Universidade Federal de Minas Gerais
Pós-Graduação em Ciência da Computação

Ferramenta Visual para Desenvolvimento de Analisadores Sintáticos em um Ambiente de Múltiplos Geradores

Proposta de Dissertação de Mestrado

Proponente: Leonardo Teixeira Passos

Orientadora: Mariza Andrade da Silva Bigonha

Co-orientador: Roberto da Silva Bigonha

Belo Horizonte, 20 de fevereiro de 2006

Universidade Federal de Minas Gerais
Pós-Graduação em Ciência da Computação

Ferramenta Visual para Desenvolvimento de Analisadores Sintáticos em um Ambiente de Múltiplos Geradores

Proposta de Dissertação de Mestrado

Proponente: Leonardo Teixeira Passos

Orientadora: Mariza Andrade da Silva Bigonha

Co-orientador: Roberto da Silva Bigonha

Belo Horizonte, 20 de fevereiro de 2006

Sumário

1 Definição do Problema

Os projetistas de compiladores utilizam comumente ferramentas para geração de analisadores sintáticos, sendo esta abordagem útil quando se deseja um desenvolvimento rápido e seguro [25].

Dentre o universo dessas ferramentas, destacam-se os geradores Yacc [24], Javacc [5], Bison [19], CUP [2] e SableCC [21] e os ambiente visuais de desenvolvimento de analisadores sintáticos Visual Parse++ [8] e ProGrammar [6].

Apesar de úteis e valiosas tanto no ensino quanto em projeto profissionais, é possível identificar sérias deficiências nessas ferramentas:

Recursos para Resolução de Conflitos: Os geradores de analisadores sintáticos não oferecem mecanismos que facilitem ao usuário a correção de conflitos resultantes de especificação de gramáticas de linguagens de programação, tampouco acompanhar interativamente a execução da análise sintática com o intuito de validar a sua especificação. Estes dois fatores geram vários problemas durante a geração de um analisador sintático, dentre os quais merecem destaque:

- gasto considerável de esforço e tempo para a correção de conflitos. No caso de gramáticas LR, isto se dá pelo fato do projetista não dispor de mecanismos de visualização do autômato a partir do qual a tabela sintática é construída;

- extrema dificuldade por parte do usuário para relacionar a especificação da gramática com o analisador sintático gerado. Isto ocorre porque a especificação, fornecida em uma linguagem que evidencia as estruturas da linguagem para a qual o analisador sintático é construído, é convertida em um código escrito em C++, Java, etc., onde são realizados inúmeros controles que acabam obscurecendo tal relacionamento;

- intrusão para a realização da depuração da especificação da gramática. O projetista se vê obrigado a incluir código no analisador sintático gerado para que possa acompanhar os passos realizados pelo mesmo;

- dificuldade no entendimento de como as estruturas de dados utilizadas pelo analisador sintático funcionam e como elas colaboram para a aceitação/rejeição de uma *string*. Este problema é muito comum entre projetistas que implementam um analisador sintático pela primeira vez [33, 16, 26, 25].

Compactação das Tabelas LR: O tamanho das tabelas LR, quando criadas para analisadores sintáticos de linguagens de alto nível, como C++, Ada, Pascal, etc., consomem uma quantidade satisfatória de memória. Desta forma, muitos geradores de analisadores sintáticos oferecem mecanismos para compactação de tabelas sintáticas. No entanto, ainda não existe nos geradores estudados [24, 5, 19, 2, 21] o conceito de níveis de compactação. Um nível de compactação serve para classificar o grau de compactação da tabela sintática e o conseqüente aumento no tempo de execução. Em analogia aos níveis de otimização fornecidos pelo compilador GCC [3], os níveis de compactação podem ser classificados em alto, médio e baixo. O primeiro corresponde ao maior grau de compactação, mas com uma deteriorização no tempo de execução. O segundo oferece uma taxa de compactação mediana, com uma leve deteriorização no tempo de execução. Finalmente, no terceiro a taxa de compactação é baixa, mas o tempo de execução é praticamente inalterado em relação ao uso da tabela não compactada. A oferta de um mais de um nível de compactação é necessária, pois possibilita a geração de analisadores sintáticos conforme a disponibilidade de memória dos computadores que irão executá-los. Por exemplo: um analisador sintático projetado para execução em celulares poderá utilizar o nível de compactação mais alto, ao passo que um projetado para execução em PDAs poderá utilizar o nível mediano.

Dificuldade de Integração com Múltiplos Geradores: Os principais ambientes visuais de desenvolvimento de analisadores sintáticos existentes [8, 6] não fornecem meios de integração com outros analisadores sintáticos, exceto o originalmente considerado durante o projeto dos mesmos. Isto faz com que os usuários desses ambientes, de forma a usufruírem das facilidades visuais fornecidas, fiquem presos a um único gerador de analisador sintático e possivelmente a uma única linguagem de especificação.

De forma a amenizar os problemas identificados, a dissertação de mestrado proposta neste texto visa o projeto e a implementação de uma ferramenta visual para a construção de analisadores sintáticos LR, cujas principais facilidades são:

- mecanismos visuais para auxiliar na solução dos conflitos LR. Para isto será disponibilizada a máquina de estados, com a indicação dos conflitos para que os usuários possam corrigí-los;
- animação da execução do analisador sintático gerado, com a apresentação das estruturas de dados envolvidas e o estado de cada uma

delas a cada passo da execução;

- flexibilidade na escolha do gerador de analisador sintático. Isto visa oferecer um ambiente no qual o projetista poderá escolher um dentre vários geradores de analisadores sintáticos disponíveis (Yacc, SableCC, etc.) conforme sua familiaridade e/ou conveniência;

- disponibilidade do mesmo conjunto de funcionalidades, independentemente do gerador de analisador sintático escolhido;

- compactação das tabelas sintáticas geradas pelos diferentes geradores de analisadores sintáticos, com a flexibilidade de escolha do nível de compactação desejado.

Isto posto, o trabalho a ser desenvolvido compreende as seguintes atividades:

- projeto e implementação de uma ferramenta visual que seja modular e multiplataforma para auxiliar no desenvolvimento de analisadores sintáticos LR;

- projeto de um arcabouço[1] para o desenvolvimento de *plugins*. Tal arcabouço é responsável pela definição da interface de acoplamento entre a ferramenta proposta e os geradores de analisadores sintáticos LR a serem utilizados. A grande contribuição do arcabouço é permitir um conjunto uniforme de funcionalidades, independentemente do gerador de analisador sintático utilizado;

- validação do arcabouço em questão. Para isto serão implementados dois *plugins*: um para o gerador Bison [19] e outro para o SableCC [21];

- realização de um estudo, aos moldes da realizada por Dencker [18], com a implementação e avaliação dos métodos de compactação de tabelas sintáticas LR. Isto tem por objetivo identificar os métodos a serem disponibilizados na ferramenta proposta, de acordo com os níveis de compactação mencionados anteriormente;

- projeto de uma linguagem de interface para mapear as diferentes construções das linguagens de especificação de geradores de analisadores sintáticos em uma única representação independente das mesmas.

[1] Tradução do termo *framework*.

1.1 Contribuições da Dissertação

As contribuições esperadas por este trabalho são:

- análise e avaliação atualizada dos métodos de compactação de tabelas. O último estudo deste tipo foi realizado por Dencker [18] em 1984, e desde então novos métodos surgiram, demandando novas comparações com os demais métodos existentes;

- geração de analisadores sintáticos com tabelas sintáticas compactadas, com a possibilidade de escolha do nível de compactação. Os métodos disponibilizados em cada nível de compactação serão os identiticados a partir do estudo mencionado no item anterior;

- construção de uma ferramenta visual para auxiliar no desenvolvimento de analisadores sintáticos. A ferramenta proposta, além de unificar as características das principais ferramentas visuais atualmente existentes [33, 16, 26, 25, 8, 6], tais como depuração visual dos conflitos referentes a má especificação de uma gramática e animação passo a passo do analisador sintático gerado, permitirá a integração com vários geradores de analisadores sintáticos, com o mesmo conjunto de funcionalidades;

- disponibilização de um arcabouço que permita o desenvolvimento de *plugins* responsáveis por realizar a integração da ferramenta com qualquer gerador de analisador sintático LR;

- disponibilização de uma linguagem de interface que mapeie as diferentes construções sintáticas dos geradores de analisadores sintáticos para um sintaxe única, facilitando a integração de diferentes ferramentas.

2 Projeto de Arcabouços

Um arcabouço consiste em um desenho reutilizável de um sistema, ou parte dele, representado por um conjunto de classes abstratas com um mecanismo de integração bem definido [23].

O objetivo de um arcabouço é estabelecer uma interface a partir da qual seja possível construir uma aplicação de domínio específico.

As principais vantagens da utilização de arcabouços são:

- modularidade: distribuição de responsabilidades via classes e seus métodos integrantes;

- reusabilidade: disponibilização de componentes genéricos;

- extensibilidade: facilidade de extensão;
- inversão de controle: para responder a certos eventos, o arcabouço transfere o fluxo de execução para tratadores de eventos previamente registrados. Isto garante um fluxo de execução bem definido, mas ao mesmo tempo customizável.

Essas vantagens, no entanto, tornam o projeto de um arcabouço mais complexo do que o projeto de aplicações [20].

Schmid [27] apresenta uma técnica sistemática para facilitar o projeto de arcabouços, que compreende basicamente em:

1. obter uma estrutura de classes que modele uma aplicação específica do domínio no qual o arcabouço pertence. Esta fase é denominada *atividade de modelagem*, cujo produto é um conjunto de classes específicas da aplicação em questão;
2. aplicar uma análise denominada *hot spot analysis*, que coleta as possíveis variabilidades (*hot spots*) de um domínio de aplicação, documentando-as em um documento denominado *Requisitos de Variabilidade*. Diferentes aplicações de um mesmo domínio diferem entre si em pelo menos um *hot spot*;
3. generalizar sucessivamente as classes obtidas no Passo 1 de modo a incorporar as variabilidades identificadas anteriormente. A cada passo é aplicada uma sequência de generalizações, uma para cada *hot spot*. Desta forma, para cada classe específica onde for identificada um ou mais *hot spots*, tem-se-á uma substituição por um subsistema *hot spot*, capaz de modelar as variabilidades encontradas.

3 Ferramentas Visuais para Auxílio no Projeto de Analisadores Sintáticos

Nesta seção são apresentadas as seguintes ferramentas para auxiliar a construção de analisadores sintáticos: *Visual Tools* [33], LLparse e LRparse [16], XTango [26], CUPV [25], Visual Parse++ [8] e ProGrammar [6].

Tais ferramentas podem ser classificadas basicamente em duas categorias: *ferramentas educativas* e *ferramentas comerciais*. As ferramentas educativas possuem um número limitado de funcionalidades e objetivam evidenciar e acompanhar o processo de análise sintática na medida em que o mesmo é executado. As ferramentas *Visual Tools*, LLparse e LRparse, XTango e CUPV

estão nesta categoria. Já as ferramentas comerciais, ao contrário das contidas na primeira categoria, são mais complexas, apresentando um número maior de funcionalidades. Este é o caso das ferramentas Visual Parse++ e ProGrammar.

3.1 Ferramentas Educativas

Visual Tools é um conjunto de ferramentas para auxiliar no entendimento das fases de um compilador. Uma de suas ferramentas, *Tree Viewer*, é utilizada para visualizar a árvore de sintaxe abstrata (ASA)[2] produzida pelo analisador sintático ao processar uma entrada específica. É permitido ao usuário:

- navegar pela ASA, expandindo ou retraindo nodos;
- solicitar à ferramenta a indicação dos nodos ligados a um nodo específico. Isto pode ser feito, por exemplo, para se obter a cadeia de uso e definição, utilizada em fases posteriores à análise sintática.

XTango [26] é uma ferramenta construída com o arcabouço Tango[30], desenvolvido para animação de algoritmos em sistemas X-Windows. XTango é utilizado como instrumento para entendimento do processo de análise sintática *top-down* e *bottom-up*. Este sistema recebe como entrada uma tabela sintática LL(1) ou SLR(1). Uma vez carregada a tabela, o usuário fornece uma *string* de entrada para ser processada pelo algoritmo de análise sintática. A execução desse algoritmo é animada, sendo mostrados passo a passo os estados das estruturas de dados envolvidas. A tela principal exibe basicamente quatro itens: a pilha, a *string* de entrada, a caixa de ação e a árvore de derivação. As Figuras 1 e 2(a) apresentam respectivamente uma animação para uma gramática LL(1) e uma outra para um gramática SLR(1). No caso das animações SLR(1), é permitido ao usuário uma visualização tabular, onde mostra-se a pilha, o quanto a *string* de entrada foi consumida e as ações tomadas a cada momento (ver Figura 2(b)). Esta visualização pode ser utilizada em conjunto com a apresentada na Figura 2(a).

Kaplan em [25], apresenta uma ferramenta denominada CUPV, uma extensão do gerador CUP[2]. Esta ferramenta gera analisadores, que ao serem executados, apresentam uma interface gráfica mostrando cada passo de execução do algoritmo de análise sintática. No final de cada passo, o usuário deve sempre solicitar a execução do passo seginte. Na tela principal, são mostradas a pilha sintática, o símbolo de entrada corrente, o estado

[2]A sigla correspondente em inglês é AST - *Abstract Syntax Tree*

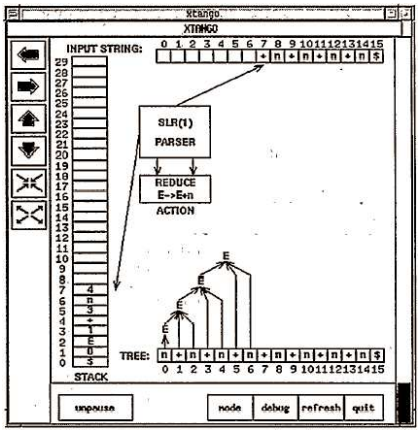
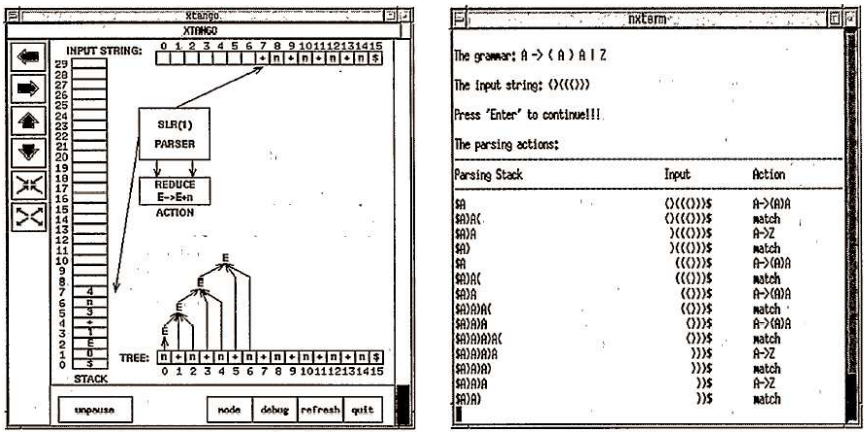


Figura 1: XTango - Animação de uma dada gramática LL(1). Extraído de [26]



(a) visualização normal (b) visualização tabular

Figura 2: XTango - Animação de uma dada gramática LR(1). Extraído de [26]

atual e o histórico de ações de reconhecimento sintático tomadas até o momento. Ainda nessa tela, é permitido ao usuário determinar um incremento de execução, ou seja, quantos passos o algoritmo deverá executar antes de solicitar a continuação ao usuário. A tela principal é apresentada na Figura 3. Além dessas facilidades, a ferramenta permite também a visualização de:

- valores semânticos, se existirem, de elementos da pilha sintática;
- conjunto de itens de um dado estado, onde para cada produção pode-se

exibir o conjunto de *lookaheads* a ela associado;

- reduções, apresentando a produção utilizada; indicação de que o lado direito, que está no topo da pilha, irá ser substituído pelo lado esquerdo; estado antecessor e o estado corrente.

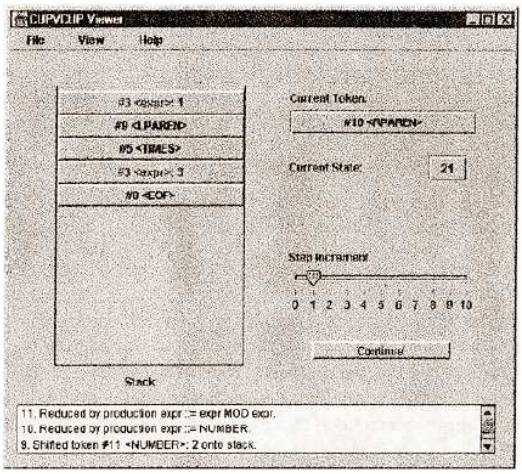


Figura 3: CUPV -Tela principal. Extraído de [25]

CUPV foi construído com o arcabouço disponibilizado em [29]. Tal arcabouço é utilizado para definir e customizar visualizações sobre analisadores sintáticos.

Finalmente, Blythe [16] apresenta duas ferramentas visuais para definição de analisadores sintáticos: *LLparse*, utilizada na definição de analisadores LL(1) e LRparse, utilizada na definição de analisadores LR(1). Na maior parte do tempo, as telas disponibilizadas nessas ferramentas são as mesmas. O processo de definição se dá de forma incremental, sendo que o próximo estágio de definição só é disponibilizado quando o anterior é concluído com sucesso. A definição de um analisador sintático LL(1) é composta por 3 estágios, que consistem respectivamente das seguintes definições fornecidas pelo usuário:

- gramática LL(1). Esta gramática não pode ter mais que 15 regras;
- conjuntos *FIRST* e *FOLLOW*;
- tabela sintática.

Após essas definições, a ferramenta LLparse permite que o usuário forneça uma *string* de entrada, onde então o sistema anima a execução do analisador sintático para a entrada fornecida. No caso da ferramenta LRparse os estágios são os mesmos, exceto pela gramática, que neste caso é LR(1), e pelo acréscimo de um novo estágio referente à definição da máquina de estados.

3.2 Ferramentas Comerciais

Nesta seção são descritas duas ferramentas comerciais para o desenvolvimento de analisadores sintáticos: VisualParse++[8] e ProGrammar[6].

VisualParse++ (VP++) é um ambiente integrado de desenvolvimento de analisadores sintáticos LALR(k). As especificações de gramáticas combinam a especificação do analisador léxico e sintático. Tais especificações devem ser escritas em uma sintaxe própria do VP++, apesar de ser possível importar especificações escritas para o Yacc[24].

Dentre as várias características do VP++, destacam-se:

- editor para criação e edição de especificações de gramáticas juntamente com as regras referentes aos símbolos terminais da linguagem;
- gramáticas consideradas ambíguas por geradores LR(1) são aceitas pelo VP++, pelo fato do mesmo ser um gerador LALR(k);
- a compilação da especificação verifica além da sintaxe, especificações de símbolos terminais duplicados e produções inalcançáveis, eliminando-as;
- disponibilização da visão *Conflict Trace View*, que na ocorrência de conflitos *shift-reduce* ou *reduce-reduce*, mostra uma árvore de derivação com o indicativo visual do motivo da ocorrência do conflito;
- integração do analisador gerado com programas escritos em C, C++, Java, Object Pascal ou Visual Basic (via controladores ActiveX);
- execução passo a passo do analisador sintático para uma entrada específica.

A ferramenta ProGrammar é um ambiente visual para a construção de analisadores sintáticos. Dentre as características desta ferramenta, destacam-se:

- ambiente gráfico de desenvolvimento;

- navegação visual de gramáticas e árvores de derivação;
- integração do analisador gerado com programas escritos em C, C++, Object Pascal, Visual Basic e com qualquer ambiente que possua suporte a componentes ActiveX;
- a linguagem GDL (*Grammar Definition Language*), utilizada para a escrita de definições de gramáticas. Por ser orientada a objetos, essa linguagem permite, por exemplo, que uma gramática herde todos os símbolos e regras de uma gramática pai, podendo seletivamente sobrescrevê-las ou extendê-las;
- definições escritas em GDL são independentes de linguagens de programação, o que permite que a partir de uma mesma definição possam ser obtidos analisadores sintáticos que poderão ser acoplados a programas codificados nas linguagens nas quais ProGrammar possui integração;
- listagem dos conflitos resultantes da especificação;
- execução passo a passo do analisador sintático para uma entrada específica.

Não foram encontradas na página do fabricante qualquer informação a respeito da classe do analisador sintático produzido, se LL(k) ou LR(k).

3.3 Avaliação das Ferramentas

As ferramentas apresentadas nesta seção não fornecem meios de serem integradas com outros geradores de analisadores sintáticos, senão o originalmente considerado durante o projeto das mesmas. Isto gera um forte acoplamente entre a aplicação e o gerador de analisador sintático e inflexibiliza o usuário de escolher o gerador de analisador sintático a ser utilizado. Além disto, o usuário fica restrito possivelmente a uma única linguagem de especificação.

O grande diferencial da ferramenta proposta quando comparada às ferramentas *Visual Tools*, LLparse/LRparse, XTango, CUPV, Visual Parse++ [8] e ProGrammar [6] é que a mesma fornece um conjunto uniforme de funcionalidades independentemente do gerador de analisador sintático em uso. Este conjunto engloba funcionalidades oferecidas tanto pelas ferramentas educativas quanto pelas comerciais, que são basicamente a disponibilidade de mecanismos visuais para correção de conflitos oriundos de uma má especificação e animação da execução do analisador sintático, com a apresentação dos estados das estrutura de dados envolvidas.

Além disto, a ferramenta proposta visa a geração de analisadores sintáticos com tabelas sintáticas compactas. Isto permite ao usuário: (i) usufruir de mecanismos de compactação quando o gerador utilizado não oferecer tal recurso; (ii) mesmo no caso do gerador fornecer algum mecanismo de compactação, o usuário possui a flexibilidade da escolha do grau de compactação a ser realizado, tendo em vista o impacto no tempo de execução.

4 Tabelas LR

Uma tabela sintática LR é normalmente representada por uma matriz decomposta em duas tabelas: *Action* e *Goto*. Em ambas, as linhas representam estados. As colunas em *Action* representam símbolos terminais e em *Goto* os símbolos não-terminais da gramática.

As entradas na tabela *Action* e *Goto* representam ações no reconhecimento sintático. Na tabela *Action* são encontradas as seguintes entradas: s_k, indica a ação de empilhar o k-ésimo estado; r_n, denota a ação de redução utilizando-se a n-ésima regra da gramática; *acc*, indica a aceitação da *string* de entrada e finalmente entradas de erro, simbolizadas por entradas em branco. A tabela *Goto* é essencialmente a representação de uma função de transição de um autômato finito determinístico, com exceção de que entradas em branco não indicam erro e nunca são acessadas. A única entrada possível nessa tabela são os estados de destino.

As tabelas LR são consultadas pelos algoritmos de análise sintática *bottom-up* [10] como guias de sua execução.

4.1 Métodos de Compactação de Tabelas LR

Uma tabela LR gerada para utilização em um analisador sintático construído para uma linguagem de alto nível, como C, Ada, etc., possui um número excessivo de entradas. Por exemplo: um experimento apresentado em [14] mostra que a tabela obtida para a linguagem Ada possui 810 estados, 93 terminais e 200 não-terminais, o que totaliza $810 * (93 + 200) = 237.330$ entradas.

Desta forma, é necessário a aplicação de métodos de compressão de tabelas LR de modo a minimizar o uso de memória. Isto é extremamente desejável, por exemplo, em analisadores sintáticos projetados para execução em dispositivos móveis, como PDAs (*Portable Digital Assistants*) e celulares. Nesses aparelhos, qualquer acréscimo de memória acarreta em um maior consumo de energia, o que diminui a autonomia do aparelho.

Nesta seção são apresentados os seguintes métodos de compactação de tabelas: compactação proposta por Aho [10], compactação proposta por Bigonha *et alli* [15], compactação por submatriz [11], compactação *row displacement* [34], compactação por coloração de grafo [28], compactação por eliminação de linhas [13], compactação por distância significativa [12], compactação *row column* [32] e compactação por supressão de zeros [18].

Com exceção dos três primeiros métodos, os demais métodos de compactação operam da mesma forma tanto na tabela *Action* quanto na tabela *Goto*. Assim, por questões de simplificação, tais métodos são apresentados somente em função da tabela *Action*, definida como: Action $[1\,..\,m, 1\,..\,n]$. Essa tabela, ao contrário da tabela *Goto*, não é esparsa por natureza, pois contém muitas entradas de erro. Para ser possível compactá-la com técnicas de compressão de matrizes esparsas, as entradas de erro devem ser fatoradas em uma matriz à parte, denominada *sigmap* [18]. Nessa matriz, entradas de não erro são representadas pelo valor lógico verdadeiro e as entradas de erro pelo valor lógico falso. O armazenamento da matriz *sigmap* consome uma quantidade satisfatória de memória. Para amenenizar isto, Dencker [18] sugere um esquema de compactação de tabelas binárias.

Existem duas estratégias principais utilizadas na compactação da tabela *Action*: remoção das entradas de erro e diminuição do número de entradas referentes a ações de redução.

Nos métodos citados, ações de redução e erro são algumas vezes referenciadas como ações padrão. Dada uma linha i da tabela *Action*, a ação padrão será de erro caso não exista nenhuma ação de redução em i, caso contrário, a ação de redução em questão, que é sempre única para um dado estado.

Método de compactação proposto por Aho [10]: Este método utiliza um vetor *rowmap* responsável por mapear cada linha i da tabela *Action* em uma lista contendo suas respectivas ações sintáticas. As ações sintáticas são pares (*símbolo terminal*,*ação sintática*) armazenados em duas posições contíguas de uma dada lista. Desta forma, se a entrada para o terminal a estiver na posição p, então a posição $p+1$ contém a ação a ser tomada. Toda lista possui uma ação padrão para todos os terminais que não possuem uma entrada na lista. Para sinalizar isto, utiliza-se um símbolo não pertencente ao conjunto de terminais, como por exemplo '#'. A entrada após esse símbolo contém a ação padrão a ser executada. Neste método, duas linhas iguais, digamos i e k, são codificadas em uma única lista. Neste caso, tem-se que $rowmap[i] = rowmap[k]$. Para simular o acesso a $Action[i, j]$ percorre-se a lista apontada por $rowmap[i]$. Se a entrada '#' for alcançada, então a ação padrão, que está armazenada na posição seguinte a '#', é retornada. Caso

contrário, uma entrada referente ao símbolo j foi encontrada. A entrada na posição seguinte à de j contém a ação desejada.

A compactação da tabela *Goto* utiliza um vetor denominado *gotomap* responsável por mapear cada não-terminal em uma lista de pares da forma (*estado corrente*,*próximo estado*). Este esquema baseia-se na propriedade de que cada estado aparece no máximo em uma coluna dessa tabela e que entradas em branco nunca são acessadas. As entradas em branco podem ser substituídas pelo estado que ocorre com maior freqüência na coluna que as contêm. Tal estado é o estado padrão para a coluna em questão. De forma a sinalizar isto, o estado padrão é sempre precedido do símbolo '#', de forma análoga ao esquema adotado na compactação da tabela *Action*. Para simular o acesso a uma entrada $Goto[i, X]$ percorre-se a lista apontada por $gotomap[X]$. Se a entrada '#' for alcançada, então o estado destino é o armazenado na entrada posterior a '#'. Caso contrário, uma entrada referente ao símbolo X foi encontrada. A entrada na posição seguinte à de X contém o estado destino.

De acordo com Aho, a compactação das tabelas *Action* e *Goto* utilizando o esquema descrito apresenta uma boa taxa de compressão, economizando em torno de 90% do espaço originalmente necessário.

Apesar de diminuir consideravelmente a complexidade espacial, este método acrescenta um custo na complexidade temporal, já que substitui o acesso à matriz original - custo $O(1)$, por um percorrimento linear. Este percorrimento custa no pior caso $O(m)$, para simular o acesso a uma entrada em *Action*, e $O(n)$, para simular o acesso à tabela *Goto*, onde m e n são respectivamente o número de linhas e colunas da tabela sintática. Além disto, poderão ser realizadas reduções desnecessárias quando o correto seria a detecção de um erro.

Método de compactação proposto por Bigonha *et alli* [15]: este método codifica a tabela sintática em um vetor denominado LR. Tal vetor é divido em intervalos, um para cada estado, onde as suas entradas correspondem a transições para outros estados em LR. A primeira entrada de cada intervalo é sempre o símbolo de acesso para o estado em questão; as demais armazenam posições de estados sucessores.

No vetor LR existem três estados especiais responsáveis pela indicação de erro, redução e aceitação, ocupando respectivamente E, R e F. Um estado que contenha somente ações de empilhamento tem com última ação em seu respectivo intervalo uma transição para o estado E. No caso de um estado possuir pelo menos uma ação de redução, a mesma é sempre codificada na última posição do intervalo correspondente. Essa transição é codificada

armazenando-se p_{rn} e R nas duas últimas posições desse intervalo, onde p_{rn} é a posição do estado em LR correspondente à ação de redução r_n.

Os estados de redução sempre ocupam posições posteriores à posição de R. Para cada um existe duas entradas: uma para o símbolo de acesso e a outra para a produção a ser utilizada. A primeira entrada é sempre desprezível, já que todo símbolo lido da *string* de entrada é sempre salvo como o símbolo de acesso de R e de E. No entanto, é mantida somente por questões de uniformidade.

O mecanismo geral do algoritmo de análise sintática com este método de compactação basea-se em transições. A cada transição para um estado k, três cenários são possíveis: (i) a posição de k em LR é menor que E - a ação representada neste caso é a de empilhamento (*shift* k); (ii) a posição de k em LR é maior que R - tem-se então uma redução de acordo com uma produção, cujo o identificador numérico encontra-se armazenado em $LR[k+1]$; (iii) os estados final ou de erro são alcançados, indicando respectivamente aceitação ou erro.

A tabela *Goto* é por construção embutida no vetor LR. Isto é possível a partir do momento em que é permitido símbolos não-terminais como símbolos de acesso.

A forma como a compactação é realizada neste método contrasta com a idéia proposta por Aho. Aho elimina listas iguais e codifica as entradas das listas como pares da forma (*símbolo terminal*, *ação sintática*). Neste método, tais pares são substituídos pelos símbolos de acesso e pelas posições dos estados no vetor LR, o que permite recuperar os dois componentes de um dado par. Além disto, as ações de redução pertencentes aos pares da forma (*símbolo terminal*, r_n), ao contrário do que ocorre no método proposto por Aho, são codificadas uma única vez. No entanto, a posição para o estado referente à n-ésima redução seja referenciada em vários intervalos do vetor LR.

As condições necessárias à eficiência deste método são [15]:

1. a relação *número de transições/número de estados* deve ser razoavelmente maior que 1. Quanto maior esta relação, maior é a taxa de compactação;

2. a relação *número de linhas repetidas/total de linhas* deve ser maior que $a/(c*m)$, onde a é o espaço necessário ao armazenamento de um ponteiro, m é o número médio de entradas por linha e c é o espaço ocupado por um par (*coluna*, *ação sintática*) utilizado no método proposto por Aho;

3. entradas de erro deverão ser a entrada mais freqüente em cada linha.

Satisfeitas a primeira e terceira condições, Bigonha [15] afirma que a taxa de compactação deste método atinge em média 96%.

Quando a segunda condição não é satisfeita, o método proposto por Aho apresenta uma compactação melhor, pois as linhas duplicadas são eliminadas.

Do ponto de vista do tempo de execução, os ganhos e perdas deste método são basicamente os mesmos do método apresentado por Aho, ou seja, tem-se a diminuição da complexidade temporal com um tempo de execução maior, devido a reduções desnecessárias e à utilização de um percorrimento linear em substituição a um método de acesso direto.

Método de compactação por submatriz [11]:

Compactação da Tabela *Action***:** este método particiona as linhas da tabela, de modo a dividí-la em um conjunto de submatrizes. Para realizar o particionamento são consideradas dois tipos de símbolos terminais para cada linha existente: os que contém ações sintáticas iguais a empilhamento ou aceitação, denominados *símbolos terminais significativos*, e os que se referem a ações de erro ou redução, denominados *símbolos terminais padrão*.

O particionamento inicia-se pelo cálculo do conjunto de símbolos terminais significativos (CTS) de cada linha. Quando o CTS de duas linhas forem iguais, cada uma se torna uma linha na submatriz $table_n$, onde n representa n-ésimo particionamento realizado.

Neste método são utilizados também os seguintes vetores auxiliares: *table*, *columnmap*, $columnmap_n$ e *row*. Os vetores *table* e *columnmap*, dado um estado, retornam respectivamente a instância de $table_n$ e $columnmap_n$ a serem utilizadas. $columnmap_n$ mapeia um símbolo terminal em uma coluna da tabela $table_n$. Finalmente, *row* obtém a linha da tabela $table_n$, dado um estado. Note que para cada submatriz $table_n$ obtida, ter-se-á uma instância de $columnmap_n$.

As colunas em $columnmap_n$ referentes a símbolos terminais não significativos sempre mapeiam para uma coluna específica em $table_n$, responsável por armazenar a ação padrão de cada estado pertencente à submatriz.

O acesso a uma entrada $Action[i, j]$ é obtido por $t[row[i], col[j]]$, onde $t = table[i]$ e $col = columnmap[i]$.

Este método apresenta uma baixa taxa de compactação, pois linhas iguais em uma mesma submatriz não são combinadas. Além disto, faz uso de um número excessivo de vetores auxiliares, como por exemplo, o vetor $columnmap_n$, que existe para cada submatriz obtida. No entanto, o tempo original de acesso à tabela *Action* é preservado, mas reduções desnecessárias poderão ser realizadas.

Compactação da Tabela *Goto*: para realizar a compactação dessa tabela, linhas em branco são eliminadas e linhas iguais são combinadas em uma única linha, resultando em uma tabela $Goto'$. Para auxiliar o processo, utiliza-se um vetor *state*, responsável por armazenar o índice da linha em $Goto'$ correspondente a um dado estado. Desta forma, para simular o acesso à tabela *Goto*, dado um estado i e um símbolo não-terminal X, faz-se: $Goto'[state[i], X]$.

Método de compactação *row displacement* [34]: este método distribui não conflitantemente as entradas iguais a s_n e *acc* de cada linha da tabela *Action* em um vetor denominado *value*. Uma distribuição não conflitante é uma que não mapeia duas entradas de linhas diferentes para uma mesma posição em *value*.

Para cada linha copiada, registra-se a posição inicial de cópia em um outro vetor, chamado *base*. Este método utiliza também dois outros vetores: *check* e *default*. O primeiro indica o estado correspondente a uma entrada em *base*. O vetor *default* contém a ação padrão para um dado estado i e um símbolo de entrada cujo símbolo terminal corresponde a um número j, caso $check[base[i] + j]$ seja diferente de i.

Para simular o acesso a uma entrada $Action[i, j]$ deve-se proceder da seguinte maneira: (i) se $base[i]$ não estiver vazia, ou seja, o estado i possui pelo menos uma entrada de empilhamento ou aceitação, então deve-se verificar se $check[base[i]+j] = i$. Se for, $Action[i, j] = value[base[i]+j]$. Caso contrário, $Action[i, j]$ refere-se a uma ação de erro; (ii) como $base[i]$ está vazia, utiliza-se então a ação padrão, armazenada em $default[i]$.

Este método preserva o tempo de acesso da tabela não compactada, mas poderá ser um pouco mais lento, uma vez que reduções desnecessárias são passíveis de ocorrer.

Para se obter uma boa taxa de compressão, é necessário que a distribuição não conflitante das linhas minimize o tamanho do vetor *value*, o que é um problema NP-completo [31]. Heurísticas para tratar esse problema são apresentadas por Ziegler [34] e Tarjan [31].

Método de compactação por coloração de grafos [28]: este método utiliza como estratégia de compactação o fato de que uma linha pode ser combinada com outra se ambas não possuírem entradas de não erro diferentes entre si para uma mesma coluna, o que caracteriza duas linhas não conflitantes. Este problema pode ser modelado por coloração de grafos da seguinte forma: (i) representa-se as linha da tabela *Action* como vértices do grafo; (ii) para cada linha i conflitante com uma linha k, cria-se uma aresta

não direcionada entre os vértices i e k; (iii) colore-se o grafo; (iv) combina-se as linhas cujos vértices no grafo possuem a mesma cor. Isto resulta em uma tabela $Action'$ com $g_r * n$ entradas, onde $g_r \leq m$ é o número de cores utilizadas para a coloração do grafo, n é o número de símbolos não-terminais e m é o número de linhas em $Action$. Neste ponto é necessário utilizar um vetor auxiliar, denominado *rowmap*. Este vetor mapeia uma linha da tabela $Action$ em uma linha correspondente em $Action'$. Desta forma, para cada linha i e k da tabela $Action$ combinadas em uma linha r em $Action'$, faz-se $rowmap[i] = rowmap[k] = r$.

Para obter um resultado de compressão ainda melhor, pode-se aplicar a mesma idéia de coloração às colunas de $Action'$, resultando em uma tabela

$$Action''[1\,..\,g_r, 1\,..\,g_c],$$

onde $g_c \leq n$ é o número de cores utilizado para a coloração referente à tabela $Action'$. Da mesma forma que na combinação de linhas, a combinação de colunas utiliza um vetor auxiliar, denominado *columnmap*, para registrar as combinações realizadas. Assim, para cada coluna j e p da tabela $Action'$ combinadas em uma coluna c em $Action''$, faz-se $columnmap[j] = columnmap[p] = c$.

Para simular o acesso a uma entrada $Action[i, j]$, tem-se: (i) a entrada $sigmap[i, j]$ é consultada para verificar se $Action[i, j]$ é referente a uma entrada de erro. Se for, uma ação sintática de erro é retornada. Caso contrário, retorna-se a ação sintática contida em $Action[rowmap[i], columnmap[j]]$.

Este método de compactação apresenta o mesmo tempo de acesso em relação à tabela original e não realiza nenhuma redução desnecessária. No entanto, a coloração ótima de um grafo é um problema NP-completo [9], sendo necessário a utilização de heurísticas.

Método de compactação por eliminação de linhas [13]: este método visa eliminar alternadamente linhas e colunas cujas entradas de não erro sejam iguais a um único valor. Este esquema utiliza quatro vetores (r, c, dr e dc) e duas matrizes (*value* e *sigmap*). Inicialmente, escanea-se a tabela $Action$ até que seja encontrada uma linha i que possua todas as suas entradas de não erro iguais a um único valor v. Faz-se $r[i] = v$ e $dr[i] = s$, onde s é um contador de eliminações realizadas. Em seguida verifica-se a possibilidade de eliminação de uma coluna, processo análogo à eliminação de linhas, com a diferença que os vetores c e dc são manipulados. O procedimento de eliminação alternada de linhas e colunas continua até que nenhuma linha possa ser eliminada. Para cada linha i e coluna j que não puderam ser eliminadas, faz-se $dr[i]$ e $dc[j]$ igual ao último escanemanto realizado (s_{max}) e $r[i]$ e $c[j]$ igual respectivamente à linha e coluna disponíveis em *value*.

Para simular o acesso a $Action[i, j]$, inicialmente o valor em $sigmap[i, j]$ é avaliado para se decidir se a entrada em questão é referente a uma entrada de erro. Se for, então uma ação de erro é retornada. Caso contrário, os valores em $dr[i]$ e $dc[j]$ são comparados. Três situações são possíveis: (i) $dr[i] < dc[j]$: a linha i foi eliminada antes da coluna j e portanto o valor de $Action[i, j] = r[i]$; $dr[i] > dc[j]$: a coluna j foi eliminada antes da linha i. Logo $Action[i, j] = c[j]$; $dr[i] = dc[j]$: a linha i e a coluna j não foram eliminadas. Portanto, $Action[i, j] = value[r[i], c[j]]$.

Este método preserva o tempo de acesso à tabela não compactada e não realiza nenhuma redução desnecessária.

Método de compactação por distância significativa [12]: neste método ignora-se as entradas de erro anteriores à primeira e posteriores à última entrada de não erro de uma dada linha da tabela *Action*. As demais entradas não ignoradas são inseridas seqüencialmente em um vetor denominado *value*, linha após linha. São utilizados também três outros vetores: *rowpointer*, *first* e *last*. Armazena-se em $first[i]$ o índice da coluna da primeira e em $last[i]$ o índice da coluna da última entrada diferente de erro da i-ésima linha da tabela *Action*. Uma entrada em $rowpointer[i]$ contém a posição em *value* que o elemento em $Action[i, 0]$ ocuparia, sendo em alguns casos armazenados índices negativos.

Para simular o acesso a uma entrada em *Action*, dado um estado i e um símbolo de entrada cujo símbolo terminal corresponde a um número j, a seguinte verificação é realizada: se $j < first[i]$ ou $j > last[i]$, então $Action[i, j]$ refere-se a uma entrada de erro; caso contário utiliza-se j como deslocamento a partir da posição armazenada em $rowpointer[i]$. Assim, tem-se que $Action[i, j] = value[rowpointer[i] + j]$.

Este método preserva o tempo de acesso à tabela não compactada e não introduz nenhum tempo adicional resultante de reduções desnecessárias. No entanto, a taxa de compressão é em muitos casos baixa, pois as entradas de erro existentes depois da primeira e anteriores à última entrada de não erro de uma linha não são eliminadas.

Método de compactação *row column* [32]: este esquema de compactação utiliza três vetores: *value*, *columnindex* e *rowpointer*. A compactação é realizada da seguinte maneira: a tabela *Action* é escaneada linha a linha, onde todas as entradas de não erro são seqüencialmente inseridas a partir da primeira posição livre em *value*. A posição inicial de inserção, dada uma linha i, é armazenada em $rowpointer[i]$. Para cada entrada em $Action[i, j]$ copiada para uma posição k em *value*, faz-se $columnindex[k] = j$. Com isto,

para simular o acesso a uma entrada $Action[i, j]$, percorre-se todas as posições de *columnindex* contidas no intervalo $[rowpointer[i] \,..\, rowpointer[i] - 1]$. Se for encontrada uma posição p tal que $columnindex[p] = j$, então $value[p]$ contém a ação desejada. Do contrário, $Action[i, j]$ é referente a uma entrada de erro, que é então retornada.

Por realizar uma busca linear na simulação de um acesso a *Action*, este método não preserva o tempo de acesso à tabela não compactada. Entretanto, tem a vantagem de não realizar nenhuma redução desnecessária e ainda eliminar todas as entradas de erro existentes. Neste aspecto, apresenta uma taxa de compactação superior à obtida pelo método de compactação por distância significativa.

Método de Compactação por Supressão de Zero [18]: este método funciona da seguinte forma: todas as linhas da tabela *Action* são escaneadas, de forma a copiar seqüencialmente as entradas de não erro em um vetor denominado *tablevector*. O número de entradas de erro antes da primeira ou depois da última entrada de não erro, ou entre duas entradas de não erro em uma dada linha da tabela, também é inserido nesse vetor. Neste último caso, cada uma dessas entradas são marcadas por um rótulo especial. Para se obter um desempenho melhor é aconselhável o uso do vetor *rowpointer*, o mesmo utilizado no método de compactação *Row Column*.

Para simular o acesso a uma entrada $Action[i, j]$, é necessário o uso de dois contadores: p e s. O primeiro é responsável por controlar o índice das entradas em *tablevector*, sendo inicializado com o valor em $rowpointer[i]$. O segundo visa controlar a coluna em acesso, o que delimita s ao intervalo $[1 \,..\, n]$. Supondo que as entradas rotuladas em *tablevector* contenham números inteiros positivos, a seguinte condição é realizada até que s se torne maior que j: se $s = j$ e $tablevector[p]$ não é um número, então $tablevector[p]$ contém a ação sintática desejada. Caso contrário, faz-se $s = s + tablevector[p]$ e $p = p + 1$. Quando s se torna maior que j, tem-se uma ação sintática de erro, que é então retornada.

Este método não preserva o tempo de acesso à tabela não compactada, pois realiza uma busca linear para simular o acesso a uma entrada em *Action*. No entanto, nenhuma redução desnecessária é realizada. Quando comparado ao método de compactação *row column*, esta técnica apresenta um desempenho melhor, pois a busca linear poderá não percorrer todas as entradas em $tablevector[rowpointer[i] \,..\, rowpointer[i + 1] - 1]$, caso em que $s > j$ e $p < rowpointer[i + 1] - 1$.

Os métodos de compactação apresentados serão analisados qualitativamente e quantitativamente de modo definir quais, ou qual a combinação deles, irão compor os níveis de compactação da ferramenta proposta.

5 Ferramenta Proposta

A ferramenta proposta a ser desenvolvida na dissertação de mestrado visa disponibilizar a projetistas de compiladores um ambiente visual que permita um desenvolvimento mais intuitivo e produtivo de analisadores sintáticos.

Espera-se que a ferramenta atenda aos seguintes requisitos:

- possibilite a integração com qualquer analisador sintático LR, via o uso de *plugins*;
- seja capaz de executar em várias plataforma, tais como Linux, MacOs e Windows;
- apresente, a partir da especificação de uma gramática em uma linguagem específica de um gerador de analisador sintático, a máquina de estados correspondente, indicando visualmente os pontos de ocorrência de conflitos;
- possibilite ao projetista corrigir a especificação diretamente na máquina de estados;
- gere a saída na mesma linguagem em que o gerador de analisador utilizado geraria. Por exemplo, se o Yacc [24] for utilizado, então o algoritmo de saída será codificado em C. No caso de o projetista optar em utilizar o CUP[2], o algoritmo de saída será codificado em Java;
- otimize as tabelas sintáticas geradas pelos geradores de analisadores sintáticos independentemente do gerador utilizado, com flexibilidade de escolha do nível de compactação realizado;
- anime a execução passo a passo do algoritmo de análise sintática para uma dada entrada, onde o projetista poderá visualizar o estado das estruturas de dados envolvidas, tais como:
  - o símbolo corrente na *string* de entrada;
  - o estado atual e o respectivo conjunto de itens;
  - a árvore de derivação obtida até o momento;
  - os valores semânticos de cada nodo árvore de derivação;

  - as ações tomadas a cada iteração (desempilhamento/empilhamento de estados da pilha, redução através de uma produção específica, aceitação, sinalização de erro, etc).

- disponibilize uma linguagem capaz de mapear as construções sintáticas dos geradores de analisadores sintáticos em uma sintaxe única, facilitando a integração de diferentes ferramentas.

A ferramenta será escrita em Java, utilizando os componentes *Swing* para a criação da GUI. Para algumas partes mais críticas, como a compactação de tabelas, a implementação será em ANSI-C++. Estes módulos serão integrados utilizando-se a interface nativa de Java (*Java Native Interface*)[4], garantindo assim, a possibilidade de execução em várias plataformas.

5.1 Validação da Solução

Para validar a ferramenta, a mesma será utilizada na implementação de um analisador sintático para a linguagem Machĭna [17].

Desta experiência serão reportados os pontos em que a ferramenta se mostrou eficiente na solução dos problemas ocorridos a partir da especificação da linguagem, assim como as dificuldades encontradas.

6 Detalhamento das contribuições

As contribuições da dissertação de mestrado proposta são:

- análise e avaliação dos métodos de compactação de tabelas. Para isto serão realizados testes com os 9 métodos de compactação apresentados na Seção 4.1 de modo a obter resultados empíricos. As linguagens a serem utilizadas nesses testes serão as linguagens de marcação XML e XHTML. Isto é justificado pelo fato da compactação de tabelas ser extremamente relevante em dispositivos com certa limitação de memória, como PDAs e celulares. Assim, analisadores sintáticos de *microbrowsers* disponíveis em tais aparelhos poderão ter um melhor aproveitamento de memória sem a perda significativa de desempenho. A partir dos resultados obtidos serão elegidos os métodos, ou a combinação deles, para compor cada nível de compactação a ser fornecido pela ferramenta. Ressalta-se que estes métodos serão aplicados nas tabelas geradas pelos geradores de analisadores sintáticos de tal forma que as mesmas sejam otimizadas;

- construção de uma ferramenta visual para auxílio no desenvolvimento de um analisador sintático. Esta ferramenta se diferencia das atualmente existentes pelos seguintes motivos:

  - reune as principais funcionalidades disponíveis em outras ferramentas [33, 16, 26, 25, 8, 6], tais como mecanismos visuais para correção da especificação e animação da execução do algoritmo de análise sintática com a apresentação dos estados de cada estrutura de dados envolvida;
  - integração com qualquer analisador sintático LR;
  - compactação da tabela sintática dos analisadores gerados com a flexibilidade de escolha do nível de compactação a ser realizado. Essa possibilidade de escolha dá ao projetista do analisador sintático uma grande flexibilidade, pois permite gerar analisadores conforme o perfil dos computadores que irão executá-los.

- criação de um arcabouço que disponibilize a desenvolvedores um conjunto de classes e interfaces responsáveis por definir o protocolo de integração entre geradores de analisadores sintáticos LR e a ferramenta em questão, garantindo uma integração não intrusiva entre elas. Isto possibilita ao usuário uma flexibilidade na escolha do gerador de analisador sintático a ser utilizado;

- projeto e implementação de uma linguagem de interface com o objetivo de unificar as diferentes sintaxes dos geradores de analisadores sintáticos existentes.

7 Cronograma

1. Fevereiro:

   (a) Estudo dos geradores Yacc [24], Bison [19], SableCC [21], AntLR [1] e CUP [2]

   (b) Escrita do texto da dissertação relativa ao assunto pesquisado

2. Março:

   (a) Estudo dos geradores Bison e SableCC

   (b) Estudo das ferramentas VisualParse++[8] e ProGrammar[6]

   (c) Escrita do texto da dissertação relativa ao assunto pesquisado

3. Abril:

   (a) Estudo do *framework* Tango[30]
   (b) Estudo de padrões de projeto[22]
   (c) Especificação e projeto do arcabouço para desenvolvimento de *plugins*
   (d) Escrita do texto da dissertação relativa ao assunto pesquisado

4. Maio:

   (a) Estudo de padrões de projeto
   (b) Especificação e projeto do arcabouço para desenvolvimento de *plugins*
   (c) Escrita do texto da dissertação relativa ao assunto pesquisado

5. Junho:

   (a) Estudo de padrões de projeto
   (b) Criação dos *plugins* para o Bison e SableCC
   (c) Estudo do *toolkit* Swing[7]
   (d) Codificação da ferramenta
   (e) Finalização do texto da dissertação relativo à revisão bibliográfica realizada

6. Julho:

   (a) Estudo do *toolkit* Swing
   (b) Codificação da ferramenta

7. Agosto:

   (a) Estudo do *toolkit* Swing
   (b) Codificação da ferramenta

8. Setembro:

   (a) Estudo do *toolkit* Swing
   (b) Codificação da ferramenta

(c) Estudo dos métodos de compactação de tabelas

(d) Codificação e testes dos métodos estudados

(e) Escrita da dissertação

9. Outubro:

(a) Estudo sobre JNI[4]

(b) Incorporção de compressão de tabelas à ferramenta via JNI

(c) Escrita da dissertação

10. Novembro:

(a) Validação da ferramenta: escrita do analisador sintático para Machĭna [17]

(b) Escrita da dissertação

11. Dezembro:

(a) Escrita da dissertação

(b) Defesa

8 Sumário da dissertação

Capítulo 1: Introdução

1.1 Definição do Problema

1.2 Solução Proposta

1.3 Contribuições

1.4 Organização da Dissertação

Capítulo 2: Análise Sintática

2.1 Introdução

2.2 Geradores de Analisadores Sintáticos

2.2.1 Yacc

2.2.2 Bison

2.2.3 SableCC

2.2.4 AntLR

9 Conclusão

Neste documento foi apresentada uma proposta de projeto e construção de uma ferramenta visual para auxiliar o desenvolvimento de analisadores sintáticos. A ferramenta descrita agrega as principais funcionalidades das ferramentas atualmente existentes, com a possibilidade de integração com qualquer gerador de analisador sintático LR. Além disso, a ferramenta visa também a geração de analisadores sintáticos cujas as tabelas sintáticas sejam compactas, com a flexibilidade de escolha do grau de compactação das mesmas.

Referências

[1] Antlr parser generator. Último acesso: 20/02/2006.

[2] Cup: Lalr parser generator in java. `http://www2.cs.tum.edu/projects/cup/`. Último acesso: 23/12/2005.

[3] Gcc: Gnu c compiler.

[4] Java native interface. `http://java.sun.com/j2se/1.5.0/docs/guide/jni/`. Último acesso: 03/01/2006.

[5] Javacc: The java parser generator. Último acesso: 02/01/2006.

[6] Programmar. `http://www.programmar.com`. Último acesso: 27/12/2005.

[7] The swing connection. `http://java.sun.com/products/jfc/tsc/`. Último acesso: 06/01/2006.

[8] Visual parse++. `http://www.sand-stone.com/Visual\%20Parse++.htm`. Último acesso: 23/12/2005.

[9] *Introduction to algorithms*. MIT Press, 2001.

[10] A.V. Aho, R. Sethi, and J.D Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[11] A.M.M. Al-Hussaini and R.G. Stone. Yet another storage technique for lr parsing tables. *Softw. Pract. Exper.*, 16(4):389–401, 1986.

[12] B. Beach. Storage organization for a type of sparce matrix. In *5th Southeastern Conference on Combinatorics , Graph Theory and Computing*, pages 245–252, 1974.

[13] J.R Bell. A compression method for compiler precedence tables. In *IFIP Congress*, pages 359–362, 1974.

[14] M.A.S Bigonha. Class notes in compiler topics, 2005.

[15] R.S Bigonha and M.A.S Bigonha. A method for efficient compactation of lalr(1) parsing tables. Technical report, Departament of Computer Science, Federal University of Minas Gerais.

[16] S.A. Blythe, M.C. James, and S.H. Rodger. Llparse and lrparse: visual and interactive tools for parsing. In *SIGCSE '94: Proceedings of the twenty-fifth SIGCSE symposium on Computer science education*, pages 208–212. ACM Press, 1994.

[17] A Linguagem de Especificação Formal Machĭna 2.0. Bigonha, r.s and tirelo, f. and iorio, v.o and bigonha, m.a.s. Technical report, Departament of Computer Science, Federal University of Minas Gerais, 2006.

[18] P. Dencker, Durre, K., and H. Heuft. Optimization of parser tables for portable compilers. *ACM Trans. Program. Lang. Syst.*, 6(4):546–572, 1984.

[19] C Donnely and R.M. Stallman. Bison - the yacc-compatible parser generator.

[20] M. Fayad and D.C. Schmidt. Object-oriented application frameworks. *Commun. ACM*, 40(10), 1997.

[21] E.M G and L.J. Hendren. Sablecc, an object-oriented compiler framework. In *TOOLS '98: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 140. IEEE Computer Society, 1998.

[22] E. Gamma, R. Helm, R. Johnson, and Vlissides. J. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[23] R.E. Johnson. Components, frameworks, patterns. In *SSR '97: Proceedings of the 1997 symposium on Software reusability*, pages 10–17. ACM Press, 1997.

[24] S.C Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, 1979.

[25] A. Kaplan and Shoup. D. Cupv: a visualization tool for generated parsers. In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 11–15. ACM Press, 2000.

[26] S. Khuri and Y. Sugono. Animating parsing algorithms. In *SIGCSE '98: Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education*, pages 232–236. ACM Press, 1998.

[27] H.A. Schmid. Systematic framework design by generalization. *Commun. ACM*, 40(10):48–51, 1997.

[28] A Schmitt. Minimizing storage space of sparse matrices by graph coloring algorithms. *In Graphs, Data Structures, Algorithms*, pages 157–168, 1979.

[29] D. Shoup. Visualizing lalr generated parsers. masters project report. Technical report, Departament of Computer Science, Clemson University, 1999.

[30] S.T. Stasko. Tango: A framework and system for algorithm animation. *SIGCHI Bull.*, 21(3):59–60, 1990.

[31] R.E. Tarjan and A.C. Yaho. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979.

[32] R. Tewarson. Row column permutation of sparce matrices. *Computer Journal*, pages 300–305, 1967/1968.

[33] S.R. Vegdahl. Using visualization tools to teach compiler design. In *Proceedings of the second annual CCSC on Computing in Small Colleges Northwestern conference*, pages 72–83. Consortium for Computing Sciences in Colleges, 2000.

[34] S. F Ziegler. Smaller faster driven parser. Unpublished manuscript, 1977.