SIC: FERRAMENTA PARA IMPLEMENTAÇÃO DE LINGUAGENS

AUTORES:
MARIZA ANDRADE DA SILVA BIGONHA
ROBERTO DA SILVA BIGONHA

ENDEREÇO:
Departamento de Ciência da Computação
ICEx - UFMG - Caixa Postal 702
CEP 30161 - BELO HORIZONTE - MG
Tel.: (031) 443-4088

CURRICULUM VITAE:
MARIZA ANDRADE DA SILVA BIGONHA, Mestre em Ciência da Computação (UFMG, 1985). Analista do DCC/UFMG. Áreas de interesse: Linguagens e Compiladores.
ROBERTO DA SILVA BIGONHA, Ph.D. em Ciência da Computação (UCLA - USA, 1981). Professor do DCC/UFMG. Áreas de interesse: Linguagens e Compiladores.

RESUMO:
SIC é uma ferramenta destinada a assistir a implementação de linguagens de programação. A partir da especificação da sintaxe e de rotinas semânticas, SIC gera compiladores baseados em reconhecedores LALR(1) dotados de recuperação automática de erro, editor de programas acoplado e gerador de código dirigido pela sintaxe.

PALAVRAS-CHAVE:
Sistema de Implementação de Compiladores, Analisador LALR(1), Recuperação de Erro, SIC.

1. INTRODUÇÃO

A tarefa de implementar linguagens é, em geral, bastante complexa e cheia de detalhes. Por exemplo, para a implementação sem auxílio de ferramentas especializadas de um compilador para uma linguagem do tipo PASCAL são necessários mais de quatro homens-ano. Por outro lado, pesquisas e experiências com compiladores trouxeram um melhor entendimento dos processos de compilação e dos fundamentos teóricos das técnicas envolvidas. Em particular, a identificação e formalização de várias fases comuns a compiladores, como por exemplo análise léxica, análise sintática e geração de código, tem permitido a mecanização dos processos de compilação.

A idéia de se usar ferramentas para automatizar, pelo menos parcialmente, a implementação de linguagens é bastante antiga. O que diferencia os novos sistemas dos mais antigos são os avanços tecnológicos incorporados. Existe atualmente um grande número de sistemas de suporte à construção de compiladores implementados nos mais diferentes computadores. Estas ferramentas podem variar desde simples programas destinados a auxiliar em tarefas bastante específicas, como por exemplo consistir gramáticas, até grandes sistemas capazes de gerar compiladores a partir das definições de sintaxe e semântica da linguagem fonte. Um importante sistema auxiliar para implementação de compiladores é o YACC [16], que foi desenvolvido por S.C. Johnson da Bell Laboratories. O YACC roda sob o sistema UNIX e produz compiladores na linguagem C.

Os métodos de análise sintática utilizados por sistemas de implementação de compiladores são, em geral, baseados em gramáticas livres-do-contexto. Estes métodos se enquadram em duas grandes categorias: os ascendentes e os descendentes. Em um método ascendente o reconhecedor sintático constrói a árvore de derivação do programa, de baixo para cima e, normalmente, da esquerda para a direita. Se a raiz da árvore produzida para todo o fonte corresponder ao símbolo de partida da gramática, o programa está sintaticamente correto. Nos métodos descendentes, o processo inverso é utilizado.

Dentre os métodos ascendentes encontra-se a família dos métodos LR(k) [1,2,3,18] originalmente desenvolvidos pelo Professor Donald E. Knuth [17]. Estes métodos têm sido considerados superiores aos demais em uso para construção de compiladores devido principalmente a sua abrangência, sendo aplicáveis à grande classe das linguagens livres-do-contexto determinísticas, englobando, portanto, a classe das linguagens de programação de interesse prático. Os sistemas mais modernos, por exemplo o YACC, usam o método LALR(1), uma variante mais eficiente do LR(k).

A geração de código nestes sistemas segue um esquema de tradução dirigida por sintaxe, no qual rotinas ou ações semânticas são associadas às produções da gramática. Estas rotinas são ativadas pelo reconhecedor sintático sempre que houver reduções [3] envolvendo as produções associadas. As rotinas semânticas, além de gerarem código, também servem para administrar os atributos sintetizados [1,2,3] dos nodos da árvore de derivação que é construída pelo reconhecedor sintático. Estes atributos, que são associados a símbolos da gramática, são estruturas de dados que servem para propagar informações de baixo para cima na árvore de derivação, para uso na geração de código. Em geral, por razões de eficiência a árvore de derivação nunca é efetivamente construída: no

seu lugar usa-se uma pilha para armazenar somente os nodos (com seus atributos) que ainda não tiveram sua raiz determinada. À medida que a análise sintática avança, nodos mais necessários são descartados.

2. CARACTERÍSTICAS GERAIS DO SIC

O SIC, Sistema de Implementação de Compiladores, é uma ferramenta de suporte à implementação de linguagens, através da automatização de alguns dos processos de compilação. O SIC é essencialmente um compilador de uma linguagem de nível mais alto que PASCAL [15]. Esta linguagem, que chamamos linguagem SIC, é um PASCAL estendido de forma a prover facilidades para implementação de compiladores. Em especial, a linguagem SIC possui os seguintes recursos:

a) Permite o uso de todas as facilidades do PASCAL.
b) Permite fácil expressão de rotinas semânticas com referências diretas aos símbolos da gramática e aos seus atributos.
c) Permite declaração e uso de atributos sintetizados em termos da gramática, com total abstração da árvore de derivação e da pilha de atributos.
d) Aceita descrição da gramática em notação próxima à BNF.
e) Permite associar "tokens" ou unidades sintáticas a seus tipos, de forma a estabelecer uma convenção para comunicação entre o analisador léxico definido pelo usuário e o sintático gerado pelo sistema.

O SIC possui ainda as seguintes facilidades:

a) Gera compiladores em PASCAL, isto é, traduz programas em SIC para PASCAL.
b) Gera tabelas LALR(1) compactadas.
c) Produz, por opção do usuário, reconhecedor LALR(1) com recuperação automática de erro de sintaxe.
d) Produz, por opção do usuário, reconhecedor LALR(1) interativo acoplado a editor de programa com ou sem recuperação automática de erro de sintaxe.
e) Consiste gramática quanto a símbolos indefinidos e inúteis, ou seja, verifica se a gramática é reduzida[1].
f) Produz listagens diversas: do fonte, da gramática, de símbolos gramaticais indefinidos ou inúteis, de referências cruzadas da gramática, de tabela LALR(1), da máquina LR(0) [2,3] e de outras informações sobre o reconhecedor sintático gerado.

O SIC foi implementado em PASCAL em microcomputadores de 16 bits baseados no microprocessador INTEL 8088 e sob sistema operacional compatível com o SISNE da SCOPUS. Requer para execução um mínimo de 265KB de memória e uma unidade de disquete de 360KB. Espaço adicional em disquete pode ser necessário dependendo porte da aplicação. Com uma memória interna de 256KB, é possível gerar um reconhecedor LALR(1) de cerca de 800 estados para a linguagem ADA.

O projeto e desenvolvimento do SIC iniciaram-se em 1983, tendo sua versão 1.0 sido concluída em 1985, com a produção de uma tese de mestrado. Os resultados dos trabalhos desta primeira fase foram divulgados em [4,5,6,7,8]. A partir de então novas facilidades foram concebidas e estão sendo incorporadas ao sistema, que também teve sua interface com o usuário completamente reprojetada. Neste trabalho apresentamos o SIC em sua versão mais recente, a versão 2.2.

3. A LINGUAGEM SIC

Um programa em SIC possui um cabeçalho seguido de várias seções e termina com palavra-chave %%END. Estas seções podem, em princípio, ocorrer em qualquer ordem, devendo apenas respeitar a regra de que a declaração deve sempre preceder ao uso. Toda seção inicia-se com uma palavra-chave própria e se estende até o início da próxima seção. As palavras-chave do SIC iniciam-se por %%. As seções do SIC têm as seguintes funções:

a) Especificação do tipo de compilador.
b) Especificação do número de passos.
c) Definição dos terminais (tokens).
d) Declaração dos atributos dos símbolos.
e) Declarações de rótulos, constantes, variáveis, procedimentos e funções.
f) Definição do mapa de escopo.
g) Definição de não-terminais de recuperação de erro.
h) Especificação da gramática e declaração das rotinas semânticas.
i) Resolução de conflitos do analisador sintático.
j) Corpo principal do compilador.

3.1 Cabeçalho

O cabeçalho, que é iniciado pela palavra-chave %%COMPILER, especifica se o compilador desejado é programa principal ou procedimento. Exemplos:

a) Para gerar compilador programa principal:
```
%%COMPILER program MODULA2:
. . . .
%%END.
```
b) para gerar compilador procedimento:
```
%%COMPILER procedure MODULA2:
. . . .
%%END.
```

3.2 Tipo de Compilador

Esta seção serve para especificar o tipo de processamento a ser usado pelo compilador gerado. As opções são:

%%INTERACTIVE NOREC
Indica que deve-se gerar complicadores interativos com editor de programa acoplado e sem mecanismo para recuperação automática de erro de sintaxe.

%%INTERACTIVE REC
Indica que deve-se gerar compiladores interativos com editor de programas acoplados e recuperador de erro de sintaxe.

%%BATCH
Indica que deve-se gerar compilador "batch" com recuperador automático de erro de sintaxe.

%%INCREMENTAL
Indica que deve-se gerar compilador incremental interativo com editor de programa.

3.3 Número de Passos

Esta seção é usada para indicar o número de pas-

sos do compilador a ser gerado e identificar cada passo:

%%FIRST PASS
Indica que o compilador será de vários passos e que este é o seu primeiro passo.

%%OTHER PASS
Indica os demais passos do compilador.

Se esta seção for omitida, o sistema supõe que o compilador desejado é o de um único passo. Para cada tipo de compilador existe um analisador sintático particular. Estes analisadores são procedimentos PASCAL lidos de arquivos pelo SIC no momento apropriado e incorporados ao compilador gerado.

3.4 Definição dos Terminais

A finalidade desta seção, que se inicia com a palavra-chave %%TOKENS, é estabelecer uma convenção de comunicação entre os analisadores sintáticos gerados pelo SIC e o analisador léxico YYSCAN, que deve ser escrito diretamente pelo usuário. Este procedimento é chamado pelos analisadores sintáticos do compilador gerado sempre que o próximo "token" ou símbolo do programa fonte for necessário para se continuar a compilação. Para cada "token" reconhecido na fonte, YYSCAB deve retornar o seu tipo e, quando for o caso, informações adicionais sobre ele, isto é, valores de seus atributos. Além disto, é necessário que o tipo de cada "token" seja conhecido pelos analisadores sintáticos de forma que se possa estabelecer a correspondência entre cada "token" que aparece na gramática e o tipo correspondente que YYSCAN retorna. A presente seção cumpre exatamente este pepal, estabelecendo um mapeamento entre "tokens" e tipos.

Exemplo:

```
%%TOKENS

      "id"     = tipodeId;

      "const"  = tipodeconst;

      "<"      = menor;

      "<"      = maior;

      "eof"    = YYEOF;
```

Os símbolos à esquerda do sinal de igual são terminais de gramática e os identificadores à direita são nomes de constantes inteiras, cujas definições serão geradas automaticamente pelo SIC, e que representam o tipo do símbolo associado. Não há restrições quanto aos identificadores que podem ser usados, exceto pela presença obrigatória do tipo reservado YYEOF, que designa a marca de fim de arquivo de programa fonte.

O procedimento YYSCAN retorna o tipo de cada "token" lido através da variável inteira global YYSIMB pré-declarada pelo SIC. O retorno de informações adicionais sobre o "token" é feito por atribuição direta a atributos do "token", conforme é mostrado na próxima seção.

3.5 Declaração dos Atributos

Esta seção, que se inicia com a palavra-chave %%STACK, serve para declarar os atributos sintetizados dos símbolos terminais e não-terminais da gramática. Cada atributo pode ser visto como um "record" do PASCAL, que é associado ao nodo correspondente ao símbolo na árvore de derivação. A partir da declaração de atributos, O SIC cria no compilador uma pilha em que cada elemento pode conter qualquer um dos atributos declarados. Esta pilha é usada para armazenar os nodos da árvore de derivação que ainda não foram inteiramente processados, ou seja, guarda a porção da árvore de derivação que ainda é necessária ao processamento.

Exemplo:

```
%%STACK 200 OF ATTRIBUTES

     expr    = (ender : integer; modo : TMODO);

     "id"    = (valor : TVALOR)  {"id".valor := BRANCO}

     "const" = (valor : integer) {"const".valor := 0  }
```

A constante inteira 200 especifica tamanho máximo da pilha. Para cada símibolo da gramática que tenha atributos deve-se especificar o seu nome, o sinal igual e, entre parênteses, os seus atributos com os respectivos tipos. O tipo de um atributo deve obrigatoriamente ser um identificador.

Os atributos de uma particular instância de símbolos terminais são definidos pelas rotinas semânticas ou, mais comumente, pelo procedimento YYSCAN por atribuições da forma:

"id". valor: = stringlido:

O SIC cuida para que o valor do atributo acima seja instalado no nodo correspondente ao "id" na árvore de derivação, quando o analisador sintático for ativado e o "token" acima for encontrado no fonte.

Os textos entre chaves que ocorrem em algumas das declarações no exemplo acima são opcionais e denotam ações de inicialização. Estas ações somente são executadas quando os símbolos correspondentes forem inseridos no programa fonte como resultado de uma ação de recuperação de erro de sintaxe. Este recurso permite assegurar que a pilha de atributos só contém valores bem definidos.

3.6 Declarações

Esta seção subdivide-se em subseções de declarações de rótulos, constantes, variáveis e procedimentos (e funções) do PASCAL. Cada uma destas subseções se inicia com uma palavra-chave própria: %%LABELS para rótulos, %%CONSTANTS para definições de constantes, %%VARIABLES para declarações de variáveis, %%TYPES para definições de tipos e %%PROCEDURES para declarações de funções e procedimentos do PASCAL.

Exemplos:

```
%%LABEL 1000,9999

%%CONSTANTS

      MAXID =   8;

      MAXTS = 500;

%%TYPES
```

```
    TMODO   = (EXPRCONST,EXPRVAR);
    TVALOR  = array[1..MAXID] of char;
%%VARIABLES
    X: integer;
    y: real;
```

3.7 Mapa de Escopo

Esta seção, que se inicia pela palavra-chave %%SCOPEMAP, tem por finalidade especificar os delimitadores de construções aninhadas da linguagem fonte, tais como procedimentos, blocos, expressões parentizadas etc. O conhecimento destes delimitadores possibilita a implementação de um sofisticado mecanismo de recuperação automática de erro de sintaxe [4,7,12,13]. Esta seção é inteiramente opcional. Exemplo:

```
%%SCOPEMAP
    "("      : ")" ;
    "BEGIN"  : "END" ;
    "WHILE"  : "DO" ;
```

3.8 Não-Terminais de Recuperação de Erro

Esta seção, que se inicia com a palavra-chave %%NTMAP, permite relacionar os símbolos não-terminais da gramática que poderão ser usados durante a fase de recuperação de erro de sintaxe como candidatos à inserção ou substituição de um outro símbolo. Se esta seção for omitida somente terminais poderão ser usados, para insetir ou substituir símbolos. Exemplo:

%%NTMAP expr, decl, cmd;

3.9 Gramática e Rotinas Semânticas

Esta seção, que se inicia com a palavra-chave %%GRAMMAR, serve para especicar as produções da gramática da linguagem fonte e as rotinas semânticas e elas associadas. A partir da gramática, o SIC produz as tabelas que dirigem os algoritmos dos analisadores sintáticos LALR(1). As rotinas semânticas, por sua vez, são coletadas, traduzidas para PASCAL e incorporadas ao compilador gerado para serem ativadas sempre que a produção associada for usada em uma redução durante a análise sintática. As rotinas semânticas, que são codificadas entre chaves após cada produção, são seqüências de comandos em PASCAL, nos quais são permitidas referências qualificadas aos atributos dos símbolos que ocorrem na produção associada. As referências ambíguas a símbolos que ocorrem mais de uma vez na produção são resolvidas via indexação, que distingue a ocorrência desejada do símbolo. A I-ésima ocorrência de um símbolo em uma produção deve ser indexada pelo inteiro > 0. Exemplo:

```
%%GRAMMAR PROG AND SEMANTICS
    PROG = expr ;
    expr = expr "+" expr
        {expr.ender := GeraTemporario;
         Gen('ADD',expr[1].ender,expr[2].ender,expr[3].ender);
         if (expr[2].modo = EXPRCONST) and
            (expr[3].modo = EXPRCONST)
         then expr.modo := EXPCONST
         else expr.modo := EXPRVAR; };
```

No exemplo acima, expr.ender, expr[1].ender, expr.modo designam os atributos ender e modo do símbolo expr que está do lado esquerdo da produção correspondente; expr[2].ender e expr[2].modo denotam os atributos de segundo expr; expr[3].ender e expr[3].modo referem-se aos atributos da terceira ocorrência de expr na produção. Note que expr.ender é equivalente a expr[1].ender.

3.10 Resolução de Conflitos

Gramáticas livres-do-contexto são muito úteis para definir com precisão a sintaxe de linguagens de programação e para geração eficiente de reconhecedores sintáticos determinísticos, por exemplo do tipo LALR(1). Existem, contudo, situações em que construções de linguagens de programação seriam especificadas de forma mais natural e compacta se gramáticas ambíguas pudessem ser usadas. Entretanto, o método LALR(1) indica necessariamente conflitos de ação quando a gramática é ambígua e não existem meios de se resolver estes conflitos somente a partir das informações contidas na gramática. Por outro lado, certos conflitos de ação podem ser resolvidos diretamente pelo usuário que dispõe de informações adicionais, como por exemplo conhecimento do contexto e prioridade de operadores.

O SIC permite que dados sobre prioridade e associatividade de operadores binários e ordem de certas ações sejam passados ao gerador LALR(1) de forma a ser possível utilizar gramáticas ambíguas. Assim, sempre que conflitos forem detectados, o gerador LALR(1) usa estas informações para resolvê-los, dando preferência a ações com maior prioridade. A prioridade de uma ação de empilhamento [2,3] é dada pela precedência e associatividade do símbolo lido, se este tiver estes dados específicos. A precedência de uma ação de redução [2,3] é a do terminal mais à direita na produção correspondente.

A declaração de precedência e associatividade é feita através de cláusulas começando com as palavras-chave %%RIGHT, para operadores que se associam à direita, %%LEFT, para aqueles que se associam à esquerda e %NONE, quando não é possível a associação. Na seção de resolução de conflitos, que é identificada pela palavra-chave %%CONFLICTS, estas cláusulas devem ser especificadas em ordem crescente de precedência.

Exemplo:

```
%%CONFLICTS
    %%NONE   "<", ">", "=" ;
```

```
%%LEFT  "+", "-";
%%LEFT  "*", "/";
%%RIGHT "**" ;
```

3.11 Corpo Principal

Esta seção, que se inicia com a palavra-chave %%PROGRAM, define o corpo principal compilador. Este corpo é uma lista de comandos PASCAL, contendo necessariamente uma ativação do procedimento YYPARSER pré-definido pelo SIC e que representa a chamada ao analisador sintático.

4. A TABELA LALR(1)

Um reconhecedor LALR(1) é constituído por um algoritmo básico, uma pilha de estados e uma tabela. O algoritmo é fixo e cada gramática tem sua própria tabela. Esta tabela, na sua forma original, é uma matriz retangular, onde os índices de linha representam estados possíveis do reconhecedor e os índices de coluna, símbolos gramaticais. Para o PASCAL, a tabela LALR(1) possui cerca de 300 × 100 entradas. Cada entrada representa uma possível ação do analisador sintático, a saber: ação de empilhamento, ação de redução, fim de análise e erro de sintaxe.

Gerar um reconhecedor LALR(1) consiste em produzir a tabela a partir das gramática. Para isto, o SIC utiliza o algoritmo proposto por Kristensen [18] estendido para tratar gramáticas ambíguas. Neste algoritmo, a resolução dos conflitos de ação é primeiro tentada com o exame da máquina LR(0) [2,3,18] e se os conflitos perdurarem, o SIC usa as informações sobre prioridades especificadas pelo usuário na seção de resolução de conflitos.

A tabela LALR(1) em sua representação original de matriz retangular ocupa muito espaço. Na prática, entretanto, a maior parte das entradas são ações de erro, o que permite a aplicação de métodos eficientes de armazenamento. O método do SIC para compactação de tabelas LALR(1), que é escrito em detalhes em [5], usa propriedades intrínsecas dos reconhecedores LALR(1) e elimina da representação interna a maioria das ações de erro e outras que podem de alguma forma ser recuperadas da tabela. O ganho em economia de espaço é substancial, atingindo-se uma porcentagem de redução no tamanho original da ordem de 96%. Para exemplificar, a tabela compactada LALR(1) para a ADA ocupa cerca de 12KB.

5. O EDITOR DE PROGRAMA

Um dos reconhecedores sintáticos gerados pelo SIC possui editor de programa acoplado. Isto permite a geração de compiladores interativos do tipo TURBO PASCAL [10], que ao detectar um erro de sintaxe posiciona-se automaticamente no ponto do erro dentro do programa fonte, em modo de edição. O editor de programas do SIC foi implementado por Eduardo Costa e Silva [14], a partir das ferramentas do TURBO EDITOR TOOLBOX da Borland [11].

6. COMPILAÇÃO INCREMENTAL

A opção de geração de compilador incremental, que ainda não está operacional no SIC, tem como proposta a produção de compiladores que ao efetuar recompilações de um dado programa fonte, aproveitem o máximo do trabalho realizado em compilações anteriores, de forma a aumentar o desempenho. Detalhes sobre esta abordagem podem ser encontrados em [9]. O desenvolvimento do um analisador sintático LALR(1) incremental faz parte do trabalho de tese de mestrado de Rosilene Terezinha Martins[19].

7. RECUPERAÇÃO DE ERRO

O método de recuperação automática de erro sintaxe utilizado pelo SIC é baseado na proposta de BURKE & FISHER [12,13]. Neste método, considera-se como a melhor ação de recuperação de erro aquela que permite ao compilador mais avançar no fluxo de entrada. Esta regra é implementada por meio de três estratégias [4,7]: inicialmente, experimenta-se inserir, remover ou substituir o símbolo do ponto do erro ou algum outro à sua esquerda. Em segundo lugar, tenta-se fechar o escopo mais próximo, a partir das informações obtidas da seção de mapas de escopo (Vide 3.7) e finalmente, a terceira estratégia consiste em remover trechos da entrada após o erro em conjunção com inserção, remoção ou substituição de símbolos à esquerda do ponto do erro. Após todas estas tentativas, o sistema seleciona como candidatos as correções que possibilitam um maior avanço do analisador depois de passado o ponto do erro. Um dos candidatos é efetivamente usado pelo analisador sintático e os demais relatados ao usuário para facilitar a identificação do erro.

8. COMPARAÇÃO COM OUTROS SISTEMAS

O sistema de implementação de compiladores que mais se aproxima do SIC é o YACC [16]. O SIC é mais poderoso que o YACC em vários aspectos:

1) O mecanismo de recuperação de erro de sintaxe do SIC é mais transparente e produz melhores mensagens de erro.
2) O leque de opções do SIC é muito mais amplo, permitindo a geração de vários tipos de compiladores, como por exemplo compilador interativo.
3) O SIC opera em sistemas operacionais compatíveis com o SISNE da SCOPUS, o que, momentaneamente, é uma vantagem em vista da atual maior difusão no Brasil destes sistemas em relação ao UNIX e seus compatíveis.
4) O método de compactação de tabelas do SIC é mais eficaz.

O YACC gera compiladores na linguagem C, enquanto que o SIC os produz em PASCAL. Até o presente, compiladores de C têm sido implementados de forma mais eficiente, produzindo códigos de melhor qualidade que os de PASCAL. Isto sem dúvida, torna o YACC mais vantajoso quando a eficiência do compilador gerado estiver em primeiro plano.

9. CONCLUSÕES

Um sistema auxiliar para implementação de compiladores em ambientes compatíveis com o SISNE da

SCOPUS foi apresentado. As principais contribuições geradas pelo desenvolvimento do SIC foram a automatização de vários aspectos do processo de compilação, a produção de uma ferramenta de suporte, a implementação de linguagens e a proposta de novas técnicas de implementação de linguagens e de sistema de implementação de compiladores.

10. AGRADECIMENTOS

Agradecemos ao Conselho Nacional de Pesquisa (CNPQ) pelo suporte financeiro concedido através de bolsa de estudos de Mestrado e de pesquisa.

11. REFERÊNCIAS BIBLIOGRÁFICAS

1. Aho, A.V. & Ullman, J.D., *The Theory of Parsing, Translation, and Compliling,* Volume I: Parsing, Prentice-Hall Inc, 1972.
2. Aho, A.V. & Ullman, J.D., *Principles of Compilier Design,* Addison-Wesley Publishing Company, 1977.
3. Aho, A.V., Sethi, R. & Ullman, J.D., *Compilers — Principles, Techniques, and Tools,* Addison-Wesley Publishing Co, 1986.
4. Bigonha, M.A.S., "SIC: Sistema de Implementação de Compiladores", Tese de Mestrado — UFMG (1985).
5. Bigonha, R.S., & Bigonha, M.A.S., "Um método de Compactação de Tabelas LR(1)", Anais do III Simpósio sobre Desenvolvimento de Software Básico para Micros, 141-151, COPPE UFRJ, Rio de Janeiro, 1983.
6. Bigonha, M.A.S., & Bigonha, R.S., "SIC: Manual do Usuário", Monografia do Departamento de Ciência de Computação — número T02/86 — ICEX — UFMG, 1985.
7. Bigonha, M.A.S. & Bigonha, R.S. "Uma experiência na implementação de um Recuperador de Erro LR(1)", Anais do V Simpósio sobre Desenvolvimento de Software Básico, 158-171, UFMG, Belo Horizonte, Minas Gerais, 1985.
8. Bigonha, M.A.S. & Bigonha, R.S., "SIC: Sistema de Suporte a Implementação de Compiladores", Anais do XIII Seminário Integrado de Software e Hardware, 429-442, UFPE, Recife, Pernambuco, 1986.
9. Bigonha, R.S. & Martins R. T., "Compilação Incremental de Linguagens LR(1)", Anais do XIV Seminário Integrado de Software e Hardware, Salvador, Bahia, 1987.
10. Borland International, *Turbo Pascal V. 3.0,* 1985.
11. Borland International, *TURBO Editor Toolbox,* 1985.
12. Burke, M. & Fisher, J.A., "A Practical Method for Syntactic Error Diagnosis and Recovery", CACM, 1982.
13. Burke, M. & Fisher, J.A. "A Practical Methods for LR and LL Syntactic Error Diagnosis and Recovery", TOPLAS, Aprill 1987.
14. Costa e Silva, E., "Um Editor de Programas para Ambientes de Programação", Trabalho a ser publicado como Relatório Técnico do Departamento de Ciência da computação da UFMG, 1988.
15. Jensen, K. & Wirth, N., *Pascal — User Manual and Report,* 2nd Edition, Springer-Verlag, 1974.
16. Johnson, S.C. "YACC — Yet Another Compiler-Compiler", Beli Laboratories Computing Science Technical Report 932, 1978.
17. Knuth, D.E., "On the Translation of Languages from Left to Right", Information and Control, vol 8, 607-639, 1965.
18. Kristensen, B. B. & Madsen, O. L., "Methods for Computing LALR(k) Lookahead", ACM Transactions on Programming Languages and Systems, vol 3, number 1, 60-83, January 1981.
19. Martins, R.T., "Compilação Incremental de Linguagens LR(1)" Dissertação de Mestrado em preparação, Departamento de Ciência da Computação, 1988