implementação da análise semântica e geração do resultado final da análise. As principais classes são as classes abstratas *TypeSupporter* e *ComputingSupporter* que são as classes bases para a implementação de novos tipos de dados e classes de suporte a computação respectivamente.

Na API existem alguns tipos pré-definidos como o *RealSupporter* para números reais, *StringSupporter* para string, *BooleanSupporter* para tipos lógicos e *IntegerSupporter* para os tipos inteiros.

Também existe uma classe pré-definida de suporte a computação de nome *ExpressionSupporter* que pode ser usada para implementar expressões aritméticas.



\*Figura 3. API de Suporte a Computação

### 4. Exemplo de utilização do framework Analyzer

Para demonstrar o potencial do Analyzer, será implementado um pequeno exemplo de um interpretador de expressões aritméticas. Outros exemplos mais complexos, como um analisador de sentenças SQL e um interpretador de uma pequena linguagem de programação, estão disponíveis no site <http://sourceforge.net/projects/genericanalyzer>.

\* A Figura 3 encontra-se em dimensões maiores no Anexo I para facilitar a visualização.

O exemplo demonstra um programa que lê uma expressão aritmética, como por exemplo:  $23 + ((23 - 3) * 2)$ , informada pelo usuário na console e devolve o resultado do cálculo na mesma console.

#### 4.1. Preparando o ambiente

Para o nosso exemplo utilizamos a IDE de desenvolvimento *NetBeans 6.0* disponível em <http://www.netbeans.org>, porém pode ser utilizada qualquer outra IDE que suporta Java.

Vamos precisar da biblioteca do *Analyzer* que está disponível no site <http://sourceforge.net/projects/genericanalyzer> na versão 1.0. Depois de baixado o arquivo *Analyzer1.0.jar*, ele precisa ser adicionado no path da aplicação.

Veja a figura 4, como deve ficar a estrutura de diretórios do projeto no *NetBeans*. Observe que o nome do projeto se chama *Expressao*.

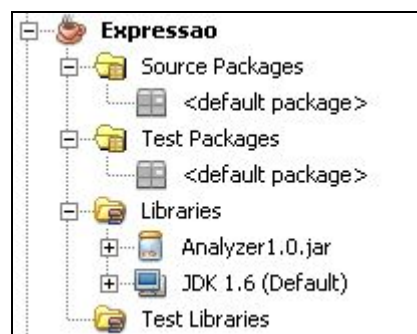


Figura 4. Estrutura de diretório do exemplo

#### 4.2. Configurando o Descritor da Linguagem.

No projeto é criado um arquivo XML com o nome *def-expr.xml* que será o Descritor da Linguagem. Em seguida é adicionado um novo diretório de nome *resources*. Os nomes do arquivo e do diretório são opcionais.

No Descritor da Linguagem são definidos os tokens que serão basicamente números inteiros, números reais, delimitador de grupo e os operadores aritméticos para soma, adição, subtração e divisão. A Gramática Livre de Contexto (GLC) será a seguinte:

$\langle \text{EXPR} \rangle \Rightarrow \langle \text{EXPR} \rangle + \langle \text{TERM} \rangle \mid \langle \text{EXPR} \rangle - \langle \text{TERM} \rangle \mid \langle \text{TERM} \rangle$

$\langle \text{TERM} \rangle \Rightarrow \langle \text{TERM} \rangle * \langle \text{SIGN-FACT} \rangle \mid \langle \text{TERM} \rangle / \langle \text{SIGN-FACT} \rangle \mid \langle \text{SIGN-FACT} \rangle$

$\langle \text{SIGN-FACT} \rangle \Rightarrow + \langle \text{FACTOR} \rangle \mid - \langle \text{FACTOR} \rangle \mid \langle \text{FACTOR} \rangle$

$\langle \text{FACTOR} \rangle \Rightarrow ( \langle \text{EXPR} \rangle ) \mid \text{real} \mid \text{integer}$

A figura 5 demonstra como deve ficar parte do descritor da linguagem. O restante da GLC segue o mesmo padrão. O código completo desse exemplo está disponível no site da aplicação citado anteriormente.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE people_list SYSTEM "language-definition.dtd">
<analyzer>
  <language-name>Interpretador de Expressao</language-name>
  <language-version>1.0</language-version>
  <computing-class>com.rim.analyzer.comp.ExpressionSupporter</computing-class>
  <lexical>
    <tokens>
      <real/>
      <integer/>
      <arithmetic-operator/>
      <delimiter>
        <pattern>\{|\}|\s</pattern>
      </delimiter>
    </tokens>
  </lexical>
  <syntactic>
    <context-free-grammar>
      <not-terminal name="EXPR" method="arithmeticOperation">
        <productions>
          <production>
            <not-terminal name="EXPR"></not-terminal>
            <terminal name="arithmetic-operator">+</terminal>
            <not-terminal name="TERM"></not-terminal>
          </production>
          <production>
            <not-terminal name="EXPR"></not-terminal>
            <terminal name="arithmetic-operator">-</terminal>
            <not-terminal name="TERM"></not-terminal>
          </production>
          <production>
            <not-terminal name="TERM"></not-terminal>
          </production>
        </productions>
      </not-terminal>
      :
    </context-free-grammar>
  </syntactic>
</analyzer>

```

Figura 5. Descritor da Linguagem do exemplo

### 4.3. Implementação da classe *Expressao*

O próximo passo é criar a única classe do exemplo: *Expressao*. Trata-se de uma classe bastante simples. Possui apenas o método *main*. Suas atribuições são: receber a expressão digitada pelo usuário na console e instanciar a classe *Analyzer* passando para ela o arquivo Descritor da Linguagem; executar a análise léxica na expressão através do método *scanner*; executar a análise sintática através do método *parse* e finalmente, calcular o resultado da expressão com a chamada ao método *compute*.

As figura 6 e 7 demonstram o diagrama de classe do *Analyzer* e *AnalyzerFactory* e o código fonte da classe *Expressao* sucessivamente.

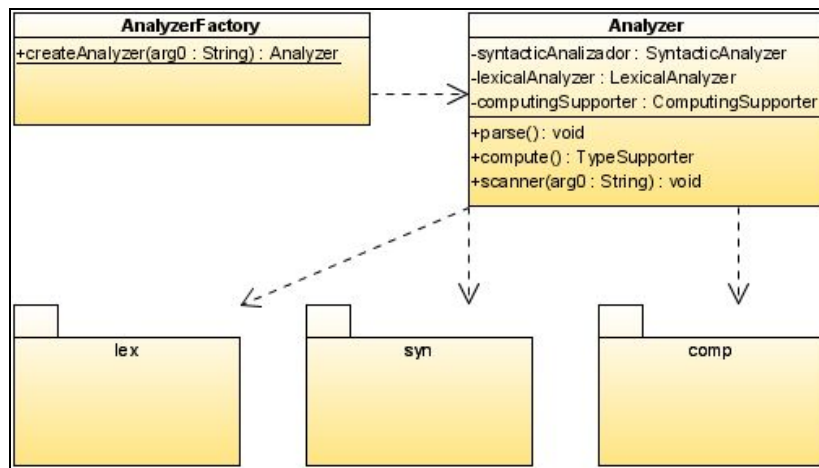


Figura 6. Diagrama de classe do *Analyzer* e *AnalyzerFactory*

```

package exemplo;

import com.rim.analyzer.Analyzer;
import com.rim.analyzer.AnalyzerFactory;
import com.rim.analyzer.comp.TypeSupporter;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Expressao {

    public static void main(String[] args){
        System.out.println("Entre com uma expressão aritmética: ");
        try {
            BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
            Analyzer analyzer = AnalyzerFactory.createAnalyzer("src/resources/def-expr.xml");
            analyzer.scanner(new StringBuffer(in.readLine()));
            analyzer.parse();
            TypeSupporter resultado = analyzer.compute();
            System.out.println("O resultado da expressão é " + resultado.toString()+"\n");
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
  
```

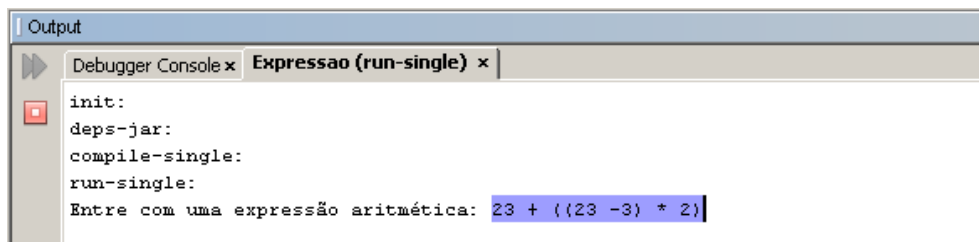
Figura 7. Implementação da classe *Expressao*

#### 4.4. Executando o exemplo

Feita a configuração do Descritor da Linguagem e a implementação da classe *Expressão*, o *Analyzer* está pronto para ser usado no exemplo proposto, cabendo ao usuário passar uma expressão qualquer para ser calculada.

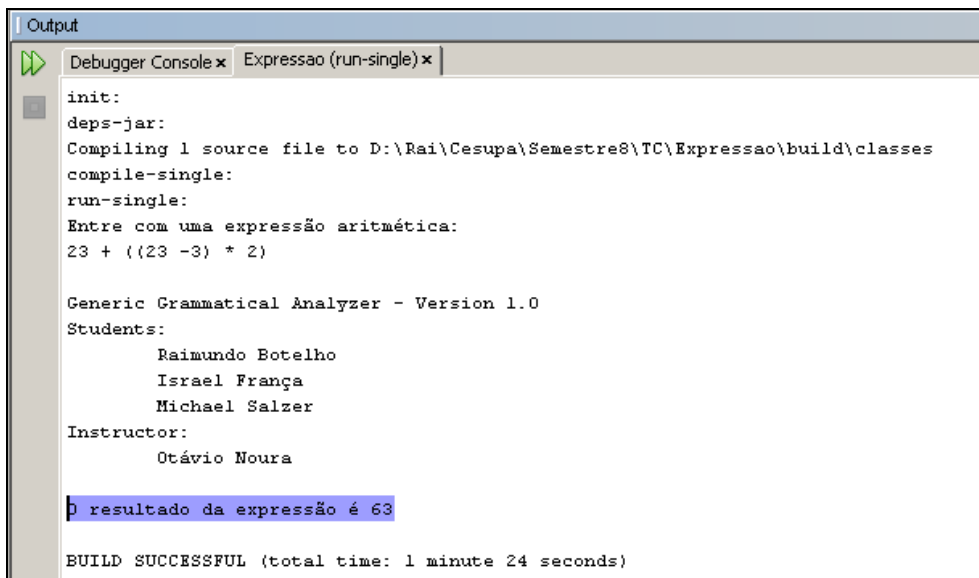
As figuras 8 e 9 demonstram a execução do interpretador de expressão aritmética. Na primeira o usuário informa os dados (expressão), executa o programa e recebe a resposta logo em seguida na mesma console (figura 9).

Aqui vale ressaltar que se um terceiro usuário, não envolvido na configuração do *Analyzer*, viesse a fazer uso do mesmo para calcular outra expressão, o processo seria transparente a esse usuário. Transparência ao usuário é outra característica da ferramenta.



```
Output
Debugger Console x Expressao (run-single) x
init:
deps-jar:
compile-single:
run-single:
Entre com uma expressão aritmética: 23 + ((23 - 3) * 2)
```

**Figura 8. Entrada da expressão para o cálculo**



```
Output
Debugger Console x Expressao (run-single) x
init:
deps-jar:
Compiling 1 source file to D:\Rai\Cesupa\Semestre8\TC\Expressao\build\classes
compile-single:
run-single:
Entre com uma expressão aritmética:
23 + ((23 - 3) * 2)

Generic Grammatical Analyzer - Version 1.0
Students:
    Raimundo Botelho
    Israel França
    Michael Salzer
Instructor:
    Otávio Noura

resultado da expressão é 63

BUILD SUCCESSFUL (total time: 1 minute 24 seconds)
```

**Figura 9. Resultado do cálculo da expressão**

## 5. Conclusão

O framework Analyser foi idealizado para oferecer suporte aos usuários em diversas áreas da informática, especificamente nas áreas de mineração de dados, algoritmos, compiladores e interpretadores. Por ser uma ferramenta de fácil utilização que permite construir analisadores gramaticais com muita rapidez sem a necessidade da implementação de todas as etapas complexas de um tradutor, ela agrega grande potencial ao dia-a-dia dos profissionais de informática envolvidos em ambiente de programação. Assim como no meio acadêmico para ensino e aprendizado aos estudantes do curso de Ciência da Computação nas disciplinas de Algoritmos e Compiladores, visto que a ferramenta envolve todos os conceitos inerentes a essas disciplinas e os demonstra na prática.



## **6. Referências**

Aho, A., Sethi, R. e Ullman, J. D. (1995.) – Compiladores: Princípios, Técnicas e Ferramentas, LTC.

Arnold, Ken, Gosling, James (2007) - A Linguagem De Programação Java, 4.Ed. – Bookman.

Deitel, Harvey M., Deitel, Paul J., Ramon Nieto, Et Al (2003) - XML: Como Programar – Bookman.

Delamaro, Márcio Eduardo (2004) – Como Construir um Compilador Utilizando Ferramentas Java – Novatec.

Furgeri, Sérgio (2001) - Ensino Didático da Linguagem XML – Érica.

Grune, D., Bal, H. E., Jacobs, C. J. H. e Langendoen, K. G. (2001) – Projeto Moderno de Compiladores: Implementação e Aplicações – Editora Campus.

Jargas, Aurélio Marinho (2006) - Expressões Regulares - Uma Abordagem Divertida – Novatec

Louden, Kenneth C. (2004) – Compiladores: Princípios e Práctica – Thomson

Menezes, Paulo Fernando Blauth (2005) - Linguagens Formais e Autômatos - Sagra Luzzatto.

Price, Ana Maria de Alencar, Toscani, Simão Sirineo (2000) - Implementação de Linguagens de Programação: Compiladores – Sagra Luzzatto.

Stubblebine, Tony (2007) - Guia de Bolso Expressões Regulares – 2. Ed. – Alta Books.

## Anexo I

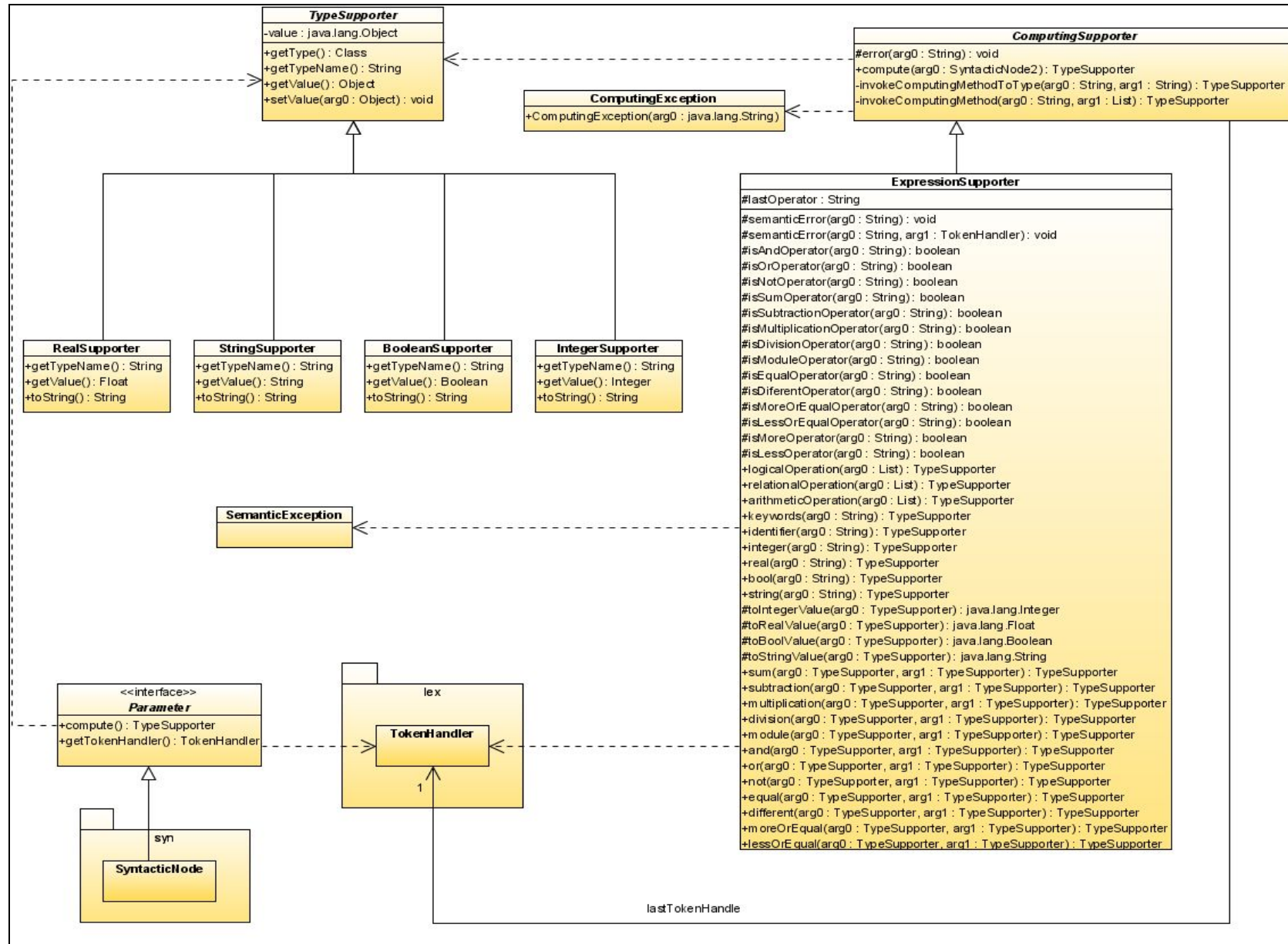


Figura 3. API de Suporte a Computação