

# Gerador de Parsers LR

César Menin de Mello

Orientador: Gilberto Fernandes Marchioro

Universidade Luterana do Brasil (ULBRA) – Curso de Ciência da Computação

cmello@live.com

***Resumo.** Este artigo apresenta uma ferramenta com ambiente gráfico para o desenvolvimento automatizado de parsers. A ferramenta abrange desde a especificação, validação e teste de gramáticas livres de contexto até a geração de código e depuração de um parser LR.*

## 1 Introdução

*Parser* ou *analisador sintático* é um componente de software que interpreta uma entrada detectando e agrupando construções hierárquicas pré-definidas. A entrada pode ser um conjunto de palavras e símbolos que formam um texto em uma linguagem formal, como por exemplo o código-fonte de um programa. Também pode ser um bloco de estruturas binárias, como as mensagens de um protocolo de comunicação. *Parsers* são utilizados em diversas aplicações, como por exemplo:

- Compiladores e interpretadores de linguagens de programação;
- Processamento de SQL, HTML, XML, CSS, XAML, ODF, OOXML e scripts em geral nos mais diversos softwares que usam estes formatos;
- Processamento de protocolos de comunicação como HTTP, SNMP, SOAP, OPC-UA, web services;
- Interpretadores de arquivos de configuração e parâmetros textuais;
- Analisadores de expressões.

Ao longo das últimas décadas foram desenvolvidas técnicas que permitem a construção mecânica de *parsers* a partir de descrições formais da linguagem de entrada.

O formato do texto ou linguagem de entrada é especificado formalmente através de uma *gramática*. A gramática é uma especificação de como as palavras e símbolos se agrupam hierarquicamente para formar as diversas construções da linguagem.

A primeira funcionalidade que esta ferramenta fornece ao usuário é uma interface gráfica de apoio na especificação de gramáticas. Elas podem ser editadas, validadas e testadas num mesmo ambiente integrado, sem necessidade de programação. Isto possibilita mais foco e produtividade no projeto da gramática, livrando o usuário de tarefas secundárias nesta etapa do projeto. O desenvolvimento de uma gramática sem erros contribui para uma grande economia de tempo e esforço nas etapas posteriores do projeto do *parser*.

As validações executadas pela ferramenta garantem que a gramática não contém ambigüidades e está adequada para a construção de um *parser*. A partir desta gramática a ferramenta então gera mecanicamente um *parser LR*.

A letra “L” da sigla *LR* significa *left to right scanning* e a “R” significa *rightmost derivation*. Um *parser LR* analisa a entrada da esquerda para a direita, identificando as construções mais básicas da gramática e a partir destas as de nível mais alto. Por este motivo este tipo de análise sintática também é conhecida como *bottom-up*. A técnica foi criada por KNUTH (1965).

Depois da gramática ser validada, a ferramenta gera o *parser* e permite testá-lo interativamente, sem necessidade de compilação externa. Recursos como execução passo-a-passo, monitoração das ações e da pilha estão presentes dentro do mesmo ambiente integrado.

Na seção 2 serão apresentados os aspectos teóricos e algoritmos existentes nos quais a ferramenta se baseia; na seção 3, serão apresentados os detalhes da implementação e um exemplo de uso; a seção 4 avalia os resultados obtidos; a seção 5 conclui o artigo citando também as limitações e funcionalidades que não foram implementadas; por fim os agradecimentos e referências bibliográficas são o conteúdo das seções 6 e 7 respectivamente.

## 2 Fundamentação Teórica

O desenvolvimento de *parsers* é uma das áreas mais maduras da Ciência da Computação. Diversas técnicas e algoritmos foram desenvolvidos e aperfeiçoados ao longo das últimas décadas, possibilitando a construção de analisadores sintáticos eficientes e precisos.

O processo de interpretação de um texto costuma ser dividido em três etapas:

- *Análise Léxica*: lê os caracteres do texto e os agrupa em palavras ou símbolos com significado comum chamados *tokens*;
- *Análise Sintática*: recebe a seqüência de *tokens* resultantes da análise léxica e identifica como eles se agrupam para formar as construções hierárquicas da linguagem. É comum gerar como saída uma árvore com as construções hierárquicas identificadas;
- *Análise Semântica*: percorre a estrutura hierárquica proveniente da análise sintática e captura características de alguns nós para adicionar significado a outros. Exemplo: identificação e validação de tipos de variáveis em linguagens de programação.

O foco deste artigo e da respectiva ferramenta é a geração do analisador sintático. Porém como na prática para utilizar o analisador sintático também é necessário um analisador léxico, a ferramenta fornece suporte básico para a especificação, teste e geração do código do mesmo.

A análise semântica é implementada pelo usuário na forma de ações customizadas que o analisador sintático chama conforme reconhece as estruturas

definidas na gramática. Detalhes sobre a implementação destas ações customizadas serão abordados na seção 3.

## 2.1 Análise Léxica

O analisador léxico lê os caracteres do texto e os agrupa em palavras ou símbolos especiais chamados *tokens*. Cada *token* representa uma categoria de palavras ou símbolos com significado comum. Um exemplo de entrada é mostrado na figura 1.

*if (altura > margem + 10.5)*

**Figura 1: exemplo de entrada para um analisador léxico**

Ao ler os caracteres desta expressão, um analisador léxico poderia gerar a saída mostrada na Tabela 1.

**Tabela 1: Exemplo de saída de um analisador léxico**

Token	Lexema
IF	if
ABRE_PARENTESE	(
IDENTIFICADOR	altura
MAIOR_QUE	>
IDENTIFICADOR	margem
OP_SOMA	+
NUMERO	10.5
FECHA_PARENTESE	)

Neste exemplo, a seqüência de *tokens* IF, ABRE\_PARENTESE, IDENTIFICADOR, MAIOR\_QUE, IDENTIFICADOR, OP\_SOMA, NUMERO e FECHA\_PARENTESE é a entrada que será fornecida ao analisador sintático.

O texto que compõe cada *token* isoladamente é chamado de *lexema*. No exemplo, o *token* IDENTIFICADOR aparece duas vezes, mas com lexemas diferentes (“altura” e “margem”).

O *token* do tipo IDENTIFICADOR é bastante comum em linguagens de programação, representando nomes definidos pelo usuário (exemplos: nomes de variáveis, funções, tipos, etc.).

Padrões de reconhecimento de *tokens* podem ser especificados através de *expressões regulares*, uma notação inicialmente abordada por KLEENE (1956).

A implementação do analisador léxico é simples e pode ser feita tanto manualmente como através de geradores de código a partir de expressões regulares.

AHO ET AL. (1986, p. 83) descreve o funcionamento, implementação e geração automática analisadores léxicos. LESK (1975) criou um gerador de analisadores léxicos muito conhecido chamado *Lex*.

A comunicação entre o analisador léxico e o analisador sintático normalmente é feita através de uma chamada de função que retorna o próximo *token* e opcionalmente os valores de seus atributos ou o lexema. O analisador sintático chama esta função do analisador léxico sempre que precisa mais um *token* para efetuar a análise.

## 2.2 Análise Sintática

A função do analisador sintático é receber a seqüência de *tokens* e identificar como eles se agrupam para formar construções hierárquicas de sintaxe. Além de identificar estas construções, o analisador sintático tem a função de detectar e informar eventuais violações de gramática encontradas no texto.

Antes de descrever as técnicas de análise sintática convém abordar uma introdução às gramáticas e seu uso.

### 2.2.1 Gramáticas

Pode-se definir a gramática de uma linguagem através dos seguintes componentes:

1. *Conjunto de símbolos Terminais*: este é o conjunto de todos os *tokens* que podem ser utilizados no texto de entrada;
2. *Conjunto de Não-Terminais*: não-terminais são abstrações sintáticas que definem como os *tokens* se agrupam para dar significado às diversas construções da linguagem. Estas abstrações podem ser recursivas, ou seja, um não-terminal pode ser composto tanto de terminais como não-terminais; exemplos de não-terminais na gramática da Língua Portuguesa são o sujeito, predicado, oração, período, etc. Exemplos na linguagem de programação C são declaração de variável, bloco de código, instrução, expressão, etc.
3. *Conjunto de Produções*: são as regras que definem a composição de cada não-terminal da linguagem;
4. *Símbolo Inicial*: é um não-terminal que representa todas as entradas válidas na linguagem.

Como exemplo pode ser considerada uma gramática para expressões aritméticas envolvendo apenas somas de números. Pode ser definido um terminal para representar os números, e outro para representar o operador de soma. Estes terminais podem ser nomeados **NUMERO** e **OPERADOR\_SOMA**.

Apenas um não-terminal precisa ser definido, e ele pode ser nomeado *expressao*. Esta gramática teria apenas duas produções, conforme mostra a figura 2.

```
expressao → NUMERO  
expressao → expressao OPERADOR_SOMA expressao
```

**Figura 2: Exemplo simples de produções de uma gramática**

Esta forma de descrever as produções é uma simplificação da representação BNF (*Backus-Naur Form*). A BNF tem origem no trabalho de Naur na linguagem Algol 60 (NAUR, 1963), sendo proposta por KNUTH (1964).

Nesta notação cada linha representa uma produção. No lado esquerdo consta o não-terminal que está sendo definido, e no lado direito a lista de terminais e/ou não-terminais que o compõem. O mesmo não-terminal pode aparecer no lado esquerdo de mais de uma produção, o que significa que este não-terminal pode assumir qualquer uma das alternativas. Para facilitar a leitura, os terminais estão destacados com letras maiúsculas (**NUMERO**, **OPERADOR\_SOMA**), e os não-terminais com letras minúsculas.

A primeira produção significa que o não-terminal *expressao* pode ser composto por um único token **NUMERO**. Ou seja, a presença de um *token NUMERO* sozinho já caracteriza uma expressão válida nesta gramática. A segunda produção diz que a expressão também pode ser a seqüência de outra expressão, seguida por um operador de soma e uma terceira expressão.

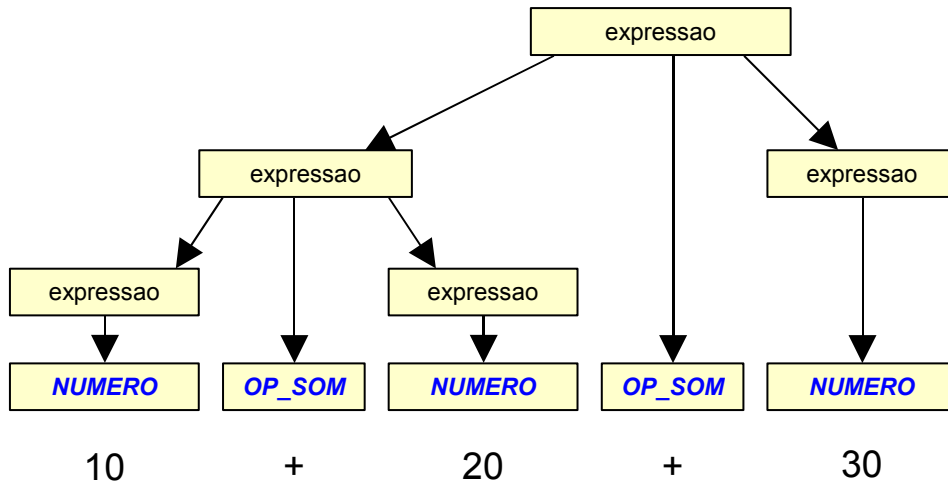
Como esta gramática é composta por um único não-terminal, este é o símbolo inicial. Isso significa que a entrada válida para a linguagem definida nesta gramática é uma expressão.

### 2.2.2 Derivação

Uma entrada válida pode ser descrita como uma seqüência de substituições de não-terminais pelos componentes da sua definição, começando pelo símbolo inicial e concluindo na seqüência de terminais. Segue um exemplo de entrada válida (figura 3) e sua respectiva seqüência de derivações na forma de árvore (figura 4).

```
Entrada: 10 + 20 + 30
```

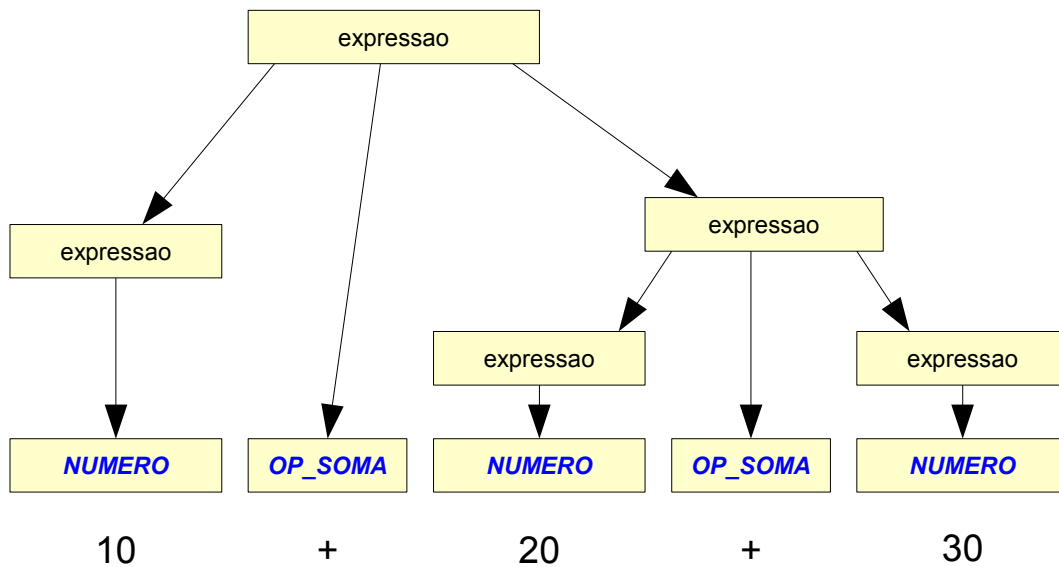
**Figura 3: Exemplo de expressão simples**



**Figura 4: Árvore de análise sintática para a expressão da figura 3**

A raiz da árvore é o símbolo inicial e representa a entrada inteira. Cada nodo com filhos é um não-terminal, cujos filhos são os seus componentes na respectiva produção. A árvore termina nas folhas, que correspondem aos terminais (*tokens*).

Nesta gramática a mesma expressão também pode ser representada por outra árvore, conforme mostra a figura 5.



**Figura 5: Uma segunda árvore sintática possível para a expressão 10+20+30**

Quando a mesma entrada pode dar origem a derivações diferentes, a gramática é considerada *ambígua*. Gramáticas ambíguas não servem para o algoritmo de análise sintática abordado neste trabalho, e devem ser modificadas para que a ambigüidade seja resolvida.

Neste exemplo, para remover a ambigüidade é necessário especificar a *associatividade* do operador soma. Como nas expressões aritméticas este operador é associado à esquerda, modificamos a gramática para permitir a recursão somente à esquerda (figura 6).

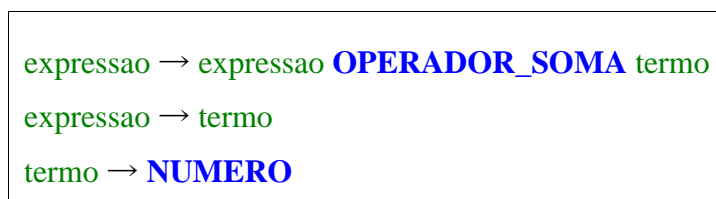


Figura 6: Produções expressando associatividade à esquerda

Desta forma obtém-se uma árvore de derivação única para a expressão 10 + 20 + 30, ilustrada na figura 7.

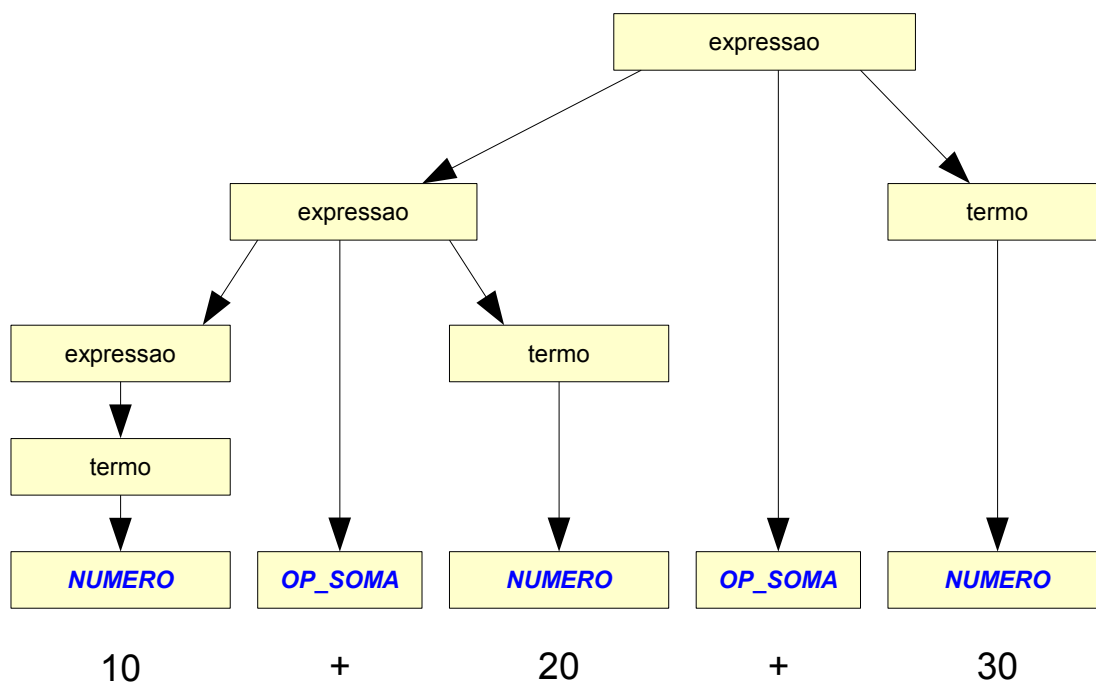


Figura 7: Árvore sintática utilizando as produções da figura 6

### 2.2.3 Técnicas de Análise Sintática

As técnicas de análise sintática diferem basicamente na maneira como identificam as estruturas hierárquicas definidas na gramática, e podem ser classificadas em dois grupos:

- *Top-down*: começam a identificação pela regra mais geral da gramática até chegar nas mais específicas. Conseqüentemente constroem uma árvore sintática de cima (símbolo inicial) para baixo (terminais);

- *Bottom-up*: identificam as regras mais específicas e com base nestas vai identificando regras mais gerais até chegar no símbolo inicial. Consequentemente a árvore sintática é construída de baixo para cima.

Ambas as técnicas permitem a construção de um *parser* mecanicamente a partir da gramática. A técnica *top-down* é mais adequada para a construção do *parser* manualmente, enquanto a técnica *bottom-up* é mais complicada e só se torna viável com a geração de código.

A técnica escolhida para a geração do *parser* foi a *bottom-up* com algoritmo LR, compartilhando muitas semelhanças com a ferramenta *YACC / Bison*.

AHO ET AL. (1986) cita que “*parsers LR podem ser construídos para reconhecer virtualmente todas as construções de linguagens de programação para as quais é possível escrever gramáticas livres de contexto*”. Outra vantagem citada para a técnica é que “*um parser LR consegue detectar erros de sintaxe tão logo quanto é possível analisando a entrada da esquerda para a direita*” (AHO ET AL., 1986, p. 215).

## 2.2.4 Algoritmo genérico de análise LR

O *parser* LR é composto de uma pilha de estados e de duas tabelas, chamadas *action* e *goto*. Estas tabelas são geradas mecanicamente a partir da gramática, conforme será explicado a seguir.

A tabela *action* é indexada por um número de estado e por um *token*. Suas linhas correspondem aos estados e suas colunas aos *tokens* da linguagem. Cada entrada desta tabela deve conter uma destas três ações: empilha (*shift*), reduz (*reduce*) ou aceita (*accept*); se a entrada não possuir nenhuma ação, indica um erro de sintaxe. Os *parsers* LR também são conhecidos como *parsers “shift-reduce”* devido a estas ações serem a base do seu funcionamento.

A tabela *goto* é indexada por um número de estado e por um não-terminal. Suas linhas correspondem aos estados e suas colunas aos não-terminais da linguagem. As entradas desta tabela indicam as substituições a serem realizadas nas reduções.

O algoritmo genérico do *parser* é resumido na figura 8 (baseado em AHO ET AL, 1986, p. 219).

As ações presentes nas entradas da tabela *action* são:

- *Empilha s'*: (*shift s'*): informa que um estado deve ser empilhado;
- *Reduzir pela produção  $A \rightarrow w$* : informa que deve ser feita uma redução utilizando a produção especificada; isto significa que os símbolos dos estados empilhados correspondem ao lado direito da produção, e portanto devem ser substituídos pelo estado correspondente ao lado esquerdo da produção;
- *Aceitar*: esta ação informa que a análise foi concluída com sucesso (foi feita a redução do símbolo inicial);
- *Entrada em branco*: indica que ocorreu um erro de sintaxe.

1. Inserir na pilha o estado inicial (0);
2. Repetir
  - 2.1. Sendo  $s$  o estado no topo da pilha e  $a$  o próximo *token* da entrada:
  - 2.2. Se  $action[s, a] = empilha\ s'$  então
    - 2.2.1. insira  $s'$  no topo da pilha
    - 2.2.2. busque o próximo *token*
  - 2.3. caso contrário se  $action[s, a] = reduzir\ pela\ produção\ A \rightarrow w$  então:
    - 2.3.1. Remova  $|w|$  estados do topo da pilha;
    - 2.3.2. Sendo  $s'$  o estado no topo da pilha, empilhe  $goto[s', A]$  no topo da pilha;
    - 2.3.3. Sinalize o reconhecimento da produção  $A \rightarrow w$ ;
  - 2.4. caso contrário se  $action[s, a] = aceitar$ , termine com sucesso;
  - 2.5. caso contrário termine com erro.

**Figura 8: Algoritmo genérico do *parser* LR**

### 2.2.5 Construção das tabelas *action* e *goto*

Existem diversos algoritmos para geração das tabelas mecanicamente a partir da gramática da linguagem. O algoritmo descrito a seguir é o “LR Canônico” (baseado em AHO ET AL, 1986, p. 230).

As seguintes definições se aplicam à notação utilizada para descrever o algoritmo:

- *Item da gramática*: representa uma configuração possível do *parser* durante a leitura da entrada. É utilizado um ponto para indicar a posição dentro da produção. Exemplo: para representar o item cuja produção é  $A \rightarrow sXE$  e indicar o estado em que o *parser* leu o “s” e ainda pode ter pela frente a sequência “XE”, utiliza-se a representação ilustrada na figura 9.

$$A \rightarrow s.XE$$

**Figura 9: Representação de um item da gramática**

- *Símbolo Inicial*: a convenção é utilizar a letra S para representar o símbolo inicial;
- *Fim do texto de entrada*: o final do texto de entrada é simbolizado com um símbolo de cifrão (\$).

O algoritmo utiliza as funções *FIRST*, *FOLLOW*, *CLOSURE* e *GOTO* definidas a seguir.

- *Função FIRST(w)*: retorna o conjunto de *tokens* que podem iniciar derivações de  $w$ ;
- *Função FOLLOW(N)*: retorna o conjunto de *tokens* que podem aparecer imediatamente após qualquer derivação de  $N$ .

- Função CLOSURE(I): o algoritmo para esta função é ilustrado na figura 10 (AHO ET AL., 1986, p. 232).

```

repita
    para cada item  $[A \rightarrow s.BE, a]$  em I,
    cada produção  $B \rightarrow y$  em  $G'$ ,
    e cada terminal  $b$  em  $FIRST(Ea)$ 
    tal que  $[B \rightarrow .y, b]$  não está em I
        adicione  $[B \rightarrow .y, b]$  ao conjunto I;
até que mais nenhum item possa ser adicionado ao conjunto I;
retorne I

```

**Figura 10: Algoritmo da função Closure**

- Função GOTO(I, X): o algoritmo para esta função é ilustrado na figura 11.

```

seja J o conjunto de itens  $[A \rightarrow sX.E, a]$  tal que
     $[A \rightarrow s.XE, a]$  está em I;
retorne CLOSURE(J);

```

**Figura 11: Algoritmo da função Goto**

O algoritmo para a construção do conjunto de itens da gramática é ilustrado na figura 12.

```

C := { CLOSURE({[S' → .S, $]})};
repita
    para cada conjunto de itens I em C e cada símbolo X da gramática
        tal que GOTO(I, X) não seja vazio e não esteja em C
            adicione GOTO(I, X) a C;
até que mais nenhum conjunto de itens possa ser adicionado em C.

```

**Figura 12: Algoritmo para construção do conjunto de itens da gramática**

### 2.2.5.1. Construção da tabela Action

A tabela *action* é construída com base no conjunto de itens retornados pelo algoritmo apresentado na figura 12, seguindo as regras a seguir.

O estado  $i$  é construído do item  $I_i$ . As ações para este estado são determinadas da seguinte forma:

a) Se  $[A \rightarrow s.aE]$  está em  $I_i$  e  $GOTO(I_i, a) = I_j$ , então a ação para  $action[i, a]$  deve ser “empilhe  $j$ ”.  $a$  deve ser um terminal.

b) Se  $[A \rightarrow s.]$  está em  $I_i$ , então a ação para  $action[i, a]$  deve ser “reduza  $A \rightarrow s$ ” para todo  $a$  em  $FOLLOW(A)$ ;  $A$  não pode ser  $S'$ .

c) Se  $[S' \rightarrow S.]$  está em  $I_i$ , então a ação para  $action[i, \$]$  deve ser “accept”.

Se alguma ação for gerada duas vezes pelo algoritmo acima, é porque foi detectada uma ambigüidade na gramática. O algoritmo não consegue gerar um *parser* neste caso.

### 2.2.5.2. Construção da tabela Goto

A entrada na tabela *goto* para o estado  $i$  contém as transições para todos os não-terminais  $A$  seguindo a regra: Se  $GOTO(I_i, A) = I_j$ , então  $goto[i, A] = j$ .

O estado inicial do *parser* é construído a partir do conjunto de itens contendo  $[S' \rightarrow .S]$ .

## 2.3 Ferramentas semelhantes

O gerador de parsers LR mais conhecido é o *YACC*, cuja implementação *open-source* se chama *Bison*. Trata-se de um excelente gerador de *parsers* que recebe uma gramática com ações anexadas na forma de blocos de código na linguagem C. A saída deste gerador é um programa C que implementa o *parser* da gramática especificada com o código das ações embutido nas reduções. O *YACC* é citado em AHO ET AL. (1986, p. 257).

Apesar do código-fonte do *Bison* ser aberto, não é viável adaptá-lo para a utilização no ambiente interativo proposto. Por isso, decidiu-se implementar o gerador de *parser* do zero, tendo em vista a utilização em depuradores e ambientes de desenvolvimento integrados desde o início. Outro benefício de criar o próprio gerador é a possibilidade de gerar código em outras linguagens de programação.

## 3 Apresentação da Ferramenta e Implementação

Foi implementada uma ferramenta baseada em interface gráfica com as seguintes funcionalidades:

- Editor de definições de terminais e expressões regulares com teste interativo: nesta tela o usuário cria as definições de terminais e ao mesmo tempo pode testar o seu funcionamento em tempo real, sem necessidade de compilação ou utilização de ferramentas externas. Esta tela é mostrada na figura 13.
- Editor de definições dos não-terminais e produções da gramática: Nesta tela é possível definir os não-terminais e produções utilizando interface gráfica simples e intuitiva. As produções são definidas com a simples seleção dos símbolos desejados a partir de listas. Esta tela é apresentada na figura 14.
- Gravação e leitura da definição da gramática em arquivo XML;
- Validação da gramática e exibição de lista de erros integrada;
- Geração e visualização das tabelas Action e Goto;
- Geração do código do *analisador léxico* e do *parser*; a ferramenta gera o fonte de um *parser* em C# que implementa as tabelas geradas para a gramática, assim como um analisador léxico de exemplo que implementa as definições de terminais informadas. Suporte a geração de *parsers* em outras linguagens de programação está previsto na arquitetura da ferramenta.

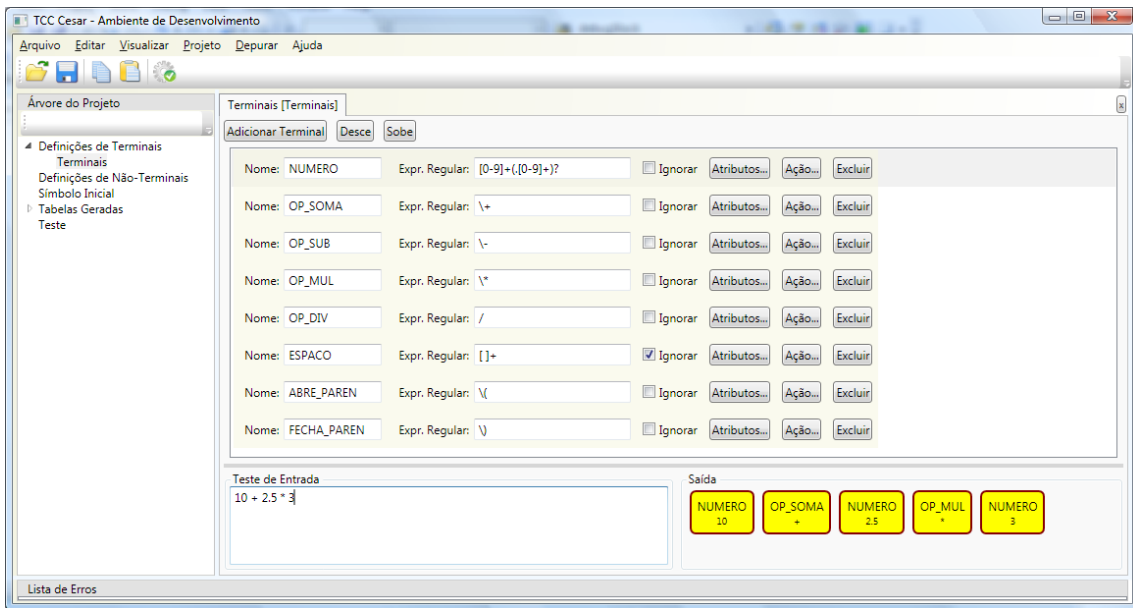


Figura 13: Editor de terminais e expressões regulares com teste interativo

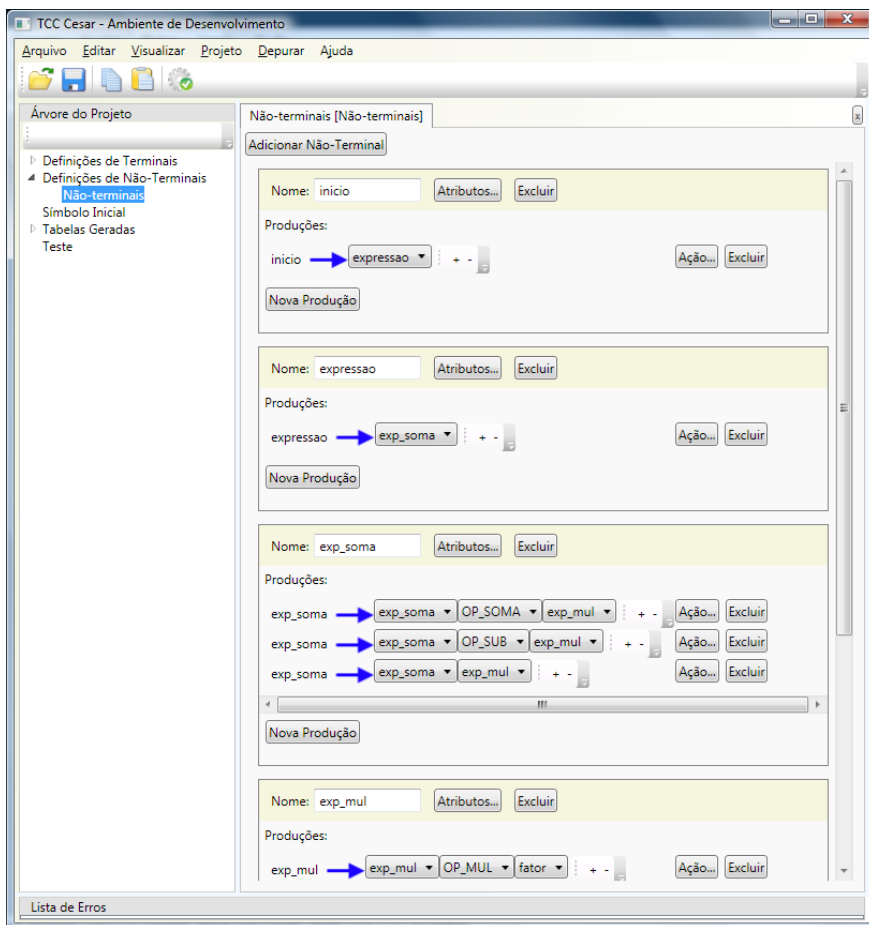


Figura 14: Edição dos não-terminais e produções

- Teste e depuração do *parser*: Além de gerar o código do *parser*, a ferramenta compila e executa o mesmo em *runtime*, utilizando as APIs próprias do .NET Framework para este fim. A tela da figura 17 mostra a execução do *parser* gerado.

A seguir são apresentados os detalhes da implementação.

### 3.1 Implementação

A implementação foi dividida em três módulos:

- *ParserDesignEngine*: implementa os algoritmos para a validação da gramática e geração do analisador sintático, além da persistência da definição da gramática em formato XML;
- *DesignEnvironment*: implementa toda a interface gráfica da ferramenta;
- *Tests*: implementa uma série de testes automáticos que garantem o funcionamento dos algoritmos para os casos de teste conhecidos e detectam problemas que possam ser introduzidos no desenvolvimento de funcionalidades novas.

O desenvolvimento foi feito utilizando a ferramenta Visual C# 2008 Express (MICROSOFT, 2008a), a linguagem C# e o .NET Framework 3.5 SP1. A interface gráfica foi desenvolvida utilizando o Windows Presentation Foundation (WPF). O sistema de controle de versão utilizado foi o Subversion via TortoiseSVN (TORTOISESVN, 2008).

Os algoritmos para construção das tabelas Action e Goto apresentados na seção 2 encontram-se implementados na classe *Grammar* do módulo *ParserDesignEngine* (métodos *First*, *Follow*, *Closure*, *Goto*, *BuildCanonicalSets* e *BuildParsingTable*). O gerador de código C# do *parser* está implementado na classe *CSharpParserGenerator*.

Com o objetivo de diminuir as dependências no código da interface gráfica, o desenvolvimento foi fortemente baseado no padrão *Presentation Model* (FOWLER, 2008). A adaptação para o WPF foi inspirada no *Composite Application Guidance for WPF* (MICROSOFT, 2008b).

### 3.2 Exemplo de uso: gerando um analisador de expressões aritméticas

A seguir será apresentado um breve roteiro de utilização prática da ferramenta, através da geração de um analisador de expressões aritméticas.

O primeiro passo é a especificação dos terminais e suas respectivas expressões regulares. Para isso é necessário clicar com o botão direito na pasta *Definições de Terminais* e selecionar a opção *Novo Grupo de Terminais*, informando um nome para o grupo. A organização em grupos facilita o gerenciamento de gramáticas com muitos itens, permitindo agrupá-los por categorias (ex: palavras-chave, operadores, etc).

Deve-se clicar no botão *Adicionar Terminal* e configurar os terminais criados de acordo com a Tabela 2.

**Tabela 2: Lista de Terminais**

Nome	Expressão Regular
NUMERO	[0-9]+(.[0-9]+)?
OP_SOMA	\+
OP_SUB	\-
OP_MUL	\*
OP_DIV	/
ESPACO	[ ]+
ABRE_PAREN	\(
FECHA_PAREN	\)

A qualquer momento é possível testar as definições de terminais digitando algum texto no campo *Teste de Entrada* na parte inferior esquerda da tela. Ao lado direito serão apresentados os *tokens* emitidos usando as regras informadas.

O *token* ESPACO não é utilizado na análise sintática, portanto é necessário marcar a opção *Ignorar* para que este tipo de *token* não seja gerado na saída do analisador léxico.

Neste momento a configuração deve estar como mostra a figura 13. É aconselhado realizar testes digitando algumas expressões aritméticas no campo Teste de Entrada.

O próximo passo é cadastrar os não-terminais da gramática. Para isso, deve-se clicar com o botão direito na pasta *Definições de Não-terminais* e escolher a opção *Criar novo grupo de Não-terminais*. Esta organização em grupos tem o objetivo de facilitar a manipulação de gramáticas com grande número de produções e não-terminais.

Os não-terminais são criados clicando no botão *Adicionar Não-Terminal*. Para este exemplo devem ser criados cinco não-terminais: *inicio*, *expressao*, *exp\_soma*, *exp\_mul* e *fator*.

Para cada não-terminal é configurada uma lista de produções, que são criadas ao clicar no botão *Nova Produção* do não-terminal desejado. Os símbolos do lado direito da produção são especificados clicando no botão com um sinal de mais (+) e selecionando da lista. Para este exemplo, as produções mostradas na figura 15 devem ser criadas.

```




inicio ⇒ expressao
expressao ⇒ exp_soma
exp_soma ⇒ exp_soma OP_SOMA exp_mul
exp_soma ⇒ exp_soma OP_SUB exp_mul
exp_soma ⇒ exp_mul
exp_mul ⇒ exp_mul OP_MUL fator
exp_mul ⇒ exp_mul OP_DIV fator
exp_mul ⇒ fator
fator ⇒ NUMERO
fator ⇒ ABRE_PAREN expressao FECHA_PAREN

```

**Figura 15: Produções para a gramática de exemplo**

Neste momento a configuração deve estar semelhante à mostrada na figura 14. A especificação da gramática criada até aqui deve ser gravada em disco clicando no botão apropriado da barra de ferramentas, ilustrado na tabela 3.

**Tabela 3: Botões da barra de ferramentas**

Botão	Ação
	Abre um arquivo XML com definição de gramática
	Salva a definição da gramática em um arquivo XML
	Valida a gramática e compila o <i>parser</i>

Após gravar a gramática, esta pode ser validada clicando no botão de validação. Mensagens de erro são mostradas na parte inferior da janela, conforme mostra a figura 16. A mensagem indica que não foi escolhido o símbolo inicial. Para configurá-lo, é necessário clicar duas vezes no item *Símbolo Inicial* da *Árvore do Projeto* e escolher o não-terminal *inicio*.

Quando a validação passar sem nenhum erro, as tabelas *Action* e *Goto* podem ser consultadas clicando duas vezes nos itens correspondentes da *Árvore do Projeto*.

Se o código do *parser* também foi gerado e compilado sem erros, pode ser testado clicando duas vezes no item *Testes*. Ao digitar uma expressão aritmética e clicar no botão *Executar*, o *parser* é executado e suas ações são mostradas no campo *Ações*. Também é possível executar a análise passo a passo visualizando a pilha do *parser*. Para isso deve ser utilizado o comando *Executar passo a passo*, conforme mostra a figura 17.

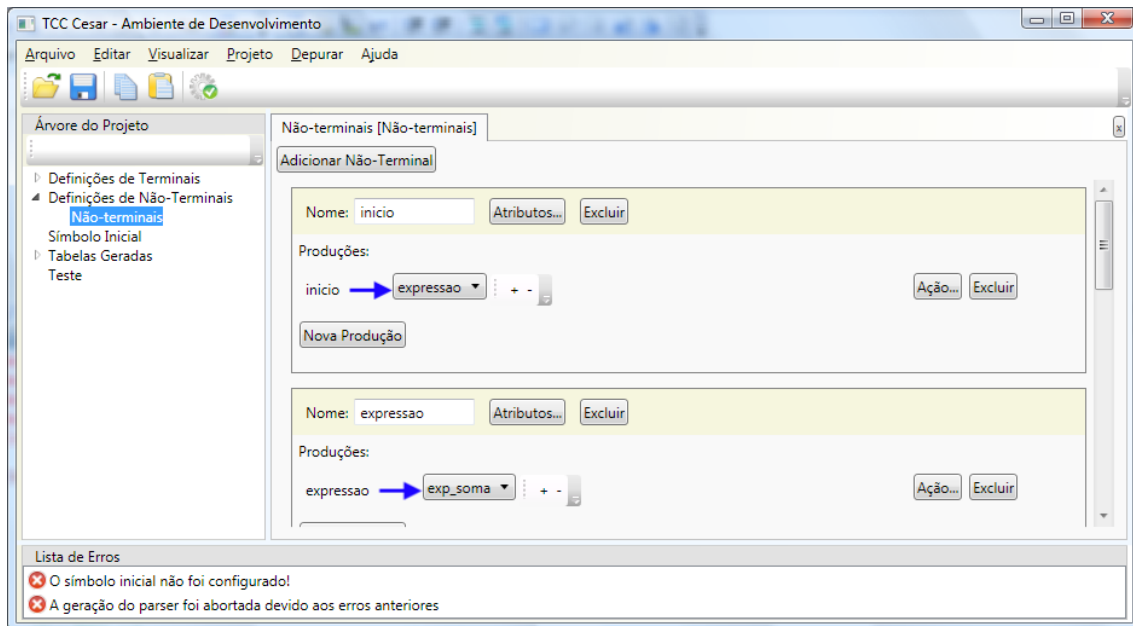


Figura 16: Mensagens de erro

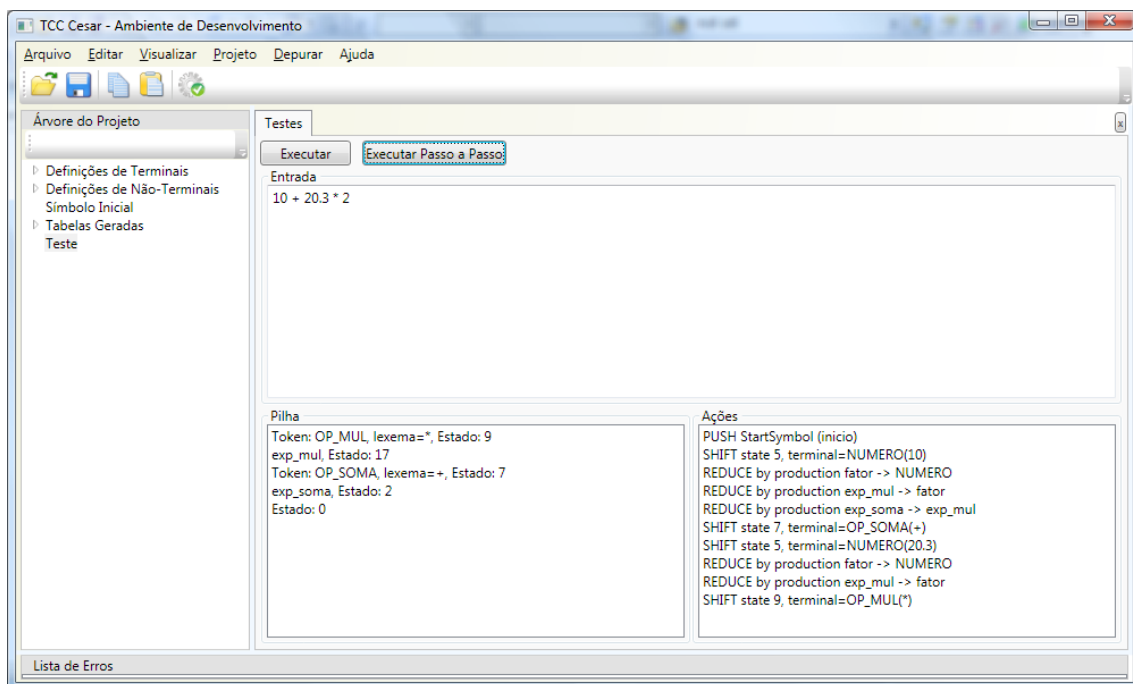


Figura 17: Parser gerado sendo executado na própria ferramenta

O arquivo com o código é gerado no mesmo diretório onde foi gravado o arquivo XML da gramática. Na versão atual da ferramenta a única linguagem suportada é o C# padrão ISO/IEC 23270:2006, compatível com Microsoft Visual C# 2008 e Mono 2.0. Este arquivo tem a extensão `.cs`.

A classe do *parser* é gerada em um *namespace* com nome `GeneratedParser.NomeDoParser`, onde `NomeDoParser` é o mesmo nome dado ao arquivo XML da gramática.

Para utilizar o *parser* gerado, basta fazer um programa com o código mostrado na figura 18.

```
using System;
using GeneratedParser.ExpressoosAritmeticas;
using GeneratedParser.ExpressoosAritmeticas.NonTerminalClasses;

namespace ExemploArtigo
{
    class Program
    {
        static void Main(string[] args)
        {
            Parser parser = new Parser("100 + 2.5 * 2");
            parser.Parse();
        }
    }
}
```

**Figura 18: Programa de exemplo utilizando o *parser* gerado**

A implementação *default* do *parser* simplesmente imprime no terminal uma descrição das ações executadas, semelhante ao que é mostrado quando o *parser* é executado dentro da ferramenta.

Para que este *parser* calcule o valor da expressão aritmética, é necessário customizar as ações. Também é necessário incluir um *atributo* nos não-terminais para guardar o valor numérico.

O código gerado inclui uma classe para cada não-terminal da gramática no namespace *GeneratedParser.NomeDoParser.NonTerminalClasses*. Estas classes podem ser estendidas utilizando o recurso de *partial classes* da linguagem C#. Para adicionar um atributo do tipo *double* às classes dos não-terminais, deve-se incluir no programa o código mostrado na figura 19.

```
namespace GeneratedParser.ExpressoosAritmeticas.NonTerminalClasses
{
    public partial class expressao { public double m_valor; }
    public partial class exp_soma { public double m_valor; }
    public partial class exp_mul { public double m_valor; }
    public partial class fator { public double m_valor; }
}
```

**Figura 19: Adicionando um atributo numérico às classes dos não-terminais**

Por fim, para customizar as ações, é necessário criar uma classe derivada de *DefaultActionHandlers*. Esta classe possui um método para cada produção, que é

chamado quando a respectiva produção for reduzida pelo *parser*. Caso for utilizado o Visual C# 2008 Express para editar o código, basta digitar a palavra *override* dentro da declaração da classe e o IDE irá mostrar a lista de métodos e gerar a sua declaração automaticamente.

Cabe lembrar que na análise *bottom-up* as reduções vão ocorrer a partir das produções mais próximas dos terminais até o símbolo inicial. Portanto, a primeira ação a ser customizada é a redução da produção  $fator \Rightarrow NUMERO$ , que cria um objeto do tipo *fator* e preenche seu atributo *m\_valor* com o valor numérico do lexema do *token* NUMERO. O código fica como mostrado na figura 20.

```
class CustomHandlers : DefaultActionHandlers
{
    public override fator On_fator(Token numero)
    {
        fator f = new fator();
        f.m_valor = Double.Parse(numero.Lexeme);
        return f;
    }
}
```

**Figura 20: Exemplo de código para customização de uma ação de redução**

As demais ações são customizadas de maneira semelhante. Como exemplo a redução que calcula a soma é mostrada na figura 21. Devido ao limite de espaço, o código completo não será fornecido neste artigo e deve ser consultado no material complementar incluído no CD.

```
public override exp_soma On_exp_soma(exp_soma exp_soma1, Token operador, exp_mul exp_mul1)
{
    exp_soma ret = new exp_soma();
    if (operador.TokenType == TokenType.OP_SOMA)
        ret.m_valor = exp_soma1.m_valor + exp_mul1.m_valor;
    else
        ret.m_valor = exp_soma1.m_valor - exp_mul1.m_valor;
    return ret;
}
```

**Figura 21: Exemplo de customização para o cálculo da soma/subtração**

Algumas observações:

- A enumeração *TokenType* e a classe *Token* são geradas automaticamente;
- Produções onde o lado direito é diferenciado apenas pelos tipos de *tokens*, como a do exemplo mostrado acima para a soma/subtração (produções  $exp\_soma \Rightarrow exp\_soma\ OP\_SOMA\ exp\_mul$  e  $exp\_soma \Rightarrow exp\_soma\ OP\_SUB\ exp\_mul$ ) são tratadas

pelo mesmo método, devendo neste caso o código da customização verificar os tipos de *tokens*;

- Para que o *parser* utilize o manipulador de ações customizado, é necessário instanciá-lo com um *constructor* que recebe tanto o objeto que implementa o analisador léxico como o objeto do manipulador customizado, conforme mostra a figura 22.

```
DefaultTokenizer tokenizer = new DefaultTokenizer("100 + 2.5 * 2");  
Parser p = new Parser(tokenizer, new CustomHandlers());
```

Figura 22: Parser instanciado com implementação de ações customizadas

## 4 Avaliação

Todas as gramáticas utilizadas nos testes da ferramenta possibilitaram a geração de código correto.

O algoritmo utilizado na geração das tabelas nesta ferramenta difere do utilizado na maioria dos outros geradores de *parser* LR. A ferramenta proposta utiliza o algoritmo *LR Canônico*, enquanto o *YACC/Bison* utiliza o algoritmo *LALR*. O *LR Canônico* é um algoritmo mais custoso tanto em uso de CPU como memória, e gera tabelas com mais estados que as geradas pelo *LALR* (no máximo o dobro de estados). Porém as tabelas geradas com o *LR Canônico* possibilitam detectar erros de sintaxe com mais precisão. Apesar do tempo de geração das tabelas ser maior com o *LR Canônico*, não é esperada diferença significativa na performance do *parser*.

O analisador léxico gerado pela ferramenta é uma versão inicial simples que não foi projetada com foco na *performance*. Porém o código do analisador sintático gerado permite a utilização de um analisador léxico customizado, bastando para isso passar no *constructor* da classe *Parser* um objeto que implemente a interface *ITokenizer*.

## 5 Conclusão

A ferramenta desenvolvida atende a maior parte dos objetivos planejados, principalmente o de servir como ferramenta didática no ensino de Linguagens Formais e Compiladores. Acredita-se que o aprendizado do projeto de gramáticas é facilitado com o ambiente integrado proposto, ao facilitar os testes e evitar a necessidade de utilização de ferramentas separadas (*Lex + Yacc + compilador C*) para testar a gramática.

Três funcionalidades que de certa forma deixam a desejar e necessitam de mais tempo para desenvolvimento são:

- *Performance do analisador léxico*: conforme introduzido na seção 4, o analisador léxico gerado pela ferramenta tem *performance* insatisfatória. O motivo é a utilização de classes de expressões regulares separadas para cada regra de terminal. A solução é gerar uma única máquina de estados para reconhecimento de todos os *tokens*; esta implementação pode ser feita sem impactos no restante da ferramenta.
- *Suporte a geração de código C++*: devido à facilidade na geração e compilação de código dinamicamente no .NET Framework, e a disponibilidade de classes de

expressões regulares na própria biblioteca de classes, a geração do código em C++ puro foi postergada em relação ao C#. Porém não há nenhuma limitação na ferramenta que impeça o desenvolvimento desta funcionalidade.

- *Clareza das mensagens de ambigüidades na gramática:* Em alguns casos as mensagens apresentadas quando a ferramenta detecta ambigüidades na gramática não são claras e podem ser melhoradas.

## 6 Agradecimentos

Agradeço ao amigo Marcelo Bihre pelo compartilhamento do seu imenso conhecimento em Ciência da Computação, esclarecimentos de dúvidas sobre compiladores, sugestões e revisões, e principalmente por ter sugerido uma parte significativa das idéias implementadas nesta ferramenta. A implementação desta ferramenta é uma simplificação de idéias mais completas e elaboradas. Agradeço à minha esposa pela paciência e pela felicidade que me traz no dia-a-dia, refletida no resultado dos meus trabalhos. Agradeço aos meus pais por terem me dado tudo o que sempre precisei para me tornar uma pessoa produtiva. Agradeço a todos os gênios da Ciência da Computação, em especial Aho e Stroustrup, por terem nos apresentado com sabedoria. E agradeço a todos os demais amigos e pessoas que convivem comigo pelas incontáveis contribuições.

## 7 Referências Bibliográficas

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. (1986) **Compilers Principles, Techniques and Tools**, Reading, Massachusetts, Addison Wesley.

FOWLER, M. (2008) **Presentation Model**. Disponível em <http://martinfowler.com/eaDev/PresentationModel.html>

KLEENE, S. C. (1956) **Representation of events in nerve nets**, Shannon and McCarthy (1956), p. 3-40.

KNUTH, D. E. (1964), **Backus Normal Form vs. Backus Naur Form**, Comm. ACM **7:12**, p. 735-736.

KNUTH, D. E. (1965) **On the translation of languages from left to right**, Information and Control **8:6**, p. 607-639.

LESK, M. E. (1975) **Lex – a lexical analyzer generator**, Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N. J.

MICROSOFT (2008a) **Visual C# 2008 Express Edition**. Disponível em <http://www.microsoft.com/express/vcsharp/>

MICROSOFT (2008b) **Composite Application Guidance for WPF – June 2008**. Disponível em <http://msdn.microsoft.com/en-us/library/cc707819.aspx>

NAUR, P. (ED.) (1963) **Revised report on the algorithmic language Algol 60**, Comm. ACM **6:1**, 1-17.

TORTOISESVN (2008) **TortoiseSVN**. Disponível em <http://tortoisesvn.tigris.org/>