UM SISTEMA DE PRODUÇÃO DE GERADORES DE CÓDIGO

Paulo Sávio da Silva Costa[1]

José Lucas Mourão Rangel Netto[2]

RESUMO

O aparecimento nas duas últimas décadas de numerosas linguagens de programação, bem como de muitos processadores funcionalmente distintos, incentivou o desenvolvimento de ferramentas destinadas a automatizar a produção de compiladores. Os primeiros progressos nesse sentido deram-se nos campos de análise léxica e sintática. Ferramentas destinadas à automação da geração de código, contudo, começaram a surgir apenas recentemente, em parte pela dificuldade de se criar modelos formais de um processo tão dependente de máquina.

Este artigo descreve AutoCode, um sistema de produção de geradores de código baseados em reconhecimento de padrões e dirigidos por tabelas criadas automaticamente a partir de uma descrição formal de máquina.

ABSTRACT

The emergence in the last twenty years of numerous programming languages, as well as several functionally contrasting machine architectures, has stimulated the development of software tools to assist the automation of compiler production. The first successful achievements happened in the field of lexical and syntactical analysis. Tools to support code generation, however, have only recently begun to appear, due in a certain degree to the difficult formalization of such a machine dependent process.

This paper describes AutoCode, a code generator production system. The code generators are based on a pattern matching algorithm and driven by tables automatically created from a formal machine description.

---

[1] *Paulo Sávio da Silva Costa é físico pela Universidade Federal do Ceará e mestre em Ciência da Computação pela PUC/RJ. Áreas de interesse: linguagens de programação e compiladores.*

[2] *José Lucas Mourão Rangel Netto é engenheiro eletrônico pela Escola de Engenharia da UFRJ, mestre em Engenharia Elétrica pela COPPE/UFRJ e doutor em Ciência da Computação pela Univ. of Wisconsin, E.U.A. É professor do Inst. de Matemática da UFRJ e Professor Associado do Dep. de Informática da PUC/RJ. Áreas de interesse: engenharia de software, compiladores, linguagens de programação; teoria da computação.*

Endereço: PUC/RJ - Rua Marquês de São Vicente 225 - Rio de Janeiro, RJ - Fone (021) 244.4449

1. APRESENTAÇÃO

A partir da década de 70 pôde-se observar o surgimento de diversas linguagens de programação; simultaneamente, o rápido avanço da tecnologia microeletrônica resultou numa grande variedade de computadores estruturalmente distintos. Como conseqüência, muitas vezes o custo de portabilidade de um compilador eleva-se a níveis proibitivos, trazendo a necessidade de se desenvolver técnicas e ferramentas destinadas a automatizar o processo de produção de compiladores.

Grandes sucessos foram obtidos na automação das fases de análise léxica e sintática. Por outro lado, pesquisas voltadas à automação da geração de código sempre estiveram bastante defasadas, em parte pela dificuldade de formalização de um processo tão dependente de máquina.

Este artigo descreve **AutoCode**, um sistema de geração automática de geradores de código, projetado e implementado segundo as linhas gerais do método introduzido por Graham e Glanville ([Glan77],[Glan78]). O enfoque adotado em AutoCode baseia-se no isolamento e automação das fases da geração de código que são independentes de máquina. Assim um compilador pode ser sistematicamente transportado para novas máquinas a custos bastante reduzidos, apenas modificando-se aquelas partes diretamente relacionadas à arquitetura da máquina.

Diversas deficiências do esquema original de Glanville foram eliminadas. A linguagem descritiva (ver seção 4) foi profundamente reestruturada e acrescida de facilidades para documentação. Introduziu-se ainda uma técnica de fatoração gramatical (seção 5) e os meios necessários à sua utilização (seção 6). Essas extensões resultaram num sistema mais amigável, flexível e dotado de maior poder de exploração dos recursos de arquiteturas pouco simétricas.

2. AUTOCODE : DESCRIÇÃO GERAL DO SISTEMA

A execução do sistema divide-se em duas fases, como mostra a figura 1.

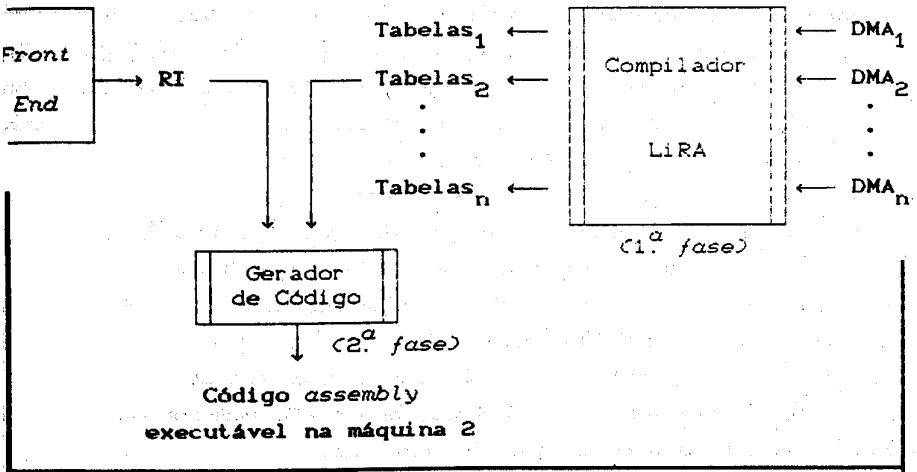

Figura 1 - Organização estrutural do sistema AutoCode

A primeira fase consiste da construção automática das tabelas que guiarão o gerador de código. Dada uma máquina para a qual se deseja gerar código objeto, o usuário inicialmente elabora uma descrição formal de seus registradores e repertório de instruções. Uma descrição de máquina é denominada DMA (Descrição de Máquina Alvo), e deve ser escrita em LiRA (Linguagem de Representação de Arquiteturas Reais). DMA's são processadas pelo compilador LiRA, projetado exclusivamente para esse fim. Esta fase é dita estática, já que é executada uma única vez para cada DMA.

A segunda fase consiste da geração de código propriamente dita. O gerador de código é um *parser* sLR(1) modificado, onde as transições entre estados são guiadas pelas tabelas criadas na primeira fase. Esta é dita uma fase dinâmica, por ser executada a cada compilação de programa fonte.

3. A REPRESENTAÇÃO INTERMEDIÁRIA

Uma RI é uma seqüência de expressões polonesas prefixadas sem parênteses, compostas por um conjunto finito de símbolos V

(vocabulário), classificados em operadores e operandos. A aridade de cada operador deve ser fixa e conhecida; um operador é classificado como externo (*root-level*) ou interno. Um operador externo não pode ser operando de outro operador, ou seja, seu único uso possível é como operador principal de uma árvore de expressão, localizado em sua raiz. Exemplos de operadores externos são := (*store*), j (*jump* - desvio incondicional) e : (definição de rótulo). Operadores internos devem ocorrer como operandos de outros operadores. Exemplos: ? (comparação), ↑ (indireção), operadores aritméticos e lógicos. Uma expressão bem-formada possui um balanceamento correto entre operadores e operandos; uma expressão de nível raiz (*root-level expression*) é uma expressão bem-formada onde a raiz e somente ela é um operador externo.

O *front-end* do compilador é responsável pela conversão do programa fonte na RI equivalente. Este trabalho trata exclusivamente da fase de geração de código de um compilador, de modo que são feitas suposições acerca do ambiente onde o código será executado. Em especial, decisões de implementação relativas à representação de dados e alocação de memória, bem como otimizações independentes de máquina, deverão ser previamente incorporadas à RI. Considere a expressão

```
A := B + C                                                    (1)
```

representando um comando de atribuição numa linguagem do tipo Pascal. Suponha que A seja uma variável local ao nível corrente, B local ao próximo nível estático e C global ao programa. Suponha ainda que variáveis globais residam em posições de memória fixadas em tempo de carregamento do programa, e que variáveis locais residam em posições endereçáveis pela soma de uma base (determinada em tempo de execução) e um deslocamento fixo. Em outras palavras, a estratégia de alocação dinâmica de memória é baseada em registros de ativação, de modo que o registro correntemente ativo encontra-se sempre no topo da pilha de execução. Se sua base for apontada por um registrador de máquina e se o registro ativo contiver um ponteiro para a base do registro correspondendo ao bloco sintaticamente mais global no programa fonte (formando uma estrutura de elos estáticos), então (1)

pode ser reescrita como

```
word st + k.a r.BP
        word + ↑ + k.b ↑ r.BP
               ↑ k.c                    (2)
```

Em (2) **a**, **b** e **c** são constantes que especificam os endereços (deslocamentos) das **variáveis** A, B e C. O operador unário **word** indica que a próxima operação será efetuada sobre uma palavra; ↑ computa o valor armazenado no local de memória apontado por seu operando; **+** indica adição inteira e **st** uma atribuição. O registrador de máquina **BP** aponta para a base local e o acesso a quaisquer variáveis não globais é feito através dele.

4. A LINGUAGEM DESCRITIVA

Uma DMA é a especificação formal da arquitetura da máquina para a qual será gerado código, e deve descrever

- os recursos da máquina, como os registradores presentes na Unidade de Processamento;
- a organização hierárquica desses recursos;
- a estrutura da linguagem *assembly* do código objeto;
- o mapeamento entre as expressões da RI e o repertório de instruções da máquina.

A linguagem LiRA foi projetada para permitir a declaração de todos os elementos presentes na gramática descritiva, possibilitando uma extensa verificação semântica da DMA. LiRA compõe-se de diversas seções, e como regra geral todas elas podem ser declaradas zero ou mais vezes, em qualquer ordem desejada. Há uma para cada tipo de declaração:

Seção de Identificação

Usada apenas para fins de documentação.

Seção de Opções

Lista de opções selecionáveis pelo usuário, determinando o conteúdo das listagens emitidas pelo compilador LiRA.

Seção de Registradores

Lista dos registradores existentes no processador e sua classificação como alocáveis ou dedicados.

Seção de Dependências

Declaração dos grupos segundo os quais os registradores podem ser organizados, por exemplo aos pares.

Seção de Classes

Declaração das classes lógicas formadas pelos registradores físicos da máquina. A cada classe corresponde um não-terminal na gramática descritiva.

Seção de Endereçamento

Lista dos não-terminais usados na gramática para fatorar os modos de endereçamento disponíveis.

Seção de Operadores

Operadores presentes na RI e na gramática descritiva, acompanhados das respectivas aridades.

Seção de Operações

Não-terminais usados na fatoração dos operadores.

Seção de Constantes

Identificadores representando deslocamentos, comprimentos, contadores, etc.

Seção de Formatos

Declaração da sintaxe adotada pelo *assembler* para cada modo de endereçamento.

Seção de Transferências

Declaração das instruções de transferência de dados entre registradores ou entre um registrador e uma posição de memória (despejo/recuperação de registrador).

Seção de Produções

Contém a gramática livre de contexto descrevendo o conjunto de instruções do processador. Cada produção corresponde a um padrão de expressão. Durante a geração de código, uma ação de redução causa a execução dos operadores semânticos associados à produção correspondente.

5. FATORAÇÃO DA GRAMÁTICA DESCRITIVA

O número de produções necessárias à descrição de uma máquina real pode ser bastante grande, devido não apenas à quantidade de instruções disponíveis, mas principalmente aos diversos modos de endereçamento e tipos de dados. Na instrução de adição do VAX/11, por exemplo, cada um dos dois operandos pode ser especificado de 16 modos diferentes. O endereço de destino pode ser (implicitamente) o mesmo do segundo operando, ou pode ser explicitamente fornecido em um dentre os 16 modos. Operandos podem ter 5 tipos (comprimentos) diferentes. Há ainda uma instrução de incremento, em que o endereço do resultado é implicitamente o mesmo do operando fornecido.

A estrutura relativamente regular de certos processadores sugere que partes comuns a vários padrões sejam fatoradas, ao invés de duplicadas nas produções. Dois tipos de fatoração gramatical foram implementados no sistema, descritos a seguir.

Modos de endereçamento

Todo não-terminal de endereço da DMA deverá ser associado, através de uma ou mais produções, a determinado padrão descrevendo um (trecho de) modo de endereçamento. Desde que os padrões fatorados sejam sub-árvores válidas, as propriedades do método de geração de código por reconhecimento de padrões são preservadas. As instruções declaradas numa regra de fatoração de endereçamento devem ser operadores semânticos destinados a gerar e salvar os atributos semânticos da expressão de endereçamento sendo reduzida.

Operadores

Em diversas arquiteturas, operadores como add, mul, and, or e xor poderiam ser agrupados num único não-terminal de operação, digamos binOp. Isto parece razoável por uma serie de razões: todos são operadores binários, todos são comutativos, possuem a mesma sintaxe e a semântica de seus operadores é idêntica. Assim, o novo não-terminal poderia substituir qualquer um dos operadores acima nas regras onde executasse a operação primária da respectiva instrução.

6. OPERADORES SEMÂNTICOS

O esquema de tradução introduzido por Glanville baseia-se em regras de uma gramática livre de contexto descrevendo o conjunto de instruções da máquina. Seu método, contudo, não fornece os meios adequados ao tratamento de certas instruções especiais, tipos de dados, modos de endereçamento complexos e outras assimetrias de arquiteturas reais. Esses problemas foram solucionados com a incorporação de operadores semânticos à linguagem LiRA, com o objetivo de efetuar intervenções semânticas em tempo de geração de código. Isso permite a utilização de informações contextuais durante uma redução, o que aumenta consideravelmente o poder de expressão da gramática descritiva.

Há 18 operadores semânticos disponíveis em LiRA, classificados segundo o tipo de aplicação a que se destinam:

Fatoração dos modos de endereçamento

1. FazBase
2. FazIndice
3. FazDesloc
4. IncNivel
5. DecNivel
6. CopiaDescr
7. SomaDescrs
8. Zera

Fatoração de operadores

9. FazNome
10. CopiaNome

Manipulação da pilha do *parser*

11. Empilha
12. IgnoraLadoEsquerdo

Alocação de registradores

13. Aloca
14. Libera

Tratamento de sub-expressões comuns

15. DefineSEC
16. UsaSEC

Uso geral

17. Custo
18. Emite

7. GERAÇÃO DAS TABELAS

O compilador LiRA é responsável pela geração, a partir de uma descrição de máquina, das tabelas que guiarão o gerador de código. Uma fração considerável do tempo de processamento de

uma DMA é gasto em testes e verificações estáticas.

Análise sintática

LiRA utiliza o método de análise sintática R*S simples (ver [Rang88] e [Schn87]). Esse método diferencia-se de outros da família LR - sLR(1) e laLR(1) - basicamente por:

- eliminar as regras simples durante a construção dos estados do *parser* e
- utilizar mais informações na decisão sobre uma possível redução. Além do estado no topo da pilha e do próximo símbolo na entrada, consulta-se também o estado a ser descoberto caso o *handle* em questão seja desempilhado.

Analisadores R*S possuem menos estados que aqueles baseados nos estados LR(0); há também menos reduções a serem executadas. Além disso, estas são mais simples, pois o método permite a obtenção direta do estado alcançado pelo empilhamento do não-terminal resultante da redução.

Análise semântica

O analisador semântico de LiRA utiliza uma estrutura de dados em forma de árvore para representar internamente a DMA sendo processada. Toda redução por regra não-simples realizada pelo analisador sintático causa uma chamada ao procedimento RotSem. Este realiza o tratamento semântico associado à respectiva regra, ao mesmo tempo construindo e manipulando a árvore da DMA.

Detecção e remoção de ciclos

Um *parser* do tipo empilha/reduz encontra-se em ciclo quando realiza um número ilimitado de transições sem consumir símbolos da entrada. Considerando a natureza do algoritmo de geração de código, um ciclo seria gerado no caso do *parser* executar uma série de reduções por regras simples, ou seja, regras do tipo $X \rightarrow Y$ onde X e Y são não-terminais. Essas reduções não alteram o tamanho da pilha nem consomem símbolos da entrada, levando o *parser* a repetir uma configuração anterior. Ciclos são automaticamente detectados e removidos

examinando-se o conjunto de estados e itens sLR(1) da gramática descritiva (ver [Cost90]).

Bloqueios sintáticos

Quando o *parser* chega a um estado em que não há ação possível com o próximo símbolo da RI (*lookahead*), ele é dito sintaticamente bloqueado, significando que a RI não faz parte da linguagem definida pela gramática descritiva.

Há uma condição suficiente para garantir que o gerador de código não bloqueará quando submetido a RI's válidas. Essa condição é definida em termos da uniformidade da gramática descritiva; a idéia básica é que operandos de um operador sejam válidos independentemente do contexto onde ocorrem.

A definição de uniformidade relaciona-se essencialmente à gramática descritiva, por isso pode ser verificada em tempo de geração das tabelas do *parser*. Suponha, entretanto, que a gramática descritiva não seja uniforme. Garante-se que não ocorrerá um bloqueio sintático certificando-se que as duplas <estado, *lookahead*> para as quais não há ação possível nunca serão configuradas. Para isso deve-se garantir que, dado o contexto, também a construção na RI que causaria o bloqueio jamais seria gerada.

Bloqueios semânticos

Quando o *parser* chega a um estado de redução, as regras correspondentes a seus itens completos mais longos são testadas seqüencialmente, em busca de alguma semanticamente compatível com a expressão no topo da pilha. Caso nenhuma o seja, o *parser* encontra-se semanticamente bloqueado. Essa situação pode ser evitada utilizando-se uma lista de produções *default*, mais curtas que aquelas restritas e que calculam a expressão desejada. Listas de reduções *default* são automaticamente geradas por LiRA para cada estado de redução em que todas as regras possuem restrições semânticas.

Uma lista *default* é construída simulando-se as ações do gerador de código ao receber como entrada o lado direito da regra restrita, utilizando apenas as regras cujos lados direitos sejam menores do que aquele. O processo consiste

basicamente em construir um gerador de código para o subconjunto de regras mais curtas, simular sua execução e anotar as regras e locais de cada redução realizada.

8. GERAÇÃO DE CÓDIGO

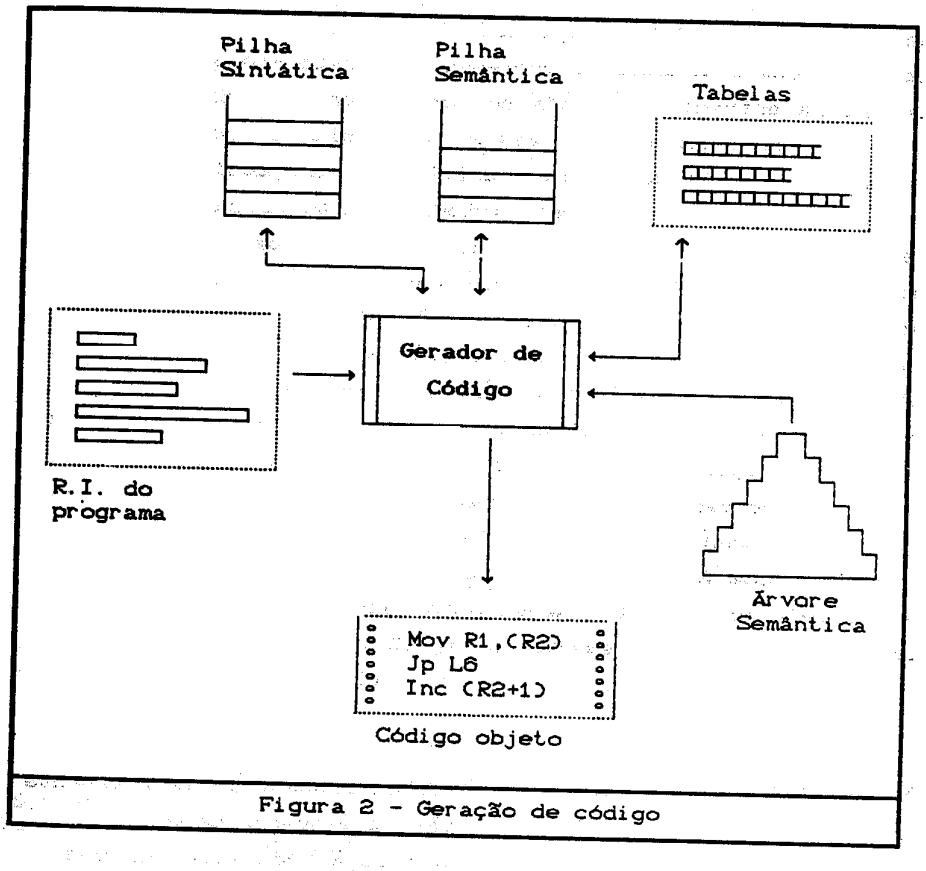

Figura 2 - Geração de código

O algoritmo de geração de código tem a estrutura de um *parser* determinístico sLR(1). Devido à natureza tipicamente ambígua de gramáticas descritivas, a técnica padrão de análise LR não se aplica, especialmente em relação

- à política de resolução de conflitos,
- ao tratamento de regras com lado esquerdo λ (ou seja, regras descrevendo expressões cujo interesse primário

reside em seus efeitos colaterais, por exemplo atribuições e desvios incondicionais) e

- à semântica associada a ações de redução em geral.

A pilha sintática do *parser* é inicializada com o estado inicial no topo. Símbolos da entrada e respectivas informações semânticas são lidos e empilhados; cada um seguido do estado alcançado com o símbolo. O processo se repete até que o próximo símbolo na entrada não possa ser empilhado a partir do estado corrente, configurando-se assim a resolução dos conflitos empilha/reduz. Nesse caso, verifica-se a possibilidade de redução. Se não for possível, AutoCode emite mensagem de erro, visto que a RI não faz parte da linguagem gerada pela gramática descritiva. Havendo mais de uma redução possível no estado, necessariamente os lados direitos de todas as regras possuem o mesmo padrão sintático; a escolha da regra é determinada tanto pelas restrições semânticas presentes em cada uma como pelo conteúdo da pilha semântica. Listas de reduções múltiplas são ordenadas segundo os custos das reduções (opcionalmente declarados em cada produção; iguais a 1 por *default*). As regras são então testadas até se encontrar uma semanticamente compatível com a expressão no topo da pilha. Caso nenhuma das regras seja compatível, usa-se então as listas *default* geradas por LiRA a fim de implementar a redução.

Uma vez determinada a regra pela qual será feita a redução, digamos $x \rightarrow \alpha$, o algoritmo desempilha $|\alpha|$ símbolos (além dos eventuais atributos semânticos) descobrindo o estado q', que encontrava-se no topo da pilha sintática imediatamente antes do empilhamento do símbolo mais à esquerda de α.

O tratamento semântico de uma redução consiste, no modelo de Glanville, simplesmente em emitir a instrução de máquina associada à regra. Em AutoCode, uma redução causa a execução dos operadores semânticos declarados no escopo da regra. Assim, pode-se emitir mais de uma instrução durante uma redução.

O lado esquerdo da regra de redução pode ser um não-terminal, especificando uma classe de registrador (por exemplo r), ou então lambda. Se $x \neq \lambda$ então r é empilhado,

seguido de seu atributo semântico (registrador físico) e do estado q'' alcançado a partir de q' com o símbolo r. A partir daí o processo continua, até que o *parser* chegue ao estado final - ou a algum estado de erro.

9. EXEMPLO: INTEL 8088

AutoCode foi utilizado na geração automática de um gerador de código para o Intel 8088. A DMA escrita para esse processador descreve um subconjunto de suas instruções, em parte porque muitas delas não possuem operações correspondentes na RI emitida pelo *front-end*. Os modos de endereçamento do 8088 (imediato, registrador, indireto, com base, indexado e base-indexado) são adequadamente expressos pela gramática. O modo direto foi ignorado, já que os rótulos de definição de dados envolvidos não existem na RI. Também foi ignorado o modo de endereçamento de *strings*, devido ao efeito colateral do auto-incremento implícito dos registradores SI e DI.

Resumo das declarações na DMA

```
 22 registradores
 22 operadores
 14 não-terminais de endereço
  4 não-terminais de operação
110 regras
177 instruções
```

Geração das tabelas

Foram gerados, inicialmente, 222 estados. 76 ciclos foram detectados e removidos e 11 novos estados foram criados. Considerando que 4 tornaram-se inacessíveis, o número total de estados ficou em 229. Listas de reduções *default* foram geradas para 22 estados de redução contendo apenas regras semanticamente restritas. LiRA gera dois arquivos durante o processamento de uma DMA. No caso do 8088, foram gerados:

- 8088.LST: listagem da DMA e diversas informações adicionais.
- 8088.TRB: arquivo contendo a árvore semântica e as tabelas que guiarão o gerador de código.
  Tamanho: 29 KBytes.

Tempos de execução

Análises (léxica, sintática e semântica)	30 seg
Geração dos estados e teste de uniformidade	169 seg
Detecção e remoção de ciclos	21 seg
Geração das listas de reduções *default*	364 seg
Compactação das tabelas	105 seg
Outros	20 seg
TOTAL	(11 min 45 seg)

NOTA: O sistema foi executado num microcomputador IBM-PC XT com 640 KB de memória, *clock* de 10 MHz e *Winchester* de 30 MB.

Exemplo de geração de código

```
Function Fatorial (n : Integer) : Integer;
Var
  i, f : Integer;
Begin
  f := 1;
  For i := 2 To n Do
    f := f * i;
  Fatorial := f;
End;
```

(RI)	(Código gerado)

word st + k.-4 r.BP k.1
word st + k.-2 r.BP k.2
word st + k.-6 r.BP
↑ + k.-8 r.BP
jp rot.2
def rot.1
word st + k.-4 r.BP
word * ↑ + k.-4 r.BP
↑ + k.-2 r.BP
word st + k.-2 r.BP
word + k.1
↑ + k.-2 r.BP
def rot.2
lt rot.1 word ? ↑ + k.-2 r.BP
↑ + k.-6 r.BP
word st + k.-10 r.BP
↑ + k.-4 r.BP
	MOV WORD -4[BP], +1
	MOV WORD -2[BP], +2
	MOV WORD -6[BP], -8[BP]
	JMP L2
	L1 :
	MOV SI, -2[BP]
	MOV AX, SI
	IMUL WORD -4[BP]
	MOV WORD -4[BP], AX
	INC WORD -2[BP]
	L2 :
	MOV AX, -6[BP]
	CMP WORD -2[BP], AX
	JL L1
	MOV WORD -10[BP], -4[BP]

10. RESULTADOS OBTIDOS

Os resultados obtidos com a pesquisa foram extremamente

favoráveis. O modelo de geração de código dirigido por tabelas, em torno do qual foi desenvolvido o trabalho, constitui uma valiosa contribuição à automação do processo de construção de compiladores. O método utilizado goza de uma clareza inexistente em geradores de código convencionais. Gramáticas descritivas podem ser sistematicamente adaptadas ou mesmo reescritas para novas máquinas. Isso confere ao método uma garantia de correção bastante superior a outros métodos automatizados. Além disso, instruções especiais e modos de endereçamento complexos são facilmente descritos; assim a qualidade do código gerado pode ser controlada e elevada a um custo realmente baixo.

Algumas dificuldades encontradas merecem ser mencionadas. O processador escolhido para teste do sistema, o 8088, possui registradores e instruções extremamente assimétricos; como conseqüência, a respectiva gramática descritiva tornou-se extensa e pouco clara. Apesar disso o 8088 foi escolhido por ser utilizado nos microcomputadores IBM-PC aos quais têm acesso os mestrandos do Departamento de Informática da PUC/RJ, facilitando a execução de testes do código gerado. Outro problema crítico está relacionado à linguagem de implementação do sistema, Turbo Pascal 5.0. Este compilador limita a área de dados de um programa a 64 KB, insuficiente para abrigar todas as estruturas de dados manipuladas por LiRA. Provavelmente esses problemas seriam evitados optando-se por um computador de maior porte para a implementação do sistema.

A qualidade do código gerado por AutoCode para o 8088 é por vezes prejudicada pela pouca simetria de seus registradores e instruções. Esse problema pode ser superado submetendo o código emitido a um otimizador do tipo *peep-hole*. Finalmente, espera-se que os experimentos a serem realizados com arquiteturas mais simétricas forneçam resultados ainda melhores.

11. REFERÊNCIAS BIBLIOGRÁFICAS

Bird82 BIRD, P. L. An Implementation of a Code Generator

Specification Language for Table Driven Code Generators. SIGPLAN Notices, 17 (6): 44-55, June 1982.

Catt80 CATTEL, R. G. G. Automatic Derivation of Code Generators From Machine Descriptions. ACM Transactions on Programming Languages and Systems, 2 (2): 173-90, April 1980.

Cost90 COSTA, P. S. Um Gerador Automático de Geradores de Código. Dissertação de Mestrado, Departamento de Informática, PUC/RJ, Fevereiro de 1990.

Craw82 CRAWFORD, J. Engineering a Production Code Generator. SIGPLAN Notices, 17 (6): 205-15, June 1982.

Gana82 GANAPATHI, M., FISCHER, C. N. e HENNESSY, J. L. Retargetable Compiler Code Generation. Computing Surveys, 14 (4): 573-92, December 1982.

Glan77 GLANVILLE, R. S. A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers. Dissertação de Ph.D. Departamentos de Engenharia Elétrica e Informática. Universidade da Califórnia, Berkeley, Dezembro de 1977.

Glan78 GLANVILLE, R. S. e GRAHAM, S. L. A New Method for Compiler Code Generation. Fifth ACM Symposium on Principles of Programming Languages. Tucson, Arizona, January 1978. *Proceedings*. ACM, New York, January 1978, p. 231-40.

Grah83 GRAHAM, S. L. Code Generation And Optimization (Preliminary Version). Construction de Compilateurs: Méthodes et Outils, INRIA, Rocquencourt, Decembre 1983.

Inte86 INTEL CORPORATION. iAPX 86/88, 186/188 User's Manual - Programmer's Reference. Santa Clara, CA, 1986.

Land82 LANDWERH, R., JANSOHN, H. S. e GOOS, G. Experience With an Automatic Code Generator Generator. SIGPLAN Notices, 17 (6): 56-66, June 1982.

Rang88 RANGEL, J. L. Manual de Operação do Sistema de Geração de Analisadores Sintáticos R*S Simples. Série Monografias em Ciência da Computação, Nr. 7, 1988. Pontifícia Universidade Católica do Rio de Janeiro.

Schn87 SCHNEIDER, S. M. Gramáticas e Analisadores R*S. Tese de Doutorado, Prog. Eng. Sistemas e Computação, COPPE / UFRJ, Setembro de 1987.