Gerador de Parsers LR

César Menin de Mello

Orientador: Gilberto Fernandes Marchioro

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

cmello@gmail.com

Resumo. *Este artigo apresenta o projeto de uma ferramenta para a construção automatizada de parsers. Recebendo como entrada a gramática de uma linguagem livre de contexto, a ferramenta gera um analisador sintático LR(1) para a mesma, ou um relatório descrevendo os erros e ambigüidades que impossibilitam a sua geração. O projeto também prevê o uso de interface gráfica para auxiliar o desenvolvimento e depuração da gramática.*

1 Introdução

Parser ou *analisador sintático* é um componente de software que interpreta um texto fornecido como entrada analisando sua estrutura gramatical.

São utilizados em diversas aplicações, como por exemplo:

- Compiladores e interpretadores de linguagens de programação;
- Interpretadores de consultas SQL em bancos de dados;
- Processamento de arquivos XML;
- Processamento de arquivos de configuração;
- Interpretadores de HTML em navegadores web;
- Analisadores de expressões.

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 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 para formar as diversas construções da linguagem.

A técnica de análise sintática utilizada neste trabalho é chamada LR, introduzida por Knuth [1965]. Também é conhecida como análise *bottom-up*.

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, pág. 215).

O algoritmo do *parser* LR é genérico e obtém suas ações de tabelas geradas mecanicamente a partir da gramática. O algoritmo de análise e os algoritmos que convertem a gramática em tabelas de ações para o *parser* são o tema central deste trabalho.

1.1 Motivação

O desenvolvimento de *parsers* manualmente é uma tarefa árdua, suscetível a erros e de difícil manutenção, principalmente no caso de um *parser* LR. Segundo Aho et al. [1984], "*a principal desvantagem da técnica é a quantidade excessiva de trabalho necessário para escrever um parser LR manualmente para uma linguagem de programação típica. É necessária uma ferramenta - um gerador de parsers LR*". Diante deste fato surge a maior motivação para o desenvolvimento de um gerador de *parsers*.

A possibilidade de validar a gramática através de uma ferramenta estimula o desenvolvimento de uma gramática formal e ao mesmo tempo garante que o *parser* para esta gramática possa ser implementado automaticamente com perfeição, eliminando completamente a possibilidade de erros de programação. A geração de relatórios de erros e ambigüidades da gramática é uma ferramenta valiosa para auxiliar no desenvolvimento e entendimento da mesma.

Os recursos de interface gráfica presentes nos sistemas atuais abrem novas possibilidades de interatividade na especificação e depuração de gramáticas. O resultado é uma maior produtividade assim como maior facilidade de uso e aprendizado.

Existem excelentes geradores de *parsers*, especialmente o clone *open-source* do YACC chamado Bison. Porém este não foi projetado tendo em vista a utilização em ambientes de interface gráfica interativos, assim como não prevê a geração de código para mais de um tipo de linguagem de programação, tornando difícil o seu aproveitamento para estes fins.

1.2 Objetivos

1.2.1 Objetivos Gerais

- Criar uma ferramenta que possibilite especificar, validar e depurar uma gramática utilizando um ambiente integrado, e a partir desta gramática gerar automaticamente um *parser* que efetue as ações especificadas em anotações da gramática;
- Facilitar o ensino e aprendizado do projeto de compiladores.

1.2.2. Objetivos Específicos

- Desenvolver uma arquitetura para representação, manipulação e armazenamento de gramáticas livres de contexto, independente de geração de código;
- Desenvolver o algoritmo LR Canônico para conversão da gramática em tabelas genéricas de ações para *parsers* LR;
- Desenvolver um *parser* genérico integrado ao ambiente de desenvolvimento que permita carregar textos e analisá-los com o objetivo de testar e depurar a gramática visualmente; as estruturas deste *parser* visam a depuração e possuem

informações detalhadas sobre os símbolos da gramática e seus relacionamentos, tendo em vista facilitar a compreensão das ações;

- Desenvolver um *parser* genérico compatível com o ambiente de programação onde o *parser* gerado será compilado; este *parser* visa a implementação com o mínimo de *overhead* possível para que a *performance* do *parser* gerado não seja prejudicada;
- Desenvolver um mecanismo de geração de código extensível para alimentar o *parser* com as tabelas de ação geradas para a gramática, assim como a *linkagem* com o código das ações especificadas pelo usuário; implementações para mais de uma linguagem de programação de destino poderão ser plugadas nesta arquitetura;
- Aprofundar os conhecimentos relacionados à geração de *parsers* tendo em vista o desenvolvimento de compiladores.

Na seção 2, será apresentado o referencial teórico; na seção 3, serão abordadas questões relacionadas ao projeto; a seção 4 resume a metodologia a ser seguida, enquanto na seção 5 é apresentado o cronograma; a seção 6 por fim apresenta as referências bibliográficas.

2 Fundamentação Teórica ou Revisão Bibliográfica

2.1 Análise Léxica

Antes de chegar ao analisador sintático, o texto de entrada passa por um componente chamado *analisador léxico*. 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. Considere o exemplo a seguir:

```
1150 + 50 / 3.5
```

Ao ler os caracteres desta expressão, um analisador léxico poderia gerar a seguinte saída:

Lexema	Token
1150	NUMERO
+	OPERADOR_SOMA
50	NUMERO
/	OPERADOR_DIV
3.5	NUMERO

Neste exemplo, a seqüência de *tokens* NUMERO, OPERADOR_SOMA, NUMERO, OPERADOR_DIV e NUMERO é a entrada que será fornecida ao analisador sintático.

O texto que compõe cada *token* isoladamente é chamado de *lexema*. No exemplo, o *token* NUMERO aparece três vezes, mas com lexemas diferentes (“1150”, “50” e “3.5” respectivamente).

Os *tokens* são representados por constantes numéricas ou enumerações. Para *tokens* como o OPERADOR_SOMA, nenhuma informação adicional é necessária, já que o único objetivo é identificar que o *token* lido representa um operador de soma na expressão. Outros tipos de *tokens* porém precisam carregar mais informações para serem úteis, como o próprio lexema ou atributos adicionais. Por exemplo, para o *token* NUMERO, o valor numérico que ele representa provavelmente será necessário para o usuário do *parser* (o usuário do *parser* na maioria das vezes será uma outra camada de software, como por exemplo um compilador). Neste caso o analisador léxico pode converter o texto do lexema para número e guardar em uma propriedade numérica de ponto flutuante. Quando o analisador sintático obter este *token*, poderá acessar o valor numérico pela propriedade. A implementação do atributo pode ser feita na forma de uma variável global, ou como um membro do tipo utilizado para representar o *token*.

Um outro tipo de *token* comum em linguagens de programação é o *identificador,* que representa nomes definidos pelo usuário. Como exemplo, nomes de variáveis e funções freqüentemente são representados por este *token*. Considere o exemplo a seguir de um *if* em linguagem C:

```
if (tamanho > limite)
```

Para o qual um analisador léxico poderia gerar a seguinte seqüência de tokens:

Token	Lexema
IF	if
ABRE_PARENTESE	(
IDENTIFICADOR	tamanho
MAIOR_QUE	>
IDENTIFICADOR	limite
FECHA_PARENTESE	)

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] cap. 3 descreve o funcionamento, implementação e geração automática analisadores léxicos. Lesk [1975] criou o gerador de analisadores léxicos mais 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. O analisador sintático chama esta função do analisador léxico sempre que precisa de 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.

Traçando um paralelo com a interpretação de textos na Língua Portuguesa, o analisador léxico percorre o fluxo de letras do texto de entrada identificando e classificando as palavras e sinais de pontuação. Sua saída é uma seqüência de palavras classificadas como artigos, substantivos, adjetivos, pronomes, preposições, advérbios, sinais de pontuação, etc.

Um analisador sintático recebe essa seqüência de palavras classificadas e identifica como elas se agrupam para formar as construções de sintaxe definidas na Gramática da Língua Portuguesa (sujeito, predicado, oração, período, etc). Além de identificar estas construções, o analisador sintático tem a função de detectar e informar eventuais violações de gramática encontradas no texto.

2.2.1 Gramáticas

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

- *Conjunto de símbolos Terminais*: este é o conjunto de todos os *tokens* que podem ser utilizados no texto de entrada; equivalem às classes de palavras e sinais de pontuação;
- *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.
- *Conjunto de Produções*: são as regras que definem a composição de cada não-terminal da linguagem;
- *Símbolo Inicial*: é um não-terminal que representa todos os textos de entrada válidos na linguagem.

Considere como exemplo a gramática de expressões aritméticas envolvendo apenas somas de números. Os terminais (*tokens*) poderiam ser definidos como ***NUMERO*** e ***OPERADOR_SOMA***. O único não-terminal desta gramática poderia ser nomeado EXPRESSAO. Esta gramática teria apenas duas produções, conforme mostrado abaixo:

EXPRESSAO → ***NUMERO***

EXPRESSAO → EXPRESSAO ***OPERADOR_SOMA*** EXPRESSAO

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].

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 (*tokens*) estão destacados com fonte em negrito e itálico.

A primeira produção significa que o não-terminal EXPRESSAO pode ser um token ***NUMERO***. Ou seja, a presença de um *token* número sozinho já caracteriza uma expressão 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.

O único não-terminal é também o símbolo inicial neste caso. 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 e sua respectiva seqüência de derivações na forma de árvore:

Entrada: 10 + 20 + 30

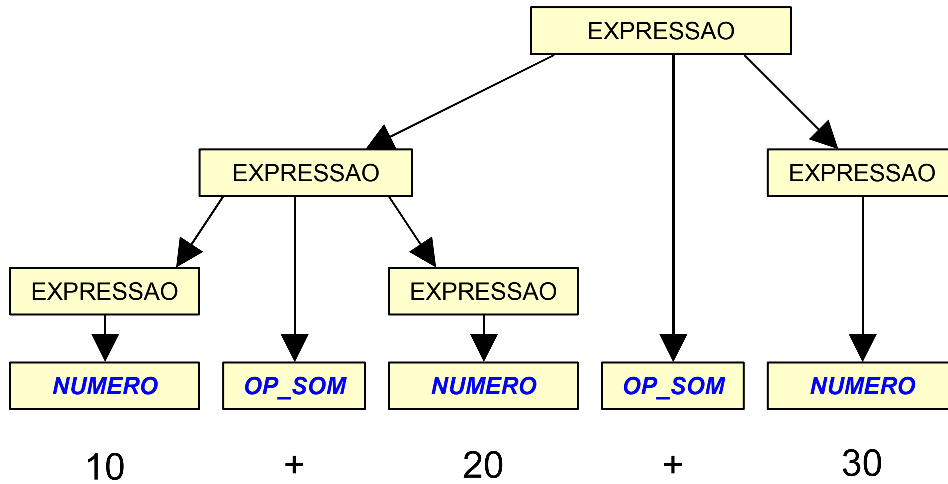

Figura 1

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*).

Esta mesma expressão também pode ser representada por outra árvore nesta gramática, conforme a figura 2.

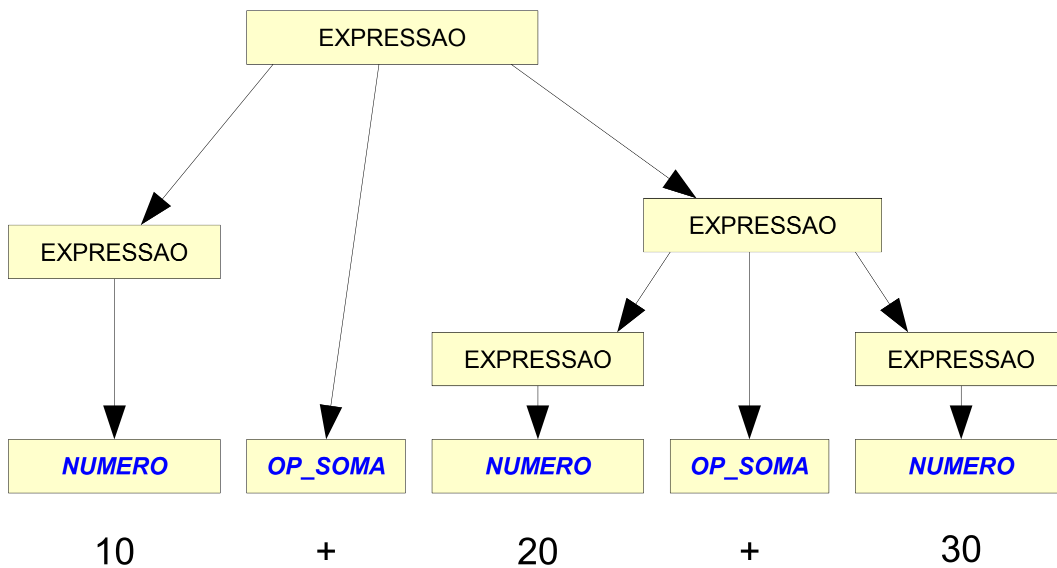

Figura 2

Quando a mesma entrada pode dar origem a derivações diferentes, a gramática é considerada *ambígua*. Gramáticas ambíguas não servem para os algoritmos de análise sintática abordados neste trabalho.

No caso desta gramática, 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:

EXPRESSAO → EXPRESSAO **OPERADOR_SOMA** TERMO

EXPRESSAO → TERMO

TERMO → **NUMERO**

Desta forma obtém-se uma árvore de derivação única para a expressão 10 + 20 + 30, ilustrada na figura 3.

EXPRESSAO
EXPRESSAO TERMO
EXPRESSAO TERMO
TERMO
NUMERO *OP_SOMA* *NUMERO* *OP_SOMA* *NUMERO*
10 + 20 + 30

Figura 3

2.2.3 Algoritmo genérico de análise LR

A análise sintática LR constrói a árvore de derivação para um texto de entrada partindo das folhas e subindo até a raiz. Por isso também é conhecida como análise “bottom-up”.

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 abordado posteriormente.

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 conter 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 a seguir (baseado em Aho, 1986, pág. 219):

- Inserir na pilha o estado inicial (0);
- 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* → *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 → w;

  2.4. caso contrário se *action[s, a] = aceitar*, termine com sucesso;

  2.5. caso contrário termine com erro.

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

- *Empilha s'*: (*shift s'*): informa que um estado deve ser empilhado;
- *Reduzir pela produção A → 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.

2.2.4 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, 1986, pág. 230). Este algoritmo utiliza as funções descritas a seguir:

Função FIRST(w): conjunto de *tokens* que podem iniciar derivações de *w*;

Função FOLLOW(N): conjunto de *tokens* que podem aparecer imediatamente após qualquer derivação de *N*.

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 → sXE e indicar a posição após o símbolo "s", utiliza-se a representação:

A → s.XE

- *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 ($).

Função CLOSURE(I):

repita

    para cada item [A → s.BE, a] em I,

    cada produção B → y em G',

    e cada terminal *b* em FIRST(Ea)

    tal que [B → .y, b] não está em I

        adicione [B → .y, b] ao conjunto I;

até que mais nenhum item possa ser adicionado ao conjunto I;

retorne I

Função GOTO(I, X):

seja J o conjunto de itens [A → sX.E, a] tal que

    [A → s.XE, a] está em I;

retorne CLOSURE(J);

Definição do algoritmo de construção do conjunto de itens da gramática G':

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.

A tabela *action* é construída com base no conjunto de itens retornados pelo algoritmo acima, da seguinte forma:

- O estado *i* é construído do item I*i*. As ações para este estado são determinadas da seguinte forma:

a) Se [A → 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 → s.] está em I*i*, então a ação para *action[i,* a*]* deve ser "*reduza A → s*" para todo *a* em FOLLOW(A); A não pode ser S'.

c) Se [S' → 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.

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

O estado inicial do *parser* é construído a partir do conjunto de itens contendo

[S' → .S].

2.3 Implementações existentes e proposta de sistema para TCC2

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 [1986], pág. 257.

A idéia do trabalho de TCC é implementar uma ferramenta que tire melhor proveito da interatividade com o usuário proporcionada pelos sistemas de interface gráfica da atualidade. A especificação e depuração de gramáticas pode ser bastante facilitada com o auxílio de um ambiente interativo.

Apesar do código fonte do *Bison* ser aberto, não é viável adaptá-lo para o objetivo desejado. 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. A entrada da gramática no sistema também ficará muito facilitada com o uso de um sistema baseado em interface gráfica.

Outra vantagem viabilizada pelo desenvolvimento de um gerador *parsers* do zero é o suporte a geração de código para mais de uma linguagem de programação (ex: C++, C#, Java). A arquitetura será planejada desde o início tendo esta extensibilidade em mente.

3 Projeto

O projeto proposto envolve o desenvolvimento dos seguintes módulos:

- Gerenciamento de Gramática;
- Interface gráfica para manipulação de Gramática;
- Algoritmo de construção de tabelas *action* e *goto*;
- *Parser* genérico para depuração e teste da gramática;
- Interface gráfica de depuração;
- *Parser* genérico para geração de código C++ nativo;
- Gerador do código das tabelas para C++.

A tecnologia a ser utilizada no desenvolvimento é a linguagem C++, utilizando a ferramenta de desenvolvimento Microsoft Visual C++ 2008 Express disponível gratuitamente para download na URL:

http://www.microsoft.com/express/vc/Default.aspx

O requisito de software para rodar o sistema completo com interface gráfica será o sistema operacional Microsoft Windows XP ou Microsoft Windows Vista. O restante do código que não envolve interface gráfica será independente de plataforma e portanto compatível com outros sistemas operacionais. Dependendo da disponibilidade de tempo no final do projeto, a interface gráfica também poderá ser portada para ambiente Unix.

O código gerado pela ferramenta será C++ padrão, porém dependendo da disponibilidade de tempo no final do projeto o suporte a outras linguagens também poderá ser implementado.

4 Metodologia

O desenvolvimento será feito seguindo os princípios da metodologia Ágil, cujo manifesto pode ser obtido na URL:

http://www.agilemanifesto.org/

Com destaque para os seguintes aspectos:

- Build automático com testes unitários para manter a qualidade alta durante todo o ciclo de desenvolvimento;
- Iterações curtas (previsão de lançamento de uma versão funcional por mês);
- Evitando criação de documentação que não seja imprescindível. A prioridade é o desenvolvimento de software que funcione bem e supere as expectativas.

5 Cronograma

O trabalho na implementação já foi iniciado com o desenvolvimento de protótipos, esboços e testes unitários dos algoritmos de análise LR e geração das tabelas. Abaixo um exemplo mostrando a execução do algoritmo para uma gramática simples de expressão com soma:

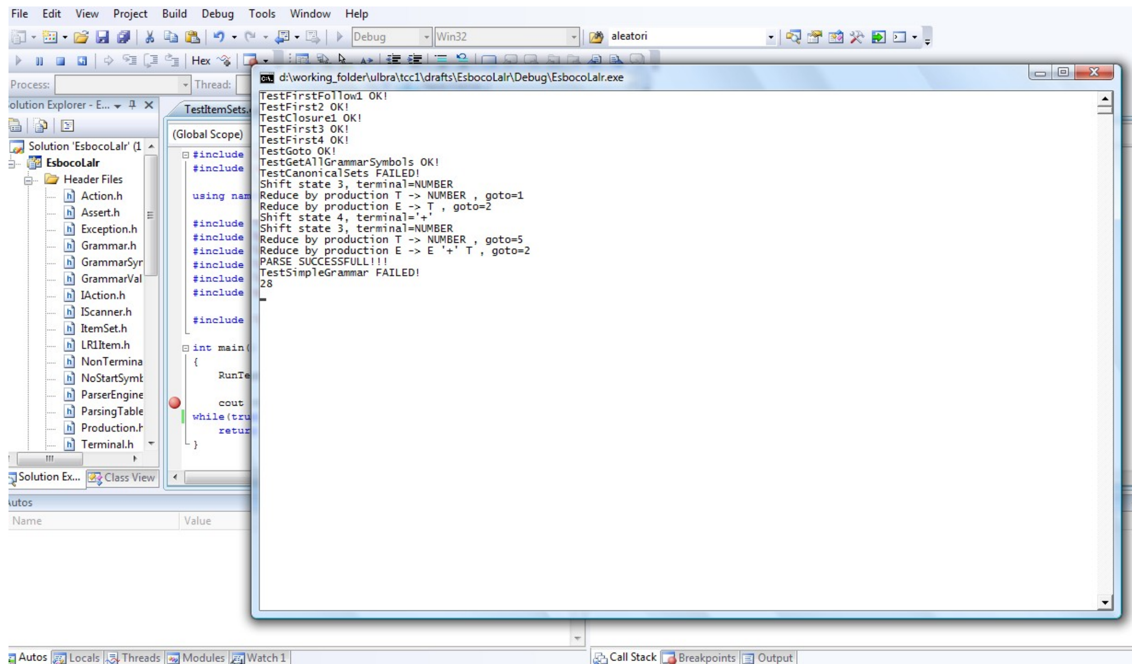

Figura 4

Com o desenvolvimento destes esboços foi possível reduzir o risco no prazo dos algoritmos mais difíceis, chegando ao seguinte cronograma:

	Julho	Ago	Set	Out	Nov	Dez
Gerenciamento da Gramática	x					
Algoritmo de geração tabelas	x					
Parser genérico para depuração		x	x			
Interface gráfica ger. gramática	x	x				
Interface gráfica depuração gram.			x	x	x	
Parser p/ C++ e gerador tabelas C++					x	
Entrega e apresentação						x

6 Referências Bibliográficas

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

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

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

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

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

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