JÚLIO CÉZAR LUZ

TECNOLOGIA ADAPTATIVA APLICADA À OTIMIZAÇÃO DE CÓDIGO EM COMPILADORES

Dissertação apresentada à Escola
Politécnica da Universidade de
São Paulo para obtenção do
Título de Mestre em Engenharia Elétrica.

São Paulo
2004

JÚLIO CÉZAR LUZ

TECNOLOGIA ADAPTATIVA APLICADA À OTIMIZAÇÃO DE CÓDIGO EM COMPILADORES

Dissertação apresentada à Escola
Politécnica da Universidade de
São Paulo para obtenção do
Título de Mestre em Engenharia Elétrica.

Área de Concentração:
Sistemas Digitais

Orientador:
Prof. Dr.
João José Neto

São Paulo
2004

Este exemplar foi revisado e alterado em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, ... de ..... de 200...

Assinatura do autor

Assinatura do orientador

FICHA CATALOGRÁFICA

Luz, Júlio Cézar
Tecnologia Adaptativa Aplicada à Otimização de Código em Compiladores. Edição Revisada. São Paulo, 2004.
146 p.

Dissertação (Mestrado) – Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1. Teoria e Técnicas de Programação 2. Montadores e Compiladores 3. Teoria e Construção de Compiladores 4. Geração de Código I. Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais II. t

Para os meus pais,
Cézar de Oliveira Luz e
Vera Maria Gonsalves Luz

AGRADECIMENTOS

Ao meu orientador, *Prof. Dr. João José Neto*, pelas inúmeras sugestões e comentários visando o aprimoramento deste trabalho, pelo incentivo e apoio dados e pela disposição em solucionar as minhas dúvidas, por menores que fossem, com paciência e compreensão.

Aos amigos e colegas, *Mestre José Antônio Fonseca* e *Prof. Dr. Flávio Matsuyama*, por me incentivarem a ingressar no programa de pós-graduação e pelo apoio dado.

A todos os colegas do *LTA - Laboratório de Linguagens e Tecnologias Adaptativas* e do *Departamento de Engenharia de Computação e Sistemas Digitais* que contribuíram, direta ou indiretamente, para a realização deste trabalho.

E, finalmente, à minha esposa, *Sra. Luciana Fredi*, por me incentivar a ingressar no programa de pós-graduação, pelo carinho e apoio dados ao longo desta caminhada e pela paciência e compreensão em sacrificar muitos finais de semana para que este trabalho se tornasse realidade.

RESUMO

O espaço de memória de programação é normalmente limitado em sistemas microcontrolados embutidos. Embora os compiladores atuais apliquem transformações otimizantes ao software embutido, a exigüidade de espaço de memória pode se tornar um problema crítico para o projetista quando da necessidade de introdução de novas funcionalidades e correções no software original. Por outro lado, estações de trabalho hospedando sistemas de desenvolvimento para aplicações embutidas são mais rápidas e dispõem de maior quantidade de memória. Diante desse panorama, o presente trabalho apresenta um otimizador *peephole* que explora uma técnica adaptativa que, embora mais exigente em termos de memória e tempo de execução, é capaz de proporcionar uma melhor taxa de redução do código objeto do que um otimizador *peephole* convencional. Ações adaptativas permitem que o algoritmo auto-modifique o seu comportamento em resposta a condições de entrada específicas e procure a seqüência de regras de otimização que melhor aperfeiçoe o código objeto dentre as muitas seqüências possíveis resultantes da superposição de duas ou mais regras de otimização igualmente aplicáveis.

ABSTRACT

The programming memory space is usually limited in embedded microcontrolled systems. Although, nowadays compilers apply optimizing transformations to embedded software, the lack of memory space may become a critical problem to the designer as an effect of the introduction of new features and corrections into the original software. In contrast, workstations hosting development systems for embedded applications are faster and have much more memory space. In such a scenario, our work introduces a peephole optimizer exploring an adaptive technique that, although requiring more memory space and execution time, is capable to achieve better reduction ratio of the object code than conventional peephole optimizers. Adaptive actions enable the algorithm to self-modify its behavior in response to a specific input condition and to search among the many possible sequences resulting from the superposition of two or more equally applicable optimization rules for the sequence of optimization rules that best improves the object code.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

1. INTRODUÇÃO

Os computadores surgidos após o final da segunda guerra mundial eram máquinas grandes, caras e com capacidade de computação limitada, quando comparadas com a dos computadores atuais. O espaço reduzido de memória era um dos fatores limitantes da capacidade de computação. Os programas escritos em linguagem de montagem para esses computadores eram otimizados manualmente a fim de maximizar o uso do espaço de memória disponível. Desde aquela época, os avanços proporcionados pela integração em grande escala de circuitos semicondutores reduziu o tamanho dos computadores, barateou a sua produção e aumentou extraordinariamente a sua capacidade de computação. Paralelamente, surgiram ferramentas de programação mais adequadas para o desenvolvimento de programas aplicativos maiores e mais complexos.

Como conseqüência desse processo, os esforços gastos antigamente para reduzir o espaço ocupado pelos programas foram redirecionados para a arquitetura de programas sofisticados que aplicam intensivamente o conceito de reutilização de código. A reutilização de código foi um dos fatores que motivaram o desenvolvimento das atuais linguagens de programação de alto nível (Sammet, 1969), pois aumentou expressivamente a produtividade da programação pelo reaproveitamento do código escrito por outros projetistas na forma de bibliotecas de rotinas ou pelo mecanismo de herança. A utilização de linguagens de programação de alto nível permitiu também que os projetistas se concentrassem na lógica e na correção do programa ao invés do espaço de memória ocupado e o tempo de execução (Aho; Ullman, 1979) e (Watson, 1989).

Contudo, na programação de sistemas microcontrolados embutidos[1] o espaço de memória ocupado e o tempo de execução do programa ainda desempenham um papel importante. De Sutter e De Bosschere (2003) apontam duas razões para isso.

A primeira delas é que o *hardware* empregado nos sistemas de telecomunicações móveis e nas redes sem fio continua caro. A produção em massa desses equipamentos faz com que o custo do *software* por unidade produzida seja reduzido em comparação com o custo do *hardware*. O fator dominante que encarece

[1] *Embedded microcontrolled systems*

o *hardware* é a área física total ocupada pelo circuito semicondutor, sendo uma fração substancial dessa área ocupada pela memória para armazenar código e dados, permanente (ROM[2]) ou temporariamente (RAM[3]). Uma forma de reduzir o custo do *hardware* é diminuir a área ocupada pela memória, mas isso só pode ser feito se os programas armazenados ocuparem um espaço menor, ou, ainda, dada uma área fixa de memória, a redução do espaço ocupado pelo programa libera espaço de memória para o acréscimo de novas funcionalidades, que agregam valor ao sistema final.

A segunda delas é que os paradigmas de programação atuais não se adequam às restrições impostas pelos sistemas microcontrolados embutidos. Pelo contrário, o estímulo à reutilização de código resulta em programas cada vez maiores em função da necessidade de se produzirem bibliotecas genéricas que atendam um grande número de aplicações. Como resultado, tais bibliotecas fornecem mais funcionalidades do que um programa aplicativo específico realmente precisa. A menos que seja possível eliminar as funcionalidades desnecessárias, um programa aplicativo específico será maior do que o necessário para conter as funcionalidades de que necessitam. A Microsoft, por exemplo, removeu as funções menos utilizadas do sistema operacional Windows, e passou a chamá-lo de Windows CE[4], a fim de reduzir o espaço ocupado pela versão completa e possibilitar a sua execução em dispositivos de computação móveis, como PDA's[5], *palmtops*, etc., que contêm espaço de memória reduzido (Bentley, 2000).

1.1. Objetivos

Assim, embora os compiladores das linguagens de programação atuais apliquem transformações otimizantes ao programa-fonte, a falta de espaço de memória pode se tornar um problema crítico para o projetista de programas de sistemas microcontrolados embutidos à medida que tais sistemas evoluem com a introdução de correções e/ou novos recursos. Devido aos acréscimos inerentes dos paradigmas de programação, o espaço ocupado pelo código gerado a partir de uma

---

[2] *Read only memory*

[3] *Random access memory*

[4] *Windows compact edition*

[5] *Personal digital assistant*

especificação de requisitos freqüentemente deve ser otimizado manualmente (De Sutter; De Bosschere, 2003). Por outro lado, a cada ano, os computadores pessoais que hospedam os sistemas de desenvolvimento de microcontroladores embutidos dispõem de mais memória e estão cada vez mais rápidos.

Diante deste panorama, pretende-se aplicar uma técnica de projetos de algoritmos adaptativos a um novo algoritmo de otimização que requer mais espaço de memória e/ou tempo de execução em troca de uma maior redução do tamanho do código-objeto ocupado pelo software do sistema embutido. Mais precisamente, o presente trabalho trata o tema *tecnologia adaptativa aplicada à otimização de código em compiladores*.

Na revisão da literatura se apresentam conceitos de tecnologia adaptativa e um estudo detalhado de uma das formas mais populares de otimização do código-objeto, a otimização *peephole*.

Na parte teórica se discute o projeto de um algoritmo de otimização *peephole* implementado com tecnologia adaptativa, em outras palavras, de um algoritmo de otimização *peephole* adaptativo, e se realiza uma análise de complexidade do algoritmo resultante.

Na parte prática, se apresenta uma implementação do algoritmo de otimização *peephole* adaptativo proposto, se realiza a validação de um conjunto de regras de otimização e se esboça um pequeno estudo experimental comparativo do algoritmo implementado de modo convencional com o algoritmo adaptativo.

Espera-se que a aplicação de tecnologia adaptativa descrita neste trabalho desperte interesse e sirva como fonte de consulta para a realização de novos trabalhos. Portanto, ao longo da parte teórica e da parte prática, indicam-se algumas sugestões para trabalhos futuros.

1.2. Motivação

O estudo de trabalhos recentes sobre redução de código sugerem a existência de duas grandes frentes de pesquisa nesta área tecnológica: uma delas explora a aplicação de algoritmos de compressão de código e a outra investiga a aplicação mais eficaz das técnicas clássicas de redução de código.

Trabalhos na primeira categoria optam pela obtenção de um código

comprimido que possa ser descomprimido antes da execução (Ernst et al., 1997), (Franz; Kistler, 1997) e (Drinic et al., 2003), ou, alternativamente, possa ser interpretado sem descompressão (Proebsting, 1995). Contudo, estas técnicas requerem modificações no ambiente de execução[6] para descomprimir o código, ou então no próprio hardware do processador, para possibilitar a execução direta do código comprimido (Debray et al., 2000), acarretando assim o encarecimento do produto final.

Trabalhos na segunda categoria optam por produzirem um código que seja diretamente executável pelo processador, e requerem, portanto, compiladores otimizadores que utilizem mais eficientemente as técnicas clássicas de redução de código. Entre tais técnicas, a abstração de procedimento[7] é uma das mais estudadas (Fraser et al., 1984), (Cooper; McIntosh, 1999), (Debray et al., 2000), (Runeson, 2000), (Nyström et al., 2001) e (Kim; Lee, 2002). Nessa técnica, procuram-se seqüências repetidas de instruções no código-objeto que possam ser substituídas por chamadas de um procedimento equivalente. Pode ser aplicada diretamente ao código-objeto (Fraser et al., 1984), (Cooper; McIntosh, 1999), (Debray et al., 2000) e (Kim; Lee, 2002) ou ainda ao código intermediário (Runeson, 2000) e (Nyström et al., 2001).

A obtenção de taxas cada vez mais expressivas de redução de código se deve ao contínuo aprimoramento da técnica e/ou do uso da abstração de procedimento em combinação com técnicas tradicionais de redução de código. Cooper et al. (1999, 2002) são exemplos de utilização mais eficiente das técnicas tradicionais de redução de código: no primeiro caso (Cooper et al., 1999) emprega-se um algoritmo genético na coordenação da aplicação das transformações otimizantes de um compilador otimizador, utilizando o tamanho do código produzido como realimentação na busca de uma seqüência de transformações otimizantes que melhor reduza o espaço ocupado pelo código; no segundo caso (Cooper et al., 2002) descreve-se um protótipo de um compilador que procura uma seqüência de transformações otimizantes que minimize uma determinada função objetivo.

Van de Wiel, R. (2004) descreve uma extensa bibliografia de trabalhos

[6] *Run-time system*

[7] *Procedural abstraction*

dedicados ao tema da redução de código.

O presente trabalho se classifica na segunda categoria, porém sugere um tratamento mais sistemático para a obtenção de algoritmos de otimização em função do emprego da tecnologia adaptativa. A técnica de otimização *peephole* foi escolhida porque: (a) é uma técnica de otimização simples que se aplica em uma vizinhança limitada do código-objeto, ou de formas intermediárias de representação do programa, e na qual se procura eliminar instruções redundantes e substituir seqüências de instruções ineficientes por outras mais eficientes. Por se tratar de uma técnica simples, é adequada para exemplificar o uso de métodos adaptativos na obtenção de algoritmos poderosos a partir de algoritmos convencionais; (b) pode ser empregada em qualquer etapa da síntese do código-objeto, ou seja, na otimização do código intermediário ou na do código-objeto, (Cooper et al., 1999), (Morgan, 1998) e (c) pode acelerar o processo de compilação como um todo em virtude da redução do número de instruções a serem processadas nos passos subseqüentes (Debray et al., 2000).

1.3. Organização desta dissertação

O capítulo 2 conceitua a tecnologia adaptativa e mostra um exemplo simples de projeto de algoritmo adaptativo através do projeto de um autômato adaptativo a partir de um autômato finito.

O capítulo 3 situa o contexto em que se insere o trabalho. Para tanto descreve o processo de compilação para uma arquitetura monoprocessada hipotética até a etapa de interesse para o trabalho, ou seja, a fase de otimização do código-objeto. Conceitua-se então a otimização *peephole*, descrevem-se os trabalhos anteriores e se traça a sua evolução desde os algoritmos primordiais até os trabalhos mais recentes, passando pelo emprego de técnicas de análise de fluxo para a geração automática das regras de otimização e os algoritmos de otimização *peephole* descritos na literatura.

O capítulo 4 apresenta um algoritmo de otimização *peephole* adaptativo, descreve um exemplo simples de operação, faz algumas considerações a respeito de alguns fatores que afetam o funcionamento de tais algoritmos de otimização *peephole* em geral e faz uma análise de complexidade pessimista do algoritmo adaptativo.

O capítulo 5 complementa o anterior com uma implementação prática do

algoritmo proposto e relaciona várias sugestões para a elaboração de trabalhos futuros.

O capítulo 6 faz uma série de considerações finais e encerra o trabalho com a análise e avaliação dos resultados, dos objetivos propostos, bem como das conclusões conceituais e práticas fundamentadas nos resultados obtidos.

A lista de referências relaciona todo o material referenciado no texto e consultado durante a elaboração do trabalho. Em seguida, seguem-se um glossário de termos e os apêndices que complementam o trabalho.

2. TECNOLOGIA ADAPTATIVA

A tecnologia adaptativa trata técnicas e dispositivos que modificam espontaneamente o seu comportamento em resposta a certas ocorrências de entrada (José Neto, 2001). Os dispositivos adaptativos são obtidos a partir de dispositivos mais simples através do acoplamento de mecanismos que estendem a capacidade do dispositivo original. Os autômatos adaptativos formam uma classe particular de dispositivos adaptativos, como resultado da associação de *ações adaptativas* aos autômatos de pilha estruturados. Uma ação adaptativa é um procedimento inerente ao formalismo que modifica o comportamento do autômato de pilha estruturado subjacente. As ações adaptativas ficam associadas a determinadas transições do autômato, e as transições assim estendidas passam a se chamar *transições adaptativas*. Quando se reúnem as condições necessárias para a execução de uma transição adaptativa, o autômato promove a execução da ação adaptativa associada a essa transição.

A teoria formal dos autômatos adaptativos (o apêndice A apresenta uma breve introdução à teoria e notação dos autômatos adaptativos) e algumas aplicações de autômatos adaptativos à teoria de linguagens são descritos em (José Neto, 1993) e (José Neto, 1994). Mais exemplos de aplicações de autômatos adaptativos, conseqüentes da propriedade de os autômatos adaptativos apresentarem a mesma potência das máquinas de Turing, são discutidos em (José Neto, 2001).

A evolução da tecnologia adaptativa, a partir dos autômatos adaptativos até a elaboração de um formalismo que unificasse a representação e o tratamento de dispositivos adaptativos baseados em regras e a introdução de tabelas e árvores de decisão adaptativas, passando por um breve apresentação das aplicações à teoria das linguagens, especificação e projeto de sistemas reativos complexos, ao processamento de linguagens naturais, aprendizagem e visão computacional e modelagem de sistemas distribuídos, é descrita em (Pistori , 2003).

2.1. Noção de projeto de algoritmos adaptativos

Pela Tese de Church (Lewis; Papadimitriou, 1981), há uma identificação possível entre autômatos e algoritmos. Assim sendo, pode-se dizer que um algoritmo é adaptativo quando modificar espontaneamente o seu comportamento em resposta a uma condição especial de entrada. Como conseqüência, podem-se formular algoritmos adaptativos pela introdução de ações adaptativas em algoritmos convencionais, da mesma forma como se formulam autômatos adaptativos pelo acoplamento de ações adaptativas a autômatos convencionais (José Neto, 2002).

Vamos ilustrar esta noção de projeto de algoritmos adaptativos através do projeto de um autômato adaptativo que aceita a linguagem $\{a^n b^n c^n \mid n > 0\}$ a partir de um autômato finito que aceita a linguagem {abc}.

Da teoria das linguagens sabe-se que o reconhecimento da linguagem $\{a^n b^n c^n \mid n > 0\}$não pode ser feita por um autômato finito ou de pilha, pois essa linguagem não é regular nem livre de contexto (Lewis; Papadimitriou, 1981). Seja, portanto, o autômato finito, representado na figura abaixo, que aceita a linguagem {abc}:

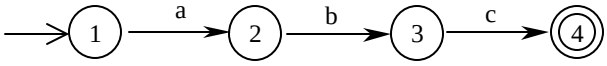

Figura 2.1 - Aut ômato finito que aceita apenas a cadeia abc

O autômato finito aceita a cadeia abc. Caso se acrescente ao autômato uma transição adicional do estado 2 para o próprio estado 2, consumindo o átomo 'a', o autômato passa a reconhecer a linguagem a^+bc. Acoplando-se a essa nova transição a ação adaptativa seguinte:

A(x,y) = {p*, q*:
-[(2,b) → x]
+[(2,b) → p*]
+[(p*,b) → x] -[(y,c) → 4]
+[(y,c) → q*] +[(q*,c) → 4]
-[(2,a) → 2,A(x,y)]
+[(2,a) → 2,A(p*, q*)] }

e estabelecendo-se uma configuração inicial adequada para o autômato, conforme mostra a figura 2.2, obtém-se um autômato adaptativo que aceita a linguagem $\{a^n b^n c^n \mid n > 0\}$ pela extensão gradativa da aceitação da subcadeia bc para bbcc, bbbccc, e assim sucessivamente, até $b^n c^n$, em resposta ao consumo de cada átomo 'a' adicional

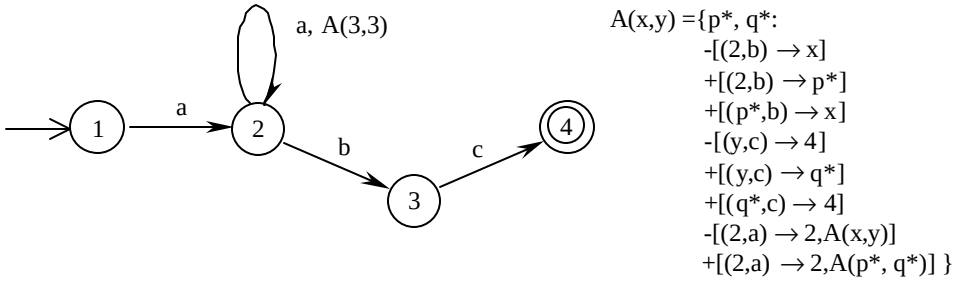

Figura 2.2 - Topologia. inicial do autômato adaptativo que aceita a cadeia $a^n b^n c^n$

Os elementos p* e q* são geradores referentes aos novos estados inseridos no autômato a cada execução da ação adaptativa A. As três primeiras ações adaptativas elementares efetuam a inclusão de um novo estado entre o estado 2 e o estado para o qual a transição com consumo de 'b' se dirigia. As próximas três ações efetuam a mesma tarefa para o estado do qual a transição com consumo de 'c' saía em direção ao estado 4 e as duas últimas ações efetuam a troca da transição adaptativa do estado 2 por uma nova transição adaptativa que prepara o autômato para um eventual consumo de outro átomo 'a'. Assim, dada a cadeia de entrada aaabbbccc e o autômato adaptativo representado na figura 2.2, o consumo do primeiro átomo 'a' leva a configuração do autômato do estado 1 para o estado 2 e o consumo do segundo átomo 'a' mantém o autômato no estado 2 e transforma a topologia inicial do autômato naquela representada na figura 2.3:

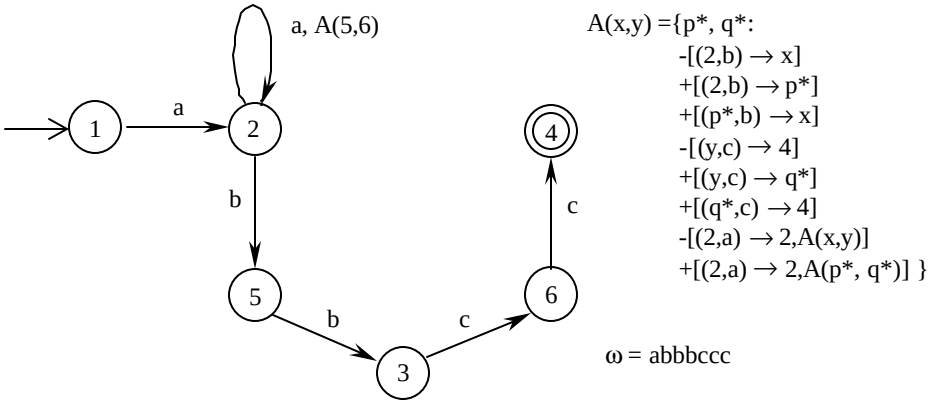

Figura 2.3 - Topologia do autômato após o consumo do segundo átomo 'a'

O consumo do terceiro átomo 'a' também mantém o autômato no estado 2 e transforma a topologia da figura 2.3 naquela representada na figura 2.4:

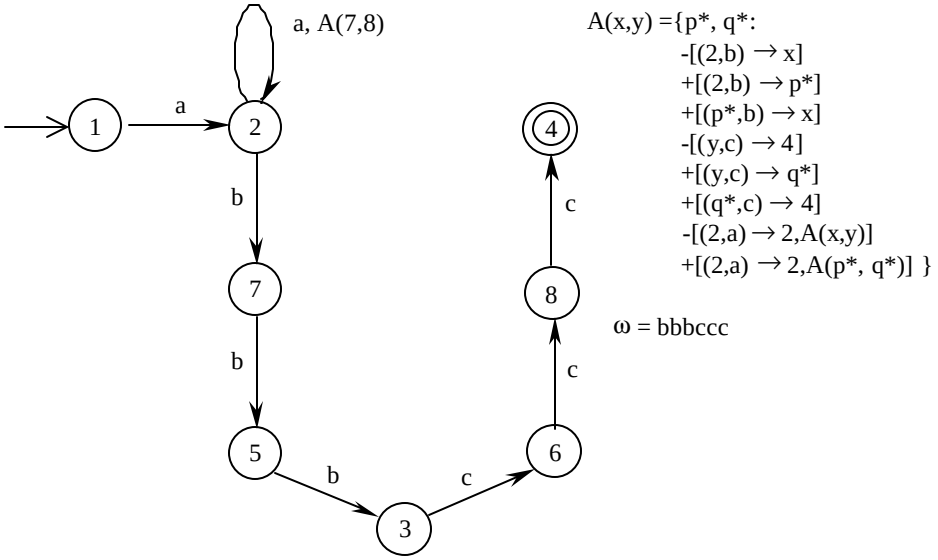

Figura 2.4 - Topologia do autômato após o consumo do terceiro átomo 'a'

Como não restam mais átomos 'a' na cadeia de entrada, a topologia do autômato da figura 2.4 se mantém e o autômato consome o restante da cadeia de entrada transitando do estado 2→7→5→3→6→8→4, terminando por aceitar a cadeia de entrada. Este exemplo mostra como problemas de solução complexa para formalismos usuais podem ser resolvidos de maneira simples e eficiente com o uso de autômatos adaptativos. Outros problemas de difícil solução, mas que podem ser tratados de forma relativamente confortável usando autômatos adaptativos são descritos na literatura (José Neto, 2001).

Tendo uma noção de projeto de algoritmos adaptativos, o capítulo seguinte descreve as fases de um compilador otimizador hipotético para arquiteturas monoprocessadas em que se pretende aplicar este conceito, mais especificamente, na fase de otimização do código-objeto.

3. ORGANIZAÇÃO DE UM COMPILADOR OTIMIZADOR

Um compilador é um programa que aceita como texto de entrada um programa-fonte descrito em alguma linguagem de programação de alto nível e produz como texto de saída um programa equivalente descrito em alguma linguagem de montagem, ou código-objeto, que possa ser executada por um processador[8]. Esta definição pode ser estendida para sistemas que traduzam de uma linguagem de alto nível para outra, de uma linguagem de montagem para outra, de uma linguagem de alto nível para uma forma de representação intermediária, etc.

Conforme Ceruzzi (1998), a programação dos primeiros computadores eletromecânicos, como o Mark I de Harvard, era feita pela perfuração de códigos em fita de papel para cada instrução de máquina. Com o passar do tempo, algumas das seqüências mais utilizadas de códigos de furos eram permanentemente gravadas no hardware do computador sem aumentar consideravelmente, contudo, a sua flexibilidade. Como a máquina não tinha capacidade para armazenar programas, os programadores eram compelidos a codificar as mesmas seqüências de códigos de furos a cada nova fita de papel. Não passou despercebido para uma dessas programadoras pioneiras, Grace Murray Hopper, que se poderia poupar muito esforço de programação caso se descobrisse uma forma de se reutilizar as fitas de papel codificadas para outros problemas. O Mark I não permitia a implementação desta idéia com facilidade, tendo sido preciso esperar as gerações seguintes de computadores para colocá-la em prática.

Com a evolução dos computadores, as seqüências de códigos de furos foram substituídas por lotes de cartões perfurados e a elaboração de programas tornou-se consideravelmente mais ágil: primeiramente selecionavam-se os lotes apropriados de cartões, em seguida perfuravam-se os cartões que carregavam o valor dos argumentos de entrada e armazenavam os resultados de saída dos lotes selecionados e, finalmente, agrupavam-se os cartões resultantes em um novo lote de cartões. Esta

---

[8] Em princípio, uma linguagem de programação de alto nível é uma linguagem usada para preparar programas de computador mais próxima da linguagem natural empregada pelo ser humano. Por outro lado, uma linguagem de montagem é uma linguagem mais próxima do conjunto de instruções executado pelo processador. Consulte o glossário para definições mais precisas dos termos.

seqüência de ações passou a ser conhecida como compilar um programa. No início da década de 50 do século passado, desenvolveram-se programas que automatizavam estas tarefas e que passaram a ser conhecidos como compiladores. Os primeiros compiladores desse tipo foram desenvolvidos por Grace Murray Hopper para o UNIVAC. De acordo com Hopper, um compilador era um programa que copiava o código de uma sub-rotina no local apropriado do programa principal onde se desejava executar a operação a ela correspondente. As sub-rotinas tratadas pelos compiladores eram muito específicas, restringindo-se ao cálculo de funções matemáticas, como senos, co-senos, logaritmos, etc., e operações aritméticas de ponto flutuante (Ceruzzi, 1998).

Embora o termo compilador tenha surgido para designar esses compiladores pioneiros, tais programas não lembram, contudo, o funcionamento dos compiladores atuais. O primeiro sistema de programação a funcionar como um compilador moderno foi desenvolvido por Laning e Zierler para o computador Whirlwind do MIT no início da década de 50 do século passado (Ceruzzi, 1998). Diferentemente dos compiladores UNIVAC, o sistema desenvolvido por Laning e Zierler recebia comandos digitados pelo usuário em notação algébrica e os traduzia para seqüências de instruções de máquina, que o computador executava. Além de executar os comandos digitados pelo usuário, o sistema gerenciava a memória, tratava enlaces repetitivos e desempenhava outras tarefas de manutenção do computador. O sistema proposto por Laning e Zierler se destinava à resolução de equações algébricas, mas se aproximava de um interpretador de uma linguagem de programação de propósito geral. Mesmo com toda a publicidade recebida pelo Whirlwind, o sistema de Laning e Zierler passou despercebido (Backus, 1980). Um dos motivos que refreou a aceitação do sistema de Laning e Zierler foi o fato de o sistema ser dez vezes mais lento, em média, que outros sistemas codificados manualmente para o mesmo computador (Knuth, 1980). Para que os compiladores automáticos fossem finalmente aceitos, seria necessário que o código produzido por um compilador automático fosse tão bom quanto o código produzido manualmente.

No final da década de 50 do século passado, esta lacuna foi preenchida com o surgimento do primeiro compilador para a linguagem de alto nível FORTRAN (Backus, 1967). Morgan (1998) considera o projeto do compilador FORTRAN um

marco no desenvolvimento dos compiladores, pois o código produzido pelo compilador era tão eficiente quanto o código produzido manualmente pelos programadores em linguagem de montagem, sendo, portanto, um compilador otimizador; compilava uma linguagem completa, embora o projeto da linguagem não especificasse como fazê-lo e a tecnologia para produzir tal compilador não existisse, tendo sido necessário desenvolvê-la, como foi o caso do conceito dos passos, ou fases, de compilação. Após o surgimento do compilador FORTRAN, foram desenvolvidos outros compiladores, para as linguagens de alto nível ALGOL e COBOL (Ceruzzi, 1998). O aumento da produtividade, ou seja, da quantidade de código produzida por número de linhas de programa, proporcionado pelo uso dessas linguagens consolidou a programação em linguagem de alto nível como uma alternativa viável à programação manual em linguagem de montagem e estimulou o aparecimento de novas linguagens de programação.

Com a proliferação de novas linguagens de programação a partir do início da década de 60 até meados da década de 70 do século passado (Sammet, 1969), passou-se a estudar métodos e ferramentas que agilizassem a geração de compiladores mesmo que o código produzido pelo compilador não fosse muito eficiente. Paralelamente, avanços na teoria das linguagens suscitou o desenvolvimento de técnicas eficazes, que poderiam ser aplicadas no desenvolvimento de analisadores léxicos e sintáticos desses geradores de compiladores (Grune et al., 2000). Um exemplo de tais ferramentas são o Lex (Lesk; Schmidt, 1975 apud Tremblay; Sorenson, 1985) e o Yacc (Johnson, 1975 apud Tremblay; Sorenson, 1985).

A partir de meados da década de 70 do século passado a necessidade de compiladores para novas linguagens de programação e/ou novas arquiteturas de computadores diminuiu gradativamente e a demanda por compiladores mais confiáveis e eficientes, tanto no uso quanto na qualidade do código gerado, aumentou (Grune et al., 2000), provocando o deslocamento do foco dos estudos para novas técnicas de otimização e geração de código. Alguns exemplos desse novo enfoque são: o projeto PQCC[9] conduzido por Leverett et al. (1980) que desenvolveu um

[9] *Production-Quality Compiler-Compiler*

programa de construção automática de geradores de código otimizadores inspirado no compilador otimizador para a linguagem BLISS-11 (Wulf et al., 1975) e o desenvolvimento da forma SSA[10] por Reif e Lewis (1977) que se tornou um dos métodos mais empregados na fase de otimização global dos compiladores otimizadores atuais e que foi aperfeiçoado por Cytron et al. (1991).

Com o advento dos supercomputadores e dos processadores RISC no final da década de 70 e início dos anos 80 do século passado, desenvolveram-se novas técnicas de otimização de código para usar de forma eficaz a linha de montagem de instruções[11] e disponibilizar os valores requeridos pelas instruções em execução. Para tanto, reordenou-se a seqüência de instruções original a fim de se iniciar a execução de um certo número de instruções que não dependessem do resultado da conclusão da primeira instrução. Estas técnicas foram originalmente desenvolvidas pelos projetistas de compiladores para os computadores Cray-1 (Morgan, 1998).

Ao final da década de 90 do século passado, a pesquisa de construção de compiladores se defrontou, por exemplo, com o tratamento de longas latências de memória e a geração de código para processadores paralelos, mas a estrutura e a organização dos compiladores desenvolvidos em passado recente se mostrou suficientemente flexível para que novas fases de otimização e geração de código fossem incluídas nesses compiladores (Cooper; Torczon, 2004).

O restante do capítulo mostra como a estrutura e a organização dos compiladores são responsáveis pela flexibilidade desses sistemas de programação. A princípio, pode-se decompor o processo de compilação em duas grandes partes: a análise do programa-fonte e a síntese do código-objeto conforme mostra a figura 3.1 abaixo.

---

[10] *Static single-assignment*

[11] *Instruction pipeline*

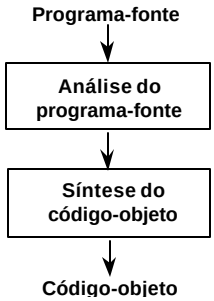

Figura 3.1 - Estrutura geral de um compilador
Adaptada de Lorho (1984)

3.1. Análise do programa-fonte

A análise do programa-fonte decompõe o texto de entrada em seus elementos básicos e o recria em uma forma de representação mais adequada para a síntese do código-objeto que se denomina forma de representação intermediária. A análise do texto de entrada, por sua vez, se subdivide em três passos: a análise léxica, a análise sintática e a análise semântica.

A análise léxica é desempenhada por um analisador léxico. O analisador léxico faz a leitura do programa-fonte na forma de uma seqüência de caracteres e produz uma seqüência de elementos aos quais se associam categorias sintáticas da linguagem de programação na qual foi escrita o programa-fonte. O analisador léxico agrupa os caracteres de entrada do programa-fonte segundo regras bem definidas e determina se cada agrupamento é um elemento léxico legal da linguagem de programação e, em caso positivo, o classifica em uma dada categoria sintática. Os elementos assim classificados são também denominados de átomos da linguagem.

A análise sintática é feita por um analisador sintático que consome os átomos provenientes do analisador léxico um após outro e determina se os átomos formam uma sentença sintaticamente válida da linguagem de programação, ou seja, um programa. Caso isso seja verdade, o programa é submetido, como se verá mais adiante na seção 3.1.3, a uma fase de análise semântica que determina se o programa tem um significado consistente (Cooper; Torczon, 2004).

Do ponto de vista lógico, estes três passos de análise são tarefas separadas. Do ponto de vista prático, porém, estas tarefas são executadas intercaladamente, como, por exemplo, na compilação dirigida por sintaxe em que o analisador sintático aciona o analisador léxico, para requerer novos átomos do programa-fonte, e o analisador

semântico, para verificar a significado das várias partes que compõe o programa-fonte, à medida em que reconhece comandos válidos da linguagem (José Neto, 1987). A figura 3.2 mostra a organização lógica destes três passos de análise.

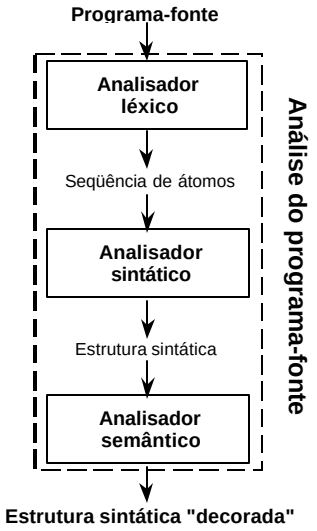

Figura 3.2 - Organização lógica da análise do programa-fonte
Adaptada de Lorho (1984)

3.1.1.Análise léxica

A análise léxica extrai e classifica átomos, freqüentemente representados como pares ordenados da forma (classe do átomo, valor do átomo), do texto do programa-fonte para o analisador sintático. Dentre as classes de átomos identificadas pelos analisadores léxicos pode-se mencionar: os identificadores, as palavras reservadas, números inteiros, números de ponto flutuante, cadeias de caracteres, sinais de pontuação e de operação, caracteres especiais, símbolos compostos por dois ou mais caracteres que delimitam, por exemplo, os comentários, etc. (José Neto, 1987).

Além desta função básica, o analisador léxico pode realizar a eliminação de delimitadores e comentários, conversões numéricas simples e algum tipo de tratamento dos identificadores. Grune et al. (2000) mencionam três propriedades das linguagens de programação atuais que requerem o tratamento dos identificadores pelos analisadores léxicos: a existência de identificadores que afetam a análise do programa-fonte adiante, o processamento de macros, que inclui a definição e expansão de macros, inclusão de arquivos e a inclusão de texto condicional, e o tratamento das palavras reservadas.

Como os analisadores léxicos são módulos funcionais do compilador, cujas funções são ativadas inúmeras vezes durante a compilação do programa-fonte (José Neto, 1987), os analisadores léxicos empregam reconhecedores especializados de padrões[12] em cadeia de caracteres que tornam o processo de análise bastante eficiente: os autômatos finitos. Os métodos para obtenção destes reconhecedores são descritos em vários textos da literatura, como (Tremblay; Sorenson, 1985), (Aho et al., 1986), (José Neto, 1987), (Fischer; LeBlanc, 1991), (Grune et al., 2000), (Cooper; Torczon, 2004).

3.1.2.Análise sintática

A análise sintática verifica se o fluxo de átomos proveniente do passo de análise léxica forma um programa válido da linguagem de programação e constrói um modelo concreto do programa para uso nas próximas fases do processo de compilação, em geral uma árvore sintática abstrata. A fase seguinte analisa a árvore e inclui informações semânticas na mesma, originando uma árvore sintática decorada. Caso a seqüência de átomos não forme um programa válido, o analisador sintático deve informar o usuário da presença de erros juntamente com informações que auxiliem na sua correção.

Nem todas as implementações constróem fisicamente a árvore sintática abstrata, contudo, o princípio de construção da mesma se manifesta através da seqüência de derivações ou reduções que conduzem à obtenção do código-objeto (José Neto, 1987). Os motivos que provocaram a construção da árvore sintática em separado foram o interesse em se obter geradores de compiladores[13] e compiladores capazes de gerar código-objeto para diferentes arquiteturas de computadores, ou compiladores redirecionáveis[14]. Graham (1984) menciona que a fragmentação da síntese do código-objeto em uma seqüência de passos mais elementares reduz a complexidade do processo de compilação a ponto de se aplicarem geradores de compiladores a cada um deles e considera, também, que compiladores redirecionáveis têm a vantagem de que a interface com a linguagem de programação,

---

[12] *Patterns*

[13] *Compiler-compilers*

[14] *Retargetable compilers*

ou seja, a análise léxica, sintática e semântica, permanece praticamente inalterada enquanto que as fases dependentes da arquitetura do computador são, desde que assim projetadas, relativamente simples de alterar.

Os dois principais métodos empregados na análise sintática são determinísticos, sendo a duração da análise linear em relação ao comprimento do programa. Contudo, não se pode empregá-los na análise sintática de qualquer linguagem livre de contexto, classe das linguagens às quais pertencem as linguagens de programação que são descritas por um formalismo denominado gramática[15] livre de contexto, apenas em um subconjunto restrito das gramáticas livre de contexto formado pelas gramáticas LL(k) e LR(k) que se discutem mais adiante (Grune et al., 2000).

Caso uma linguagem livre de contexto não possa ser descrita por uma das duas gramáticas para as quais se conhecem reconhecedores determinísticos, ainda sim a teoria das linguagens assegura a existência de reconhecedores não determinísticos que a reconheçam (Lewis; Papadimitriou, 1981), porém a duração da análise dos algoritmos de reconhecimento resultantes perde a propriedade de ser linear em relação ao comprimento do programa e, na melhor das hipóteses, passa a ser polinomial de ordem 3 em relação ao comprimento do programa (Aho et al., 1972). Além da duração da análise, sabe-se que se é possível a construção de um reconhecedor determinístico para uma certa gramática livre de contexto, então a gramática não é ambígua (Aho et al., 1972, 1986), (José Neto, 1987), (Grune et al., 2000). Sendo a não ambigüidade uma propriedade requerida nas linguagens de programação, embora isto não implique a existência de um reconhecedor determinístico, a determinação da existência de um reconhecedor determinístico é o melhor teste de não ambigüidade disponível para uma linguagem de programação (Grune et al., 2000).

Em uma árvore sintática abstrata, a raiz e os nós interiores da árvore representam comandos ou operações e as folhas representam os operandos. A figura 3.3 abaixo mostra um exemplo de árvore sintática abstrata de cálculo do valor da

---

[15] Uma gramática é um formalismo empregado na descrição das sentenças válidas de uma linguagem a partir do seu alfabeto. Para uma definição formal, consultar a literatura sobre teoria de linguagens (Aho et al., 1972), (Lewis; Papadimitriou, 1981) ou sobre projeto e técnicas de construção de compiladores (Aho et al., 1986), (José Neto, 1987) ou (Grune et al., 2000), por exemplo.

expressão aritmética $b^2 - 4ac$ e atribuição do mesmo à variável *d* pelo comando de atribuição em linguagem C: 'd = b*b - 4*a*c;'.

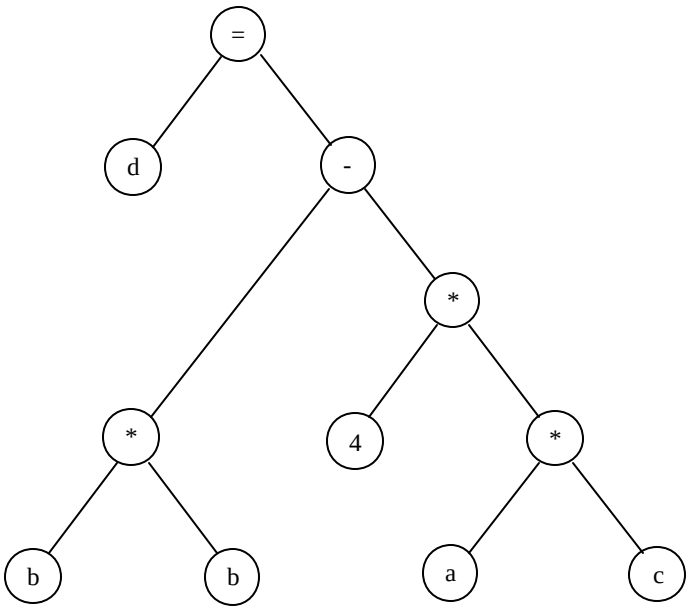

Figura 3.3 - Árvore sintática abstrata do comando d = b*b - 4*a*c
Adaptada de Grune et al. (2000)

Os métodos de análise sintática se destacam pela forma como constróem a árvore sintática abstrata: no método descendente[16] o reconhecedor constrói a árvore a partir da raiz em direção às folhas da árvore e no método ascendente[17] o reconhecedor constrói a árvore a partir das folhas em direção à raiz.

Os reconhecedores descendentes se aplicam às linguagens livre de contexto descritas pelas gramáticas LL(k). A abreviatura LL(k) indica gramáticas que permitem a construção de reconhecedores determinísticos descendentes que analisam os átomos do programa-fonte da esquerda, *left*, para a direita, *right*, o primeiro L de LL(k), e expandem o nó mais à esquerda da árvore em direção às folhas[18], o segundo L de LL(k), em função do estado corrente do reconhecimento e da inspeção[19] de até *k* átomos à frente do átomo atual, o (k) de LL(k). A operação do reconhecedor leva à construção da árvore sintática abstrata da raiz em direção às folhas e da esquerda

[16] *Top-down*

[17] *Bottom-up*

[18] *Leftmost derivation*

para a direita conforme mostra a figura abaixo.

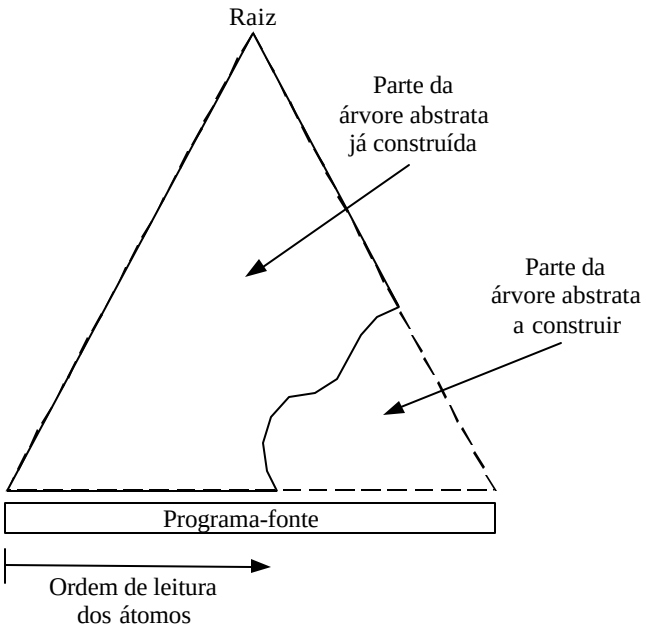

Figura 3.4 - Construção da árvore abstrata por reconhecedor descendente
Adaptada de Lewis; Papadimitriou (1981)

Pode-se implementar os reconhecedores descendentes de várias maneiras, sendo a que conduz aos reconhecedores descendentes recursivos, a mais conhecida e aplicada delas.

Os reconhecedores ascendentes se aplicam às linguagens livre de contexto descritas pelas gramáticas LR(k). A abreviatura LR(k) indica gramáticas que permitem a construção de reconhecedores determinísticos ascendentes que analisam os átomos do programa-fonte da esquerda para a direita, o L de LR(k), e deduzem o nó da árvore em direção à raiz pela determinação da derivação mais à direita a ser aplicada[20], o R de LR(k), em função do estado corrente do reconhecimento e da inspeção de até *k* átomos à frente do átomo atual, o (k) de LR(k). A operação do reconhecedor leva à construção da árvore sintática abstrata das folhas em direção à raiz e da esquerda para a direita conforme mostra a figura abaixo.

---

[19] *Look ahead*

[20] *Rightmost derivation*

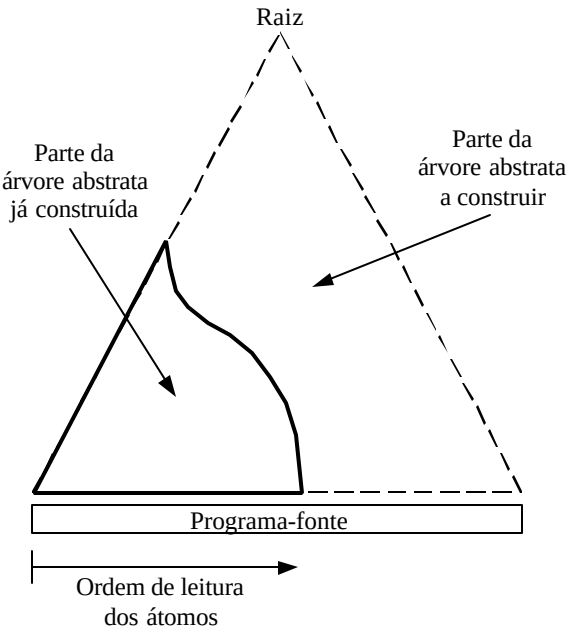

Figura 3.5 - Construção da árvore abstrata por reconhecedor ascendente
Adaptada de Lewis; Papadimitriou (1981)

Existe uma certa variedade de reconhecedores ascendentes: LR(k), LALR(k) e SLR(k), porém todos são implementados por geradores de analisadores sintáticos, pois o mecanismo de controle desses analisadores é difícil de ser implementado manualmente para uma linguagem de programação completa.

Os mesmos textos indicados no projeto e construção de reconhecedores para a análise léxica também descrevem minuciosamente os métodos para a obtenção de reconhecedores descendentes e ascendentes.

3.1.3.Análise se mântica

A análise sintática somente verifica a correção da estrutura do programa. Embora estejam diretamente relacionados com a estrutura da árvore sintática abstrata, certos aspectos dependentes de contexto não são expressos pelas gramáticas livres de contexto. A análise semântica verifica se os elementos da árvore abstrata satisfazem os aspectos dependentes de contexto especificados para a linguagem de programação, ou as regras semânticas da linguagem, como são as regras relacionadas com escopo e tipo, e registra informações essenciais para a síntese do código-objeto por meio de "algum tipo de decoração da estrutura sintática do programa, construída pelo analisador sintático" (Lorho, 1984). Esta estrutura sintática decorada, também

conhecida por árvore sintática abstrata decorada, é a forma de representação intermediária pela qual se inicia a síntese do código-objeto.

As regras de escopo delimitam o escopo dos identificadores, ou melhor, a região do programa-fonte na qual os identificadores declarados tem efeito, sendo a sua utilização limitada a esta região. Freqüentemente, não se pode utilizar o mesmo identificador duas vezes no mesmo escopo. Portanto, cada linguagem de programação deve definir precisamente o escopo dos identificadores declarados e como associar o uso do identificador com a correspondente declaração (Watt, 1984).

As expressões das linguagens de programação consistem de operandos e operações, sendo os identificadores e os literais os operandos mais primitivos das expressões. Também se permite, através de regras gramaticais da linguagem de programação, que novas expressões sejam operandos. As regras de tipo, ou seja, uma série de propriedades compartilhadas por todos os elementos pertencentes a um certo conjunto de valores, associam aos operandos das expressões tipos e determinam a equivalência ou compatibilidade entre operandos e inferem os tipos resultantes de cada expressão em função dos tipos dos operandos. Muitas linguagens de programação também incluem regras para a conversão implícita de valores de um tipo para o outro em função do contexto, ou coerção de tipo (Cooper; Torczon, 2004).

Como as associações feitas pelo módulo de análise semântica, ou analisador semântico, entre a ocorrência de um identificador e a sua declaração são necessárias nas fases seguintes da compilação, o analisador semântico deve preservar tais informações permanentemente. Uma das formas encontradas para preservar estas informações foi o seu armazenamento em uma tabela de símbolos.

Uma técnica alternativa, considerada por Watt (1984) superior em muitos aspectos, foi o armazenamento destas informações na própria árvore sintática abstrata. Os nós da árvore abstrata são dotados de campos de dados adicionais nos quais se armazenam as informações semânticas, ou atributos, dos nós. O armazenamento de informações na árvore sintática abstrata é denominado de decoração da árvore. Além de atributos relacionados ao escopo, a árvore pode armazenar atributos relacionados ao tipo.

O analisador semântico efetua uma varredura da árvore sintática abstrata. Ao

encontrar uma declaração, inclui uma entrada para aquela declaração na tabela de símbolos. Ao encontrar uma ocorrência de uso do identificador, procura, de acordo com as regras de escopo da linguagem, o identificador na tabela de símbolos e decora a árvore sintática com, por exemplo, o endereço do identificador se o mesmo estiver presente, ou indica a ocorrência de um erro se o identificador não for localizado na tabela. Um procedimento similar se aplica para as regras de tipo. A figura 3.6 mostra a árvore sintática abstrata decorada da figura 3.3.

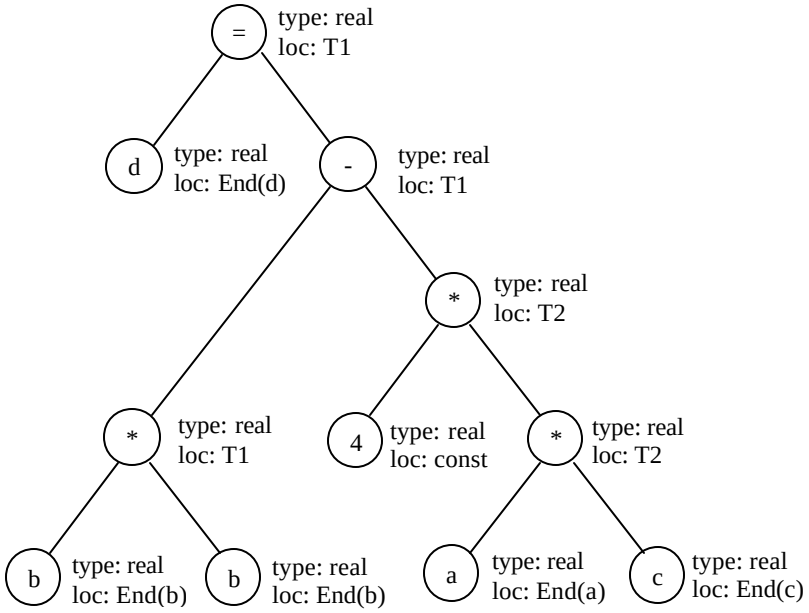

Figura 3.6 - Árvore sintática abstrata decorada
Adaptada de Grune et al. (2000)

A obtenção de módulos de análise semântica a partir de uma especificação de alto nível, como a gramática de atributos, não é tão difundida quanto a obtenção de analisadores léxicos e de analisadores sintáticos a partir de gramáticas regulares e gramáticas livres de contexto, respectivamente. A dificuldade para a realização de cálculos globais e a necessidade de varreduras periódicas da árvore abstrata para descobrir respostas a questões simples são alguns dos obstáculos que desencorajam a adoção desta idéia (Cooper; Torczon, 2004). Grune et al. (2000) indicam que os ganhos proporcionados pela adoção de métodos automáticos ainda são pequenos em relação ao que se pode obter manualmente. (Tremblay; Sorenson, 1985), (Aho et al., 1986), (Fischer; LeBlanc, 1991), (Grune et al., 2000) e (Cooper; Torczon, 2004) são alguns dos textos que estudam a gramática de atributos e técnicas elaboradas

manualmente para a realização de ações semânticas intercaladas com o processo de análise sintática. Watt (1984) faz um apanhado dos aspectos dependentes de contexto mais comuns exibidos pelas linguagens de programação e mostra como podem ser eficientemente verificados por uma analisador semântico obtido manualmente; mostra também como elaborar formalmente os aspectos dependentes de contexto através da gramática de atributos e compara avaliadores de atributos obtidos automaticamente com analisadores semânticos obtidos manualmente. Finalmente, Lorho (1984) contém uma coletânea de artigos que tratam desde a formalização da gramática de atributos e métodos de avaliação de atributos até a obtenção automática de avaliadores de atributos.

3.2. Síntese do código-objeto

Esta é a parte do processo de compilação que mais distingue um compilador otimizador e um compilador não otimizador. É na síntese do código-objeto de um compilador otimizador que se encontram uma ou mais fases adicionais de otimização de código, cuja função é melhorar a qualidade do código-objeto, em relação à daquele produzido por um compilador não otimizador. A síntese do código-objeto também pode ser subdividida em várias etapas, ou fases, em que a otimização e a geração de código se intercalam. As grandes fases que compõem a síntese do código-objeto são: expansão de código, otimização global, geração de código e otimização do código-objeto.

Na fase de expansão de código, ou fase de geração do código intermediário, os comandos específicos da linguagem representados na árvore sintática abstrata decorada são traduzidos em instruções, ou seqüência de instruções, para certas classes de arquiteturas de computadores, que se denomina código intermediário.

Na fase de otimização global, o código intermediário sofre transformações que independem da arquitetura do computador. Para isso, exige-se a coleta de informações de grandes trechos do código intermediário, daí a denominação de otimização global, em contraste com as transformações que podem ser processadas separadamente, somente com o auxílio de informações levantadas localmente em vários pontos do código (Morel, 1984).

Na fase de geração de código é processada a geração do código na linguagem

de programação da máquina, ou código-objeto. Selecionam-se as instruções do código-objeto que correspondem às instruções do código intermediário e, opcionalmente, a alocação de registradores[21], isto é, o mapeamento das variáveis temporárias utilizadas no código intermediário para os registradores do computador, caso o escalonamento das instruções[22] não seja necessário.

Na fase de otimização do código-objeto, o código gerado pela fase anterior sofre transformações altamente dependentes da arquitetura do computador, tem a ordem de execução das instruções do código-objeto alterada visando utilizar a linha de montagem de instruções da forma mais eficiente possível e os registradores alocados. A figura 3.7 mostra uma possível organização da síntese do código-objeto em torno dessas quatro fases.

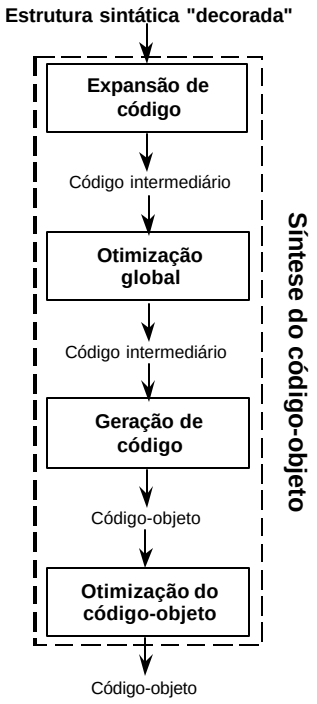

Figura 3.7 - Organização da síntese do código-objeto
Adaptada de Lorho (1984)

[21] *Register allocation*

[22] *Instruction scheduling*

Como as fases de otimização transformam alguma forma de representação para a mesma forma de representação, diferentemente das fases de análise que traduzem de uma forma de representação para outra, isto torna típica a ausência de interdependência entre as partes componentes da síntese do código-objeto (Fischer; LeBlanc, 1991). A organização apresentada na figura 3.7 é uma das organizações geralmente encontradas na literatura (Graham, 1984), (Tremblay; Sorenson, 1985), (Bacon et al., 1994), (Muchnick, 1997), (Grune et al., 2000).

As fases de síntese do código-objeto de compiladores otimizadores para arquiteturas de computadores que não necessitam de escalonamento das instruções também podem ser organizadas de acordo com a figura 3.8 abaixo. (Aho et al., 1986) e (Fischer; LeBlanc, 1991) adotam este tipo de organização. Nela, além da seleção das instruções do código-objeto que correspondem às instruções do código intermediário, a fase de geração de código realiza a alocação de registradores e algum tipo de transformação otimizante dependente da arquitetura do computador.

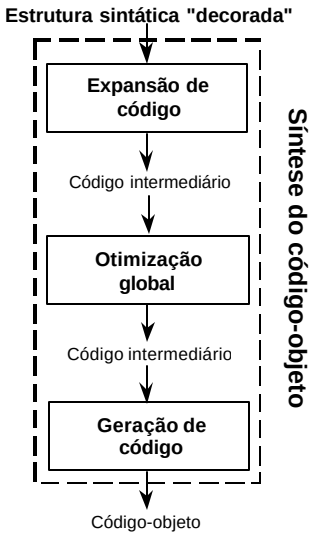

Figura 3.8 - Organização alternativa da síntese do código-objeto (1)

Finalmente, é possível aglutinar as fase de geração de código e parte das funções da fase de otimização do código-objeto da figura 3.7 em uma única fase de geração de código e executar em outra fase o escalonamento das instruções e a alocação de registradores. Neste tipo de organização, as fases de síntese do código-objeto se organizam de acordo com a figura 3.9 abaixo. Esta concepção é adotada nos textos de projeto e construção de compiladores mais recentes (Morgan, 1998) e

(Cooper; Torczon, 2004).

Este tipo de organização é resultado da interdependência entre o escalonamento das instruções e a alocação de registradores. Em outras palavras, se o compilador reordena as instruções visando diminuir o tempo de execução, pode requerer um número maior de registradores para manter valores temporários. Por outro lado, se o compilador efetua a alocação de registradores antes do escalonamento das instruções, os ganhos auferidos pelo reordenamento das instruções podem ser limitados. Isto é conhecido como problema de ordenamento das fases[23] (Morgan, 1998). Neste caso, a iteração do processo pode ser necessária para a obtenção de uma solução satisfatória por meio de aproximações sucessivas.

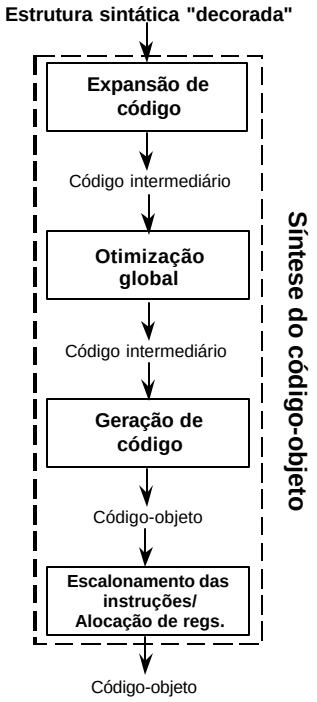

Figura 3.9 - Organização alternativa da síntese do código-objeto (2)

[23] *Phase-ordering problem*

3.2.1.Expansão de código

Nesta fase do processo de compilação a varredura da árvore sintática abstrata decorada produz uma seqüência de instruções em alguma forma de representação intermediária, ou código intermediário, para um modelo idealizado da máquina real responsável pela execução do código-objeto, também conhecido como máquina virtual. Embora seja um modelo idealizado, a máquina virtual impõe algumas condições sobre o ambiente de execução da máquina real, tais como: forma de representação de matrizes na memória; convenção de chaveamento de contexto; forma de alocação de memória dinâmica etc. A máquina virtual combina instruções, ou seqüência de instruções, em alguma forma de representação intermediária, e aciona rotinas de apoio do ambiente de execução, que provocam a correta execução dos comandos do programa-fonte na máquina real.

Fischer e LeBlanc (1991) consideram que a utilização de uma máquina virtual apresenta as seguintes vantagens: (1) separa as operações de alto nível do código intermediário da sua implementação física, possivelmente dependente de máquina; (2) como a geração de código e a atribuição de variáveis temporárias aos registradores da máquina real são claramente separadas das rotinas semânticas, que fazem interface somente com a máquina virtual, as dependências de máquina ficam confinadas aos estágios finais do processo de compilação e (3) como ao menos parte da otimização do código pode ser feita já no código intermediário, grande parte das otimizações de código tornam-se independentes da máquina real, o que facilita a elaboração de rotinas de otimização complexas e a sua reutilização em novos projetos de compiladores já que as formas de representação intermediária são mais abstratas e uniformes que as instruções da máquina real.

Existem várias formas de representação intermediária. A forma mais simples de representação talvez seja a *notação polonesa reversa*, ou *notação pós-fixa* (Fischer; LeBlanc, 1991). Nela, os operadores aparecem após os operandos aos quais se aplica a operação. A figura 3.10 ilustra a correspondência entre o comando da figura 3.3 e a sua representação em notação pós-fixa.

Comando	Notação pós-fixa
d = b*b - 4*a*c	d b b * 4 a c * * - =

Figura 3.10 - Exemplo de representação pós-fixa

Os principais atrativos da notação pós-fixa são a simplicidade e a concisão da representação, o que a torna útil como forma de representação para o acionamento de um interpretador. A notação pós-fixa não se mostra eficaz como forma de representação intermediária para otimização ou geração de código, a menos que a máquina real possua arquitetura de pilha, o que na prática raramente ocorre.

A próxima classe de representação intermediária é chamada por Fischer e LeBlanc (1991) como *códigos com três endereços*[24]. As formas de representação desta classe assemelham-se a linguagens de montagem generalizadas para máquinas virtuais com três endereços. Cada instrução consiste de um operador e três endereços, dois para os operandos e um para o resultado da operação. As *triplas* e as *quádruplas* são os membros mais conhecidos desta classe. A principal diferença entre estas representações e a notação pós-fixa é a introdução de referências explícitas ao resultados intermediários, enquanto na notação pós-fixa estes resultados correspondem implicitamente ao topo de uma pilha. A diferença entre as triplas e as quádruplas está na referência aos resultados intermediários. Nas triplas, os resultados intermediários são referenciados por um número que identifica a tripla que o gerou, enquanto nas quádruplas são referenciados explicitamente através de uma variável temporária. A figura 3.11 mostra a representação equivalente do comando da figura 3.3 em triplas e quádruplas.

Comando: d = b*b - 4*a*c

Triplas	Quádruplas
(1) (*, b, b)	(1) (*, b, b, t1)
(2) (*, 4, a)	(2) (*, 4, a, t2)
(3) (*, (2), c)	(3) (*, t2, c, t3)
(4) (-, (1), (3))	(4) (-, t1, t3, t4)
(5) (=, (4), d)	(5) (=, t4, d, -)

Figura 3.11 - Exemplo de representação de códigos com três endereços (1)

[24] *Three address codes*

As triplas são mais concisas, mas a referência a triplas previamente criadas gera uma dependência que dificulta otimizações que envolvem a movimentação e a remoção de partes do código intermediário. As duas formas contêm a mesma informação que a notação pós-fixa, mas são mais convenientes para a fase de geração de código. Uma vez obtido o código intermediário a partir da varredura da árvore abstrata, pode-se visualizar a fase de geração de código como um processo de expansão de macros, em que se consideram as referências às variáveis armazenadas na memória ou variáveis temporárias como argumentos de entrada para as macros que representam cada um dos operadores. Contudo, as triplas e as quádruplas isoladamente não possuem todas as informações semânticas necessárias para gerar código por meio da expansão de macros. No caso do exemplo da figura 3.11 faltam informações relativas aos endereços e tipos das variáveis e possíveis coerções de tipos. Se estas informações fossem recuperadas da árvore sintática decorada da figura 3.6, obteríamos os resultados mostrados na figura 3.12.

Comando: d = b*b - 4*a*c

Triplas	Quádruplas
(1) (MULTF, End(b), End(b))	(1) (MULTF, End(b), End(b), t1)
(2) (FLOAT, 4, -)	(2) (FLOAT, 4, t2, -)
(3) (MULTF, (2), End(a))	(3) (MULTF, t2, End(a), t3)
(4) (MULTF, (3), End(c))	(4) (MULTF, t3, End(c), t4)
(5) (SUBF, (1), (4))	(5) (SUBF, t1, t4, t5)
(6) (ASSIGN, (5), End(d))	(6) (ASSIGN, t5, End(d), -)

onde:
MULTF = multiplicação em ponto flutuante
FLOAT = extensão de número inteiro p/ ponto flutuante
SUBF = subtração em ponto flutuante
ASSIGN = atribuição do 1o. argumento ao 2o. argumento, sendo o tamanho do 1o. argumento ajustado ao tamanho do 2o. argumento

Figura 3.12 - Exemplo de representação de códigos com três endereços (2)

O número de operandos permitidos por triplas e quádruplas não é ideal para todos os tipos de operação. Por exemplo, o operador de atribuição para as quádruplas das figuras 3.11 e 3.12 possui um argumento não utilizado, um salto incondicional necessita de um único operando nas duas notações e assim por diante. Portanto, foi conveniente generalizar o conceito de triplas e quádruplas a fim de permitir *tuplas* com um número variável de argumentos em função do tipo de operação. A figura 3.13 mostra o exemplo da figura 3.12 em notação de tuplas.

Finalmente, pode-se usar como forma de representação intermediária a própria árvore sintática abstrata decorada. Neste caso, as transformações que independem da arquitetura do computador poderiam ser implementadas através de uma varredura da árvore que efetuasse as transformações necessárias da mesma; uma outra varredura da árvore poderia gerar código diretamente, ou produzir uma representação mais adequada para a realização de transformações dependentes da arquitetura da máquina real. Entretanto, Morgan (1998) ressalta que a utilização de tuplas facilita a delimitação de blocos básicos, para a realização das transformações que independem da arquitetura do computador, uma vez que os pontos de controle do fluxo de execução aparecem explicitamente na representação intermediária.

Comando: d = b*b - 4*a*c

Tuplas
(1) (MULTF, End(b), End(b), t1)
(2) (FLOAT, 4, t2,)
(3) (MULTF, t2, End(a), t3)
(4) (MULTF, t3, End(c), t4)
(5) (SUBF, t1, t4, t5)
(6) (ASSIGN, t5, End(d))

Figura 3.13 - Exemplo de representação de tuplas

Um dos aspectos a ser considerado no ambiente de execução é a alocação física dos identificadores criados no programa-fonte. Graham (1984) considera que as linguagens de programação (imperativas ou orientadas a objetos) típicas requerem a criação de três tipos de objetos[25]. O primeiro está relacionado com valores que devem existir durante toda a execução do programa, sendo as variáveis globais ou estáticas exemplos característicos. O compilador deve providenciar o armazenamento permanente destes elementos em tempo de execução. O segundo refere-se aos valores cuja existência está limitada a determinados escopos de execução do programa, como as variáveis declaradas no interior de funções e procedimentos. O armazenamento requerido deve ser providenciado antes do início da fase de execução, e liberado ao seu término. Se o espaço por eles ocupado é conhecido em tempo de compilação, então o espaço total necessário para seu armazenamento pode

[25] Não confundir instância de uma classe, o objeto, das linguagens orientadas a objetos, com um objeto que designa algum elemento, ou item, concreto ou abstrato. O termo é aqui utilizado nesta segunda conotação.

ser reservado e liberado de uma só vez. O terceiro tipo de elemento está relacionado com valores cuja existência não se encaixa nos padrões estabelecidos para os dois primeiros tipos, sendo a reserva e liberação de espaço para tais entidades implementada através da ativação de rotinas de gerenciamento de memória dinâmica. A área de memória na qual se faz a reserva e liberação de espaço de memória em tempo de execução é usualmente denominada de *heap*.

Outro aspecto a ser considerado é a pilha de controle de execução[26]. Como se viu anteriormente, o espaço total necessário para o armazenamento de objetos declarados no interior de funções e procedimentos pode ser reservado e liberado em bloco. Como estes objetos são referenciados da mesma forma que os campos de um registro, este bloco de armazenamento é também chamado de registro de ativação[27].

Se a linguagem permite a chamada recursiva de funções e procedimentos, múltiplas instâncias do mesmo registro de ativação poderão ser criadas em tempo de execução. Como os escopos de execução são aninhados de tal forma que o último escopo a ser executado seja o primeiro a ser encerrado, os registros de ativação são armazenados em uma pilha, denominada pilha de controle de execução, ou pilha de controle. Ainda, se a linguagem permite a execução de processos concorrentes, pode haver necessidade de mais de uma pilha de controle de execução.

Além dos campos reservados para os elementos, criados no programa-fonte, cuja existência é limitada, o registro de ativação contém outros campos. Um deles é usado para devolver o controle de execução para o local originador da chamada da função ou procedimento, após o término da sua execução. Outros campos são usados no endereçamento de outros registros de ativação na pilha de controle.

Sendo necessária, portanto, uma pilha de controle, o compilador deve gerar uma variável global que aponte para o registro de ativação da pilha de controle que corresponde ao escopo atual em execução e que se denomina apontador da pilha[28]. Quando o escopo é encerrado, deve-se ajustar o apontador da pilha para que passe a apontar o registro de ativação do escopo originador da chamada da função ou

---

[26] *Runtime stack*

[27] *Activation record*

[28] *Stack pointer*

procedimento. Por esta razão, um registro de ativação pode conter um elo dinâmico[29] que aponta o registro de ativação do escopo originador da chamada. Caso a linguagem adote regras de escopo léxico para disciplinar o acesso a elementos fora do escopo em execução, um registro de ativação pode conter um elo estático[30] que aponta o registro do escopo que engloba o aninhamento mais interno.

O código gerado pelo compilador responsável pela alocação de um novo registro de ativação, ajuste do apontador de pilha, iniciação dos elos dinâmico e estático, armazenamento de registradores na pilha etc. que prepara a execução de um escopo, e que cuida do retorno à situação inicial ao seu término, efetua o chaveamento de contexto. O conjunto de regras de chaveamento de contexto definidas pela máquina virtual são denominadas convenções de chaveamento de contexto.

(Tremblay; Sorenson, 1985), (Aho et al., 1986), (Fischer; LeBlanc, 1991), (Muchnick, 1997), (Grune et al., 2000) e (Cooper; Torczon, 2004) são textos que tratam as formas de representação intermediária e as funcionalidades requeridas pelos ambientes de execução em mais detalhes. Graham (1984) faz uma série de considerações sobre as funcionalidades requeridas pelas diversas fases da síntese do código-objeto, trata com detalhes a fase de geração de código e relaciona referências relevantes, especialmente nas fases de geração de código e otimização do código-objeto.

3.2.2.Otimização global

Nesta fase do processo de compilação, procura-se aprimorar o código intermediário resultante de uma expansão canônica de código. Note-se que, mesmo um código otimizado, raramente pode ser considerado ótimo em algum sentido (Graham, 1984). A otimização de código não é capaz de compensar a má elaboração de um programa. De fato, nenhuma técnica conhecida pode compensar o principal componente de um programa mal elaborado: uma escolha mal feita do algoritmo adotado. Pelo contrário, a otimização de código encoraja as boas práticas de

---

[29] *Dynamic link*

[30] *Static link*

programação, pois torna as linguagens de programação mais atrativas e cuida de pequenos detalhes que afetam a eficiência da execução, fazendo com que o projetista se concentre na resolução do problema (Kennedy, 1981).

Nesta etapa, deve-se localizar os trechos do código intermediário nos quais uma alteração pode aprimorar o código resultante e determinar em quais deles a realização de uma alteração é segura, isto é, preserva as características do programa original. Os otimizadores de código empregam alguma forma de análise estática para reunir essas informações.

Neste tipo de análise, o otimizador tenta estimar o resultado da execução do código em tempo de compilação[31]. Por exemplo, se o otimizador obtém uma informação precisa sobre quais valores previamente calculados estão disponíveis em um dado trecho do código, pode substituir o cálculo de uma expressão pela atribuição imediata do resultado previamente calculado (Cooper; Torczon, 2004)[32].

Uma das formas mais conhecidas de análise estática é a análise do fluxo de dados[33]. A análise do fluxo de dados extrai, do grafo de fluxo de controle[34] do código intermediário, informações sobre o fluxo de valores que pode ocorrer em tempo de execução. A partir dessas informações, o otimizador pode decidir em quais trechos do código uma dada otimização é aplicável e segura. A análise é feita pela solução de um conjunto de equações de fluxo aplicadas ao grafo de fluxo de controle, e compreende três passos: (1) a construção do grafo de fluxo de controle, (2) a coleta de informações locais para a resolução das equações de fluxo e (3) a solução das equações de fluxo para o grafo de fluxo de controle do código.

A construção do grafo de fluxo de controle envolve a subdivisão do código intermediário em blocos básicos[35], ou seja, em seqüências consecutivas de instruções que sempre são executadas do início até o fim, e a instanciação de nós e arestas do

---

[31] Para efetuar as transformações otimizantes na análise estática, o otimizador não faz quaisquer suposições sobre os valores assumidos pelos objetos em tempo de execução; por outro lado, na análise dinâmica, o otimizador se aproveita de informações obtidas em tempo de execução para efetuar as transformações otimizantes.

[32] Este é um exemplo de eliminação de subexpressões comuns globais.

[33] *Data-flow analysis*

[34] *Control-flow graph*

[35] *Basic blocks*

grafo de fluxo de controle que representam blocos e transições entre blocos, respectivamente.

Em seguida, efetua-se uma varredura de cada bloco básico, e se coletam os dados relevantes do bloco para a resolução das equações de fluxo. Por fim, resolvem-se as equações de fluxo através da propagação das informações, obtidas no passo anterior, por todo o grafo de fluxo de controle.

Uma das formas de solução das equações de fluxo mais estudadas é conhecida como o método iterativo. Neste método, para cada nó do grafo, calcula-se, com as informações disponíveis até o momento, o resultado das equações de fluxo. A iteração dos nós do grafo pára quando o resultado das equações de fluxo deixa de sofrer alterações, ou seja, quando converge para uma solução.

Algumas das transformações otimizantes que empregam a análise do fluxo de dados são: *eliminação de subexpressões comuns globais*[36] que envolve a localização e a eliminação de expressões que calculam valores já disponíveis, *avaliação e propagação de constantes globais*[37] que substitui o uso de variáveis às quais se atribuiu um valor constante pelo próprio valor constante e *movimentação de código*[38] que movimenta para fora de enlaces repetitivos as instruções que não dependem da iteração do enlace ou reduz o espaço ocupado pelo código através da fatoração de seqüências de instruções redundantes.

Outras transformações importantes que não empregam análise do fluxo de dados, mas que operam diretamente sobre o grafo de fluxo de controle são: *eliminação de código inútil e inatingível*[39], que remove seqüências de instruções cujo resultado não é usado ou para as quais não exista um caminho válido no grafo de fluxo de controle e *substituição de operações complexas por outras equivalentes mais simples*[40] que substitui uma seqüência repetida de operações custosas por uma seqüência equivalente de operações de custo mais baixo que desempenham a mesma função. Para um catálogo mais completo de transformações otimizantes que podem

---

[36] *Global common subexpression elimination*

[37] *Global constant folding and propagation*

[38] *Code motion*

[39] *Useless and unreachable-code elimination*

[40] *Strength reduction*

ser aplicadas nesta fase, o leitor interessado pode consultar, por exemplo, (Allen; Cocke, 1972) ou (Muchnick, 1997).

Desde o surgimento da análise de fluxo de dados, no início da década de 60 do século passado, diversos problemas foram solucionados. Se cada transformação utilizar um passo de análise diferente, o esforço para implementar, depurar e manter um otimizador global pode se tornar proibitivo. Seria desejável fazer múltiplas transformações em um único passo de análise.

O grafo de atribuições singelas estáticas, ou SSA[41], tem essa propriedade, pois codifica em uma única estrutura as informações de fluxo de dados e de controle do código intermediário. Esta denominação resulta do fato de um único identificador ser definido para cada uma das operações presentes no código. Para reconciliar as diversas atribuições singelas estáticas com as variáveis que controlam o fluxo de controle, inserem-se operações especiais, denominadas funções-ϕ, nos pontos de encontros das arestas do fluxo de controle. A partir de uma única implementação que traduza o código intermediário para a forma SSA, podem-se realizar muitas das transformações apresentadas no parágrafo anterior (Cooper; Torczon, 2004).

(Tremblay; Sorenson, 1985), (Aho et al., 1986), (Fischer; LeBlanc, 1991), (Morgan, 1998) e (Cooper; Torczon, 2004) são textos de projeto e construção de compiladores que descrevem a análise do fluxo de dados e métodos iterativos de solução das equações de fluxo. (Muchnick, 1997) é um texto avançado que descreve em detalhes os principais métodos de solução das equações de fluxo. (Kennedy, 1981) faz uma breve introdução à análise do fluxo de dados, descreve vários métodos de resolução das equações de fluxo e mostra como esta técnica pode ser utilizada na fase de otimização global. (Morel, 1984) descreve a análise de fluxo de dados e, através de propriedades extraídas do grafo de fluxo, mostra como obter soluções integradas para a eliminação de código redundante e a movimentação de código. (Cytron et al., 1991) é o artigo que descreve como traduzir código intermediário para a forma SSA. (Muchnick, 1997), (Morgan, 1998) e (Cooper; Torczon, 2004) são textos recentes que estudam didaticamente a forma SSA.

---

[41] *Static single-assignment*

3.2.3.Geração de código

Após a fase de otimização global, o código intermediário está pronto para ser convertido no código a ser executado pela máquina real. A fase de geração de código efetua um mapeamento das instruções, da forma de representação intermediária para o conjunto de instruções da máquina. Nesta etapa, deve-se considerar a arquitetura do conjunto de instruções e os recursos de hardware presentes na máquina. Se a forma de representação intermediária contempla todos ou, ao menos, parte dos recursos da arquitetura da máquina, a geração de código é facilitada. Caso a forma de representação intermediária represente a máquina real em um nível de abstração mais elevado, a geração de código deve se encarregar de preencher esses detalhes. Os compiladores que realizam pouca ou nenhuma otimização de código executam a geração de código a partir da árvore sintática abstrata decorada ou em conjunto com a fase de análise semântica (Cooper; Torczon, 2004).

Em um computador típico, a complexidade do processo de geração de código se acentua por existirem diferentes maneiras de se efetuar uma dada operação. Isto é agravado pela existência de instruções de alto nível que substituem seqüências inteiras de instruções mais elementares e múltiplos modos de endereçamento (ver, por exemplo, o apêndice B). Embora estas características permitam a criação de programas mais eficientes, aumentam o número de decisões que o procedimento de geração de código deve tomar, ampliando também o espaço de implementações em potencial.

Tendo em vista esta dificuldade e o interesse na obtenção de compiladores redirecionáveis, desenvolveram-se heurísticas para se pesquisar o espaço de seqüências de instruções do código-objeto que correspondem às instruções do código intermediário. Estas técnicas limitam o espaço de busca, ou coletam antecipadamente informações que tornem mais eficiente uma busca em profundidade.

Uma das técnicas utilizadas na geração de código aplica regras de reescrita de árvores[42] e busca de padrões de árvores[43]. É indicada para as formas de representação intermediária do tipo árvore. Primeiramente, elaboram-se regras de reescrita de

---

[42] *Tree-rewriting rules*

[43] *Tree-pattern matching*

árvore para cada instrução do código-objeto. Uma regra de reescrita é a associação formada por um padrão de árvore e um valor de substituição. Um padrão de árvore é a descrição da operação realizada por uma instrução do código-objeto em forma árvore sintática abstrata decorada. A raiz da árvore contém o código de operação da instrução e as folhas contêm os operandos de origem da instrução. O valor de substituição contém o operando de destino da instrução. Um reconhecedor de padrões aplica às subárvores do código intermediário os padrões das regras de reescrita de árvore que substituam uma subárvore por um valor de substituição. Esse processo é repetido até que a árvore seja reduzida a um único nó. Ao término desse processo, o reconhecedor de padrões terá descoberto a seqüência de regras de reescrita que reduz a árvore do código intermediário. De posse dessa informação, um segundo passo de execução reproduz a seqüência de regras de reescrita, emitindo o código associado a cada regra. Surgiram vários métodos eficientes para implementar o reconhecedor de padrões. Todos associam custos às regras de reescrita a fim de produzir a seqüência mínima de regras que reduz a árvore. Diferem na tecnologia de busca empregada e/ou no modelo de custo adotado. A linguagem TWIG (Aho et al., 1989) e o sistema de reescrita ascendente, ou BURS[44], (Pelegrí-Llopart; Graham, 1988) são dois exemplos de reconhecedores de padrões de árvores.

A partir da observação que a fase de geração de código e várias fases de otimização de código-objeto simplesmente efetuavam busca de padrões, desenvolveu-se uma técnica de geração de código com a aplicação simultânea de métodos de otimização de código (Davidson; Fraser, 1984c), (Fraser; Wendt, 1986) e (Fraser; Wendt, 1988). Esta técnica é indicada para formas de representação intermediária de códigos com três endereços. Nela, a geração de código e a técnica de otimização *peephole*, descrita no item de otimização de código-objeto, são integradas em uma única fase de geração de código na qual se implementa um sistema de reescrita de regras que efetua a busca e a substituição de padrões de regras. Os padrões de regras são divididos em vários conjuntos: um efetua o mapeamento das instruções do código intermediário para alguma linguagem de transferência de registradores, ou RTL[45], outro implementa as otimizações *peephole*

[44] *Bottom-up rewrite system*

[45] *Register transfer language*

e, um último, traduz o código RTL otimizado em código-objeto.

Existem outras técnicas de geração de código descritas na literatura que precederam as técnicas mencionadas. Muchnick (1997) descreve técnicas dirigidas por sintaxe (o método de Graham-Glanville[46]), técnicas dirigidas por semântica (o método da gramática de atributos[47]) e a linguagem TWIG (Aho et al., 1989). Cooper e Torczon (2004) descrevem o sistema de reescrita ascendente e a técnica de geração de código com a aplicação simultânea de métodos de otimização de código.

Ainda na fase de geração de código, pode-se preparar o código-objeto para o escalonamento das instruções e a alocação de registradores através das seguintes transformações (Morgan, 1998):

- Renomeação de registradores[48], que atribui nomes independentes para uma mesma variável temporária utilizada em partes diferentes do mesmo procedimento com a finalidade de aumentar a eficácia da alocação de registradores.
- Aglutinamento de registradores[49], que minimiza referências futuras aos registradores da máquina pela eliminação de operações desnecessárias de cópia de uma variável temporária para outra quando nenhuma delas sofre quaisquer modificações em qualquer caminho partindo da operação de cópia até a utilização efetiva do valor copiado. Neste caso, pode-se substituir todas as referências ao valor copiado pelo valor original e eliminar a operação de cópia, o que reduz o número de registradores necessários para armazenar variáveis temporárias.
- Extravasamento de registradores[50], que armazena na memória as variáveis temporárias do procedimento que excedem o número máximo de registradores disponíveis da máquina.

---

[46] *Graham-Glanville method*

[47] *Attribute-grammar method*

[48] *Register renaming*

[49] *Register coalescing*

[50] *Register spilling*

3.2.4.Otimização do código-objeto

Nem todas as técnicas de otimização são independentes da arquitetura do computador. Quando o código sofre transformações que o fazem mais eficiente em uma certa máquina, diz-se que o código sofreu uma otimização dependente de máquina. Nesta fase utilizam-se informações sobre os limites e as características especiais da máquina que possibilitam a produção de um código-objeto mais compacto ou uma execução mais rápida do código-objeto (Tremblay; Sorenson, 1985). Esta fase é também conhecida como fase final de otimização de código (Graham, 1984) ou de micro-otimizações (Bacon et al., 1994). As principais otimizações dependentes de máquina descritas a seguir são: a super-otimização, a otimização do fluxo de controle e a otimização *peephole*. Nesta fase, também, se efetuam o escalonamento das instruções e a alocação de registradores.

3.2.4.1. Super-otimização

Na super-otimização, dados uma seqüência de instruções, que representa uma função, e um subconjunto de instruções da arquitetura do computador, faz-se uma busca exaustiva da menor seqüência de instruções, dentre todas as seqüências possíveis do subconjunto de instruções disponível, que produza os mesmos resultados que a função original ao se aplicar o mesmo conjunto de casos de teste. Massalin (1987) foi o primeiro a propor este método de busca exaustiva; Granlund e Kenner (1992) aplicaram-na de forma limitada no GCC e Joshi et al. (2002) tornaram o processo de super-otimização mais eficiente.

3.2.4.2. Otimização do fluxo de controle

A otimização do fluxo de controle reúne um conjunto de técnicas que visam minimizar e/ou racionalizar a realização instruções de desvios, condicionais ou não, em tempo de execução. Wulf et al. (1975) é uma referência clássica sobre o projeto de compiladores otimizadores que trata de otimização do fluxo de controle; Muchnick (1997) dedica um capítulo à otimização do fluxo de controle.

3.2.4.3. Otimização *peephole*

O resultado da tradução, para linguagem de máquina, de uma forma de representação intermediária efetuada pela fase de geração de código de um compilador é o código-objeto. Devido à justaposição direta de instruções e blocos de instruções traduzidos da forma de representação intermediária para o código-objeto, a fase de geração de código freqüentemente cria seqüências ineficientes de instruções. A elaboração de uma fase de geração de código que analise todas as combinações possíveis de justaposições de blocos de instruções se torna difícil devido à existência de infinitas sentenças válidas no texto de entrada. É mais fácil manter a geração de código a cargo de um programa simples e aprimorar o código-objeto através de um novo programa, o otimizador *peephole* (Davidson; Fraser, 1980) e (Davidson; Fraser, 1984a).

A técnica de otimização *peephole* tem uma longa história na literatura, pois a publicação do primeiro trabalho foi feita há quase quarenta anos (McKeeman, 1965). O otimizador move uma pequena janela, ou fresta[51], sobre o código-objeto (McKeeman, 1965), (Wulf et al., 1975) e (Aho et al., 1986), ou sobre o código intermediário (Tanenbaum et al., 1982), e compara as seqüências de instruções lidas através da fresta com as seqüências de instruções que podem ser eliminadas ou substituídas codificadas nas regras de otimização. Uma regra de otimização se divide em duas partes: uma seqüência de busca e uma seqüência de substituição. Seja a seguinte regra de otimização para a arquitetura x86 que substitui uma seqüência de salto em igualdade e salto incondicional por uma seqüência de salto em desigualdade:

```
jeq %00
jmp %01
%00:
=
jne %01
%00:
```

O argumento '%dd', onde d é um dígito, indica uma variável que armazena a cadeia de caracteres encontrada até a ocorrência de um novo átomo, ou *token*, da regra de otimização que seja diferente de uma variável (o apêndice C apresenta uma

[51] *Peephole*

definição formal das regras). Esta notação é bastante empregada na codificação das regras de otimização (Lamb, 1981), (Davidson; Fraser, 1984a), (Davidson; Fraser, 1987), (Davidson; Whalley, 1989), (McKenzie, 1989) e (Tanenbaum et al., 1982).

3.2.4.4. Otimização *peephole* em um único passo

A eliminação ou a substituição de uma seqüência de instruções altera o código e possibilita, em geral, a aplicação de novas regras de otimização, que antes não eram aplicáveis. Por essa razão, os primeiros otimizadores *peephole* efetuavam vários passos de otimização sobre o código gerado a fim de obter o código mais otimizado possível para o programa, com as regras de otimização adotadas (McKeeman, 1965), (Wulf et al., 1975), (Aho et al., 1986) e (Tanenbaum et al., 1982).

A presença de um otimizador reduzia sensivelmente a velocidade do compilador, especialmente devido ao fato de que o otimizador precisava efetuar vários passos de otimização sobre o código gerado. Contudo, o desenvolvimento de novos algoritmos de busca e substituição reduziu essa queda de desempenho do compilador otimizador por meio da eliminação, ou redução, dos passos de otimização sobre todo o código gerado (Lamb, 1981), (McKenzie, 1989), (Kim; Oh, 1997) e (Spinellis, 1999). A figura 3.14 mostra o pseudocódigo de um algoritmo de otimização *peephole* de um único passo desenvolvido por Lamb (1981).

```
Algoritmo de otimização peephole de Lamb

Entrada: (1) uma lista ligada contendo as regras de otimização e uma lista
duplamente ligada contendo as instruções do código objeto; (2) uma pilha de
controle auxiliar onde se armazenam as instruções do código objeto; (3)
variável local fim

Saída: seqüência de instruções otimizada

início
  Prepara a pilha de controle
  Aponta a primeira instrução do código objeto
  enquanto existem instruções do código objeto a serem processadas faça
    Empilha a instrução no topo da pilha de controle
    repita
      posiciona fim := VERDADEIRO
      Procura a primeira regra de otimização aplicável do topo da pilha p/ baixo
      se existe uma regra de otimização aplicável então
        Executa a otimização
        posiciona fim := FALSO
      fim se
    até (fim ? VERDADEIRO)
    Aponta a próxima instrução do código objeto
  fim enquanto
  Desempilha lista de instruções armazenadas na pilha de controle
fim.
```

Figura 3.14 - Algoritmo de otimização *peephole* de Lamb

A figura 3.15 ilustra o funcionamento do algoritmo sobre uma seqüência de instruções A;A;B;C com a aplicação da regra de otimização A;B→B. O otimizador emprega uma pilha para o armazenamento das instruções do código-objeto e efetua a seguinte seqüência de ações após a leitura de uma nova instrução: (a) insere a instrução no topo da pilha; (b) verifica se existe uma seqüência de instruções a ser eliminada ou substituída do topo da pilha para baixo; (c) efetua a otimização, caso exista, e volta para o passo (b); do contrário (d) repete o mesmo procedimento enquanto houver instruções a serem processadas. Quando as instruções terminarem, o otimizador desempilha o conteúdo da pilha, obtendo o código otimizado.

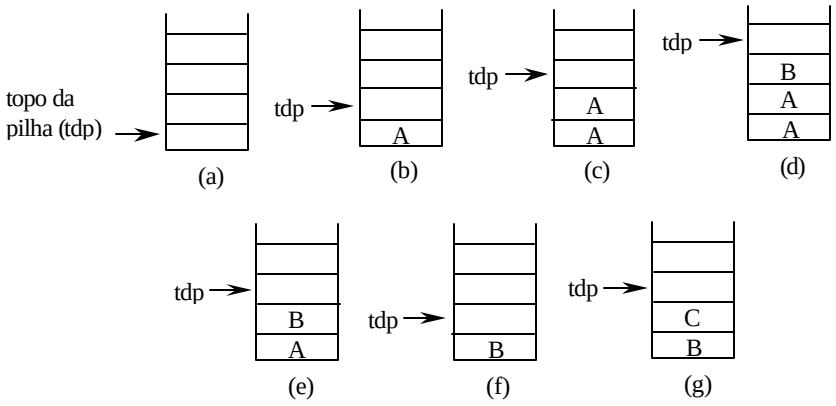

Figura 3.15 - Funcionamento do otimizador sobre a seqüência A;A;B;C: (a) no início a pilha se encontra vazia; (b) pilha após a inclusão da primeira instrução; (c) pilha após a inclusão da segunda instrução; (d) pilha após a inclusão da terceira instrução; (e) pilha após a aplicação da regra de otimização A;B→B; (f) pilha após nova aplicação da regra de otimização A;B→B e (g) pilha após a inclusão da última instrução

O primeiro aspecto que se destaca neste otimizador é a realização de vários passos de otimização sobre um trecho localizado do código, em lugar de vários passos de otimização sobre todo o código. Isto torna o otimizador mais rápido que seu antecessor direto (Wulf et al., 1975). Note-se a exploração de apenas uma das possibilidades de eliminação ou substituição quando é possível a aplicação de duas ou mais regras de otimização. O algoritmo pára a análise das instruções ao encontrar uma seqüência qualquer que possa ser eliminada ou substituída, e aplica tal modificação. Desta forma, são ignoradas todas as outras possibilidades de otimização, caso existam.

Para reduzir o número de passos de otimização *peephole* necessários, McKenzie (1989) empregou um autômato finito que analisa as instruções presentes em uma fresta de tamanho fixo. Desta análise, o autômato determina, além da

necessidade de uma otimização, o número de instruções do qual se deve deslocar a fresta de tamanho fixo. Partindo do seu estado inicial, o autômato percorre as instruções da fresta a partir de seu início até alcançar um estado final, que equivale a uma seqüência de instruções a ser eliminada ou substituída, ou até um estado não-final. Caso o autômato chegue a um estado final, efetua-se a eliminação ou substituição da instrução, ou seqüência de instruções, e o correspondente deslocamento da fresta, para que o autômato execute um novo ciclo de busca. Caso o autômato pare em um estado não-final, efetua-se apenas o deslocamento da fresta de uma ou mais instruções para a frente. Em ambos os casos, procura-se manter no interior da fresta um certo número de instruções já analisadas, que aumente a possibilidade de aplicação de uma regra de otimização no próximo ciclo de busca do autômato, e reduza o número de passos de otimização sobre todo o código. O número de instruções do deslocamento da fresta varia em função dos estados de parada do autômato. A figura 3.16 ilustra esse modelo do otimizador.

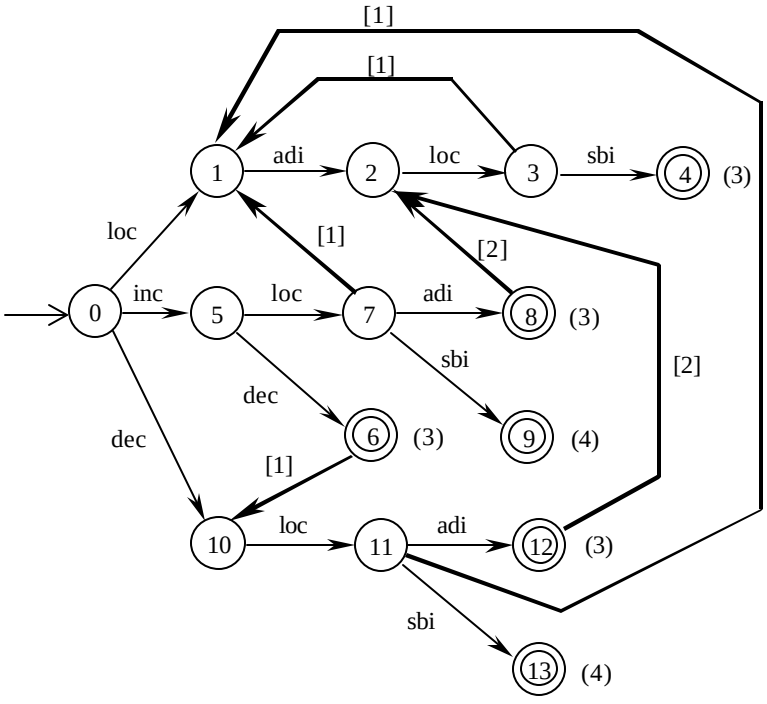

Figura 3.16 - Autômato finito construído a partir de 5 regras de otimização. Valores entre parênteses (n) indicam o deslocamento da fresta em caso de substituição. Valores entre colchetes [n] indicam o deslocamento da fresta em caso de falha. Adaptada de McKenzie (1989)

Da mesma forma que no algoritmo desenvolvido por Lamb (1981), a realização de otimizações adicionais sobre trechos localizados do código reduz a necessidade de

um passo adicional de otimização sobre todo o código[52] e resulta em um otimizador mais rápido que o seu antecessor direto (Tanenbaum et al., 1982). Por outro lado, este algoritmo não dispensa a realização de passos adicionais de otimização sobre todo o código, mas explora melhor as possibilidades de eliminação ou substituição por escolher a seqüência de instruções mais longa quando ocorre um conflito entre duas ou mais regras de otimização igualmente aplicáveis.

Seguindo esta mesma filosofia, desenvolveu-se um otimizador *peephole* usando busca e substituição de padrões de árvores, o que elimina a necessidade de passos adicionais de otimização sobre todo o código, mas com o acréscimo de fases adicionais de tradução do código para a forma de árvore e vice-versa após a otimização (Kim; Oh, 1997).

Com o intuito de tornar o otimizador mais flexível, desenvolveu-se, mais recentemente, um protótipo de otimizador *peephole* que efetua a busca e a substituição de padrões descritos por expressões regulares ao invés da busca e da substituição de cadeias fixas de caracteres, porém neste método se efetua apenas um passo de otimização, por tratar apenas uma regra de otimização (Spinellis, 1999).

3.2.4.5. Dedução automática das regras de otimização

Historicamente, a determinação das regras de otimização era efetuada manualmente, a partir da análise do código não otimizado (McKeeman, 1965), (Wulf et al., 1975) e (Lamb, 1981). Feito isso, o otimizador podia efetuar a leitura de um arquivo contendo a codificação das regras (Lamb, 1981), ou ter as regras codificadas diretamente no corpo do programa (McKenzie, 1989) e (Davidson e Whalley, 1989). Isto dificultava a elaboração de otimizadores para diferentes arquiteturas de computadores, pois era necessário deduzir novamente as regras de otimização para cada nova arquitetura. O emprego da análise do fluxo de dados possibilitou a dedução automática das regras de otimização e facilitou a elaboração de otimizadores para diferentes arquiteturas de computadores (Davidson; Fraser, 1980), (Giegerich,

[52] Como a realização de uma otimização possibilita, em geral, a aplicação de novas regras de otimização, que antes não eram aplicáveis, condiciona-se a execução de um passo adicional de otimização à realização de uma otimização no passo atual. Caso isto não ocorra, o procedimento de otimização é encerrado.

1983), (Davidson; Fraser, 1984a), (Davidson; Fraser, 1984b), (Kessler, R.R., 1984), (Kessler, P.B., 1986) e (Davidson; Fraser, 1987).

O algoritmo de análise do fluxo de dados do otimizador *peephole* PO emprega uma descrição simbólica das instruções do código-objeto para simular pares de instruções adjacentes e, quando possível, substituí-las por uma única instrução (Davidson; Fraser, 1980). Para tanto, criou-se uma gramática cujas regras de produção representem a tradução das instruções do código-objeto para a sua correspondente descrição simbólica e vice-versa. Por exemplo, a produção:

mov src,dst ↔ dst = src; NZ = (src == 0) ? 1 : 0;

descreve uma instrução que copia o conteúdo do primeiro operando no segundo operando e posiciona um indicador, que simboliza o sinal do resultado. De posse da gramática, o otimizador concatena seqüências de duas ou três instruções, calcula o resultado simbólico e procura nas regras da gramática uma instrução do código-objeto que corresponda ao resultado obtido. Caso tenha sucesso, o otimizador substitui a seqüência de instruções originais pela nova instrução. Por exemplo, o resultado simbólico da concatenação da seguinte seqüência de instruções:

mov X,r1 ↔ r[1] = m[X]; NZ = (m[X] == 0) ? 1: 0;

sub2 Y,r1 ↔ r[1]=r[1]-m[Y]; NZ=((r[1]-m[Y]) == 0)?1:0;

resulta em:

r[1] = m[X]- m[Y]; NZ = ((m[X] - m[Y]) == 0) ? 1 : 0;

que corresponde à instrução:

sub3 Y,X,r1

que substitui a seqüência de duas instruções acima. Como o otimizador deduz automaticamente as regras de otimização a partir de uma descrição formal e concisa do conjunto de instruções do código-objeto, a elaboração de otimizadores de código para diferentes arquiteturas de computadores se resume na elaboração das regras de produção da gramática que represente a tradução das instruções do código-objeto na sua correspondente descrição simbólica e vice-versa.

Empregou-se esta técnica na elaboração de compiladores otimizadores para dez diferentes arquiteturas (Davidson; Fraser, 1984b). Seguindo esta técnica, novas formas de dedução automática de regras de otimização surgiram (Giegerich, 1983), (Kessler, R.R., 1984) e (Kessler, P.B., 1986). Contudo, o desempenho desses

compiladores ficou aquém daqueles estudados anteriormente, os quais empregavam otimizadores *peephole* dedicados.

Para melhorar o desempenho, o otimizador de dedução automática passou a elaborar as regras de otimização para um outro compilador que emprega um otimizador *peephole* convencional rápido. As regras de otimização são obtidas pelo compilador de dedução automática, a partir da compilação de um conjunto de programas de treinamento, e incorporadas ao programa-fonte do compilador final (Davidson; Fraser, 1984a) e (Davidson; Fraser, 1987).

3.2.4.6. Aplicação simultânea de duas ou mais regras de otimização

Embora os otimizadores *peephole* tenham evoluído significativamente, o tratamento do conflito entre duas ou mais regras de otimização igualmente aplicáveis permaneceu pouco estudada. Um trabalho, que visava estudar o número de regras de otimização necessárias para um compilador emitir código de qualidade comparável ao de compiladores mais sofisticados, relatou que a ocorrência de conflitos entre regras de otimização é capaz de diminuir a qualidade do código-objeto produzido (Davidson; Whalley, 1989). Sejam, por exemplo, as seguintes regras de otimização, para a arquitetura x86:

```
mov e%00,0
=
xor e%00,e%00
```

e

```
mov %01,%00
mov %02,%01
=
mov %02,%00
mov %01,%00
```

Se o compilador emite a seguinte seqüência de instruções:

```
mov eax,0
mov dword ptr(_fp),eax
```

O otimizador substitui a instrução de armazenamento de zero no operando de destino por uma instrução de reposicionamento do operando de destino, o que impossibilita a aplicação de uma regra de otimização que possa trazer mais benefícios no lugar da regra aplicada. O acréscimo de novas regras de otimização que manipulem cada caso

especial contorna o problema, conforme ilustra a seguinte regra de otimização que soluciona o impasse anterior:

```
xor e%00,e%00
mov %01,e%00
=
mov %01,0
```

Contudo, Davidson e Whalley (1989) descobriram uma solução mais elegante: atrasa-se a aplicação de algumas regras de otimização até que estejam disponíveis mais informações de contexto que permitam a aplicação de regras mais específicas. Por exemplo, reformula-se a regra de otimização que substitui uma instrução de armazenamento de zero no operando de destino por uma instrução de reposicionamento do operando de destino para:

```
mov e%00,0
%01
=
xor e%00,e%00
%01
```

e altera-se a ordem de análise das regras, de modo que a regra reformulada seja analisada após aplicação da regra que otimiza instruções de armazenamento de valores dos operandos de origem nos operandos de destino. A regra reformulada não pode ser aplicada até que a segunda instrução seja examinada, o que permite a aplicação da regra que otimiza instruções de movimentação de valores dos operandos de origem para os operandos de destino. Desta forma, quando ocorre um conflito entre regras de otimização igualmente aplicáveis, este algoritmo dá preferência à substituição das seqüências mais longas de instruções.

Um outro estudo, tratando especificamente esse problema (Gill, 1991) sugeriu o acréscimo de extensões ao gerador de código do compilador: uma chamada dominó simples[53] e a outra, construção dinâmica de código[54] que, juntas, eliminariam totalmente a necessidade de um estágio de otimização *peephole*. A técnica do dominó simples visa aprimorar a justaposição de instruções e blocos de instruções traduzidos do código intermediário para o código-objeto e a técnica de construção dinâmica de código visa resolver o problema da superposição de uma ou mais regras de

---

[53] *Simple domino*

[54] *Dynamic code building*

otimização. Como esse trabalho enfoca especificamente o problema da superposição das regras de otimização, vamos analisar com mais detalhes a técnica de construção dinâmica de código e remeter o leitor interessado na técnica do dominó simples à publicação que relata o estudo realizado (Gill, 1991).

Na técnica de construção dinâmica de código considera-se que cada instrução é uma regra de otimização formada por uma única instrução de busca e uma única instrução de substituição, e atribui-se a cada instrução um peso, que representa o número de ciclos de máquina necessários para a sua execução. As regras de otimização com duas ou mais instruções recebem a soma dos pesos das instruções que compõem a seqüência de substituição. Considerem-se, por exemplo, três instruções A, B e C e as regras de otimizações para as seqüências A;B e B;C:

$A \rightarrow (A,5)$
$B \rightarrow (B,10)$
$C \rightarrow (C,15)$
$A;B \rightarrow (X,10)$
$B;C \rightarrow (Y,10)$

às quais se associam pares ordenados cujos elementos representam respectivamente a seqüência de substituição associada à seqüência de busca e o número de ciclos necessários para a sua execução. Dadas as definições acima, na ausência de opções de otimização, o custo de execução da seqüência de instruções A;B;C é:

$A;B;C \rightarrow (A,5);(B,10);(C,15) \rightarrow (A;B;C,30)$

Quando um otimizador *peephole* convencional encontra a mesma seqüência de instruções, o custo de execução da seqüência de instruções A;B;C diminui com a substituição da seqüência A;B por X e se obtém:

$A;B;C \rightarrow (X,10);(C,15) \rightarrow (X;C,25)$

Disto resulta em uma redução sub-ótima de 5 ciclos, de acordo com as definições dadas. O custo mínimo de execução e, por conseqüência, a redução ótima, são atingidos ao se substituir B;C na seqüência de instruções A;B;C:

$A;B;C \rightarrow (A,5);(Y,10) \rightarrow (A;Y,15)$

A simples substituição das seqüências de instruções mais longas não é suficiente para garantir a geração da seqüência de instruções ótima. A fim de atingir esse objetivo, emprega-se um algoritmo de programação dinâmica que minimize o custo de execução. Para uma introdução à programação dinâmica pode-se consultar,

por exemplo, Cormen et al. (2001) ou Skiena, S. S. (1998).

O algoritmo usa uma pilha para armazenar a tripla: seqüência de instruções, custo total e lista de instruções. A pilha se encontra vazia no início da execução do algoritmo:

Seq. de Instruções	Custo	Lista de Instruções	Comentários
-	0	-	Pilha vazia

Tabela 3.3 - Situação inicial da pilha de controle

O algoritmo busca a próxima instrução, verifica as substituições possíveis em função dos valores armazenados na pilha e empilha o resultado para uso na próxima iteração do algoritmo. Assim, ao buscar A, a única possibilidade de substituição é A $\rightarrow$ (A,5) e a pilha fica:

Seq. de Instruções	Custo	Lista de Instruções	Comentários
A	5	A	Uma única possibilidade
-	0	-	Pilha vazia

Tabela 3.4 - Situação da pilha de controle após a primeira iteração

Ao buscar B, existem duas possibilidades de substituição: B $\rightarrow$ (B,10) e A;B $\rightarrow$ (X,10). Como o custo de A;B é 5+10=15 e o custo de X é 10, utiliza-se a seqüência com o menor custo total, obtendo-se a nova configuração da pilha:

Seq. de Instruções	Custo	Lista de Instruções	Comentários
B	10	X	Duas possibilidades consideradas
A	5	A	Uma única possibilidade
-	0	-	Pilha vazia

Tabela 3.5 - Situação da pilha de controle após a segunda iteração

Ao buscar C, existem duas possibilidades de substituição: C $\rightarrow$ (C,15) e B;C $\rightarrow$ (Y,10). Como o custo de X;C é 10+15=25 e o custo de A;Y é 5+10=15, empilha-se a seqüência com o menor custo total, obtendo-se a seguinte configuração:

Seq. de Instruções	Custo	Lista de Instruções	Comentários
C	15	A;Y	Seqüência Ótima
B	10	X	Duas possibilidades consideradas
A	5	A	Uma única possibilidade
-	0	-	Pilha vazia

Tabela 3.6 - Situação da pilha de controle após a última iteração

Esta é a forma com que o algoritmo obtém dinamicamente a seqüência de instruções ótima[55], uma vez que a seqüência ótima se encontra no topo da pilha. Para seqüências mais longas de instruções, o algoritmo repetiria os passos de iteração anteriormente descritos e pesquisaria mais profundamente a pilha em busca de seqüências mais longas de instruções.

Este é um dos poucos algoritmos clássicos a tentar resolver o problema do conflito entre regras de otimização igualmente aplicáveis. Embora o algoritmo procure minimizar o tempo de execução do código-objeto, é possível adaptá-lo facilmente para minimizar o espaço ocupado pelo código.

Como o interesse do trabalho está voltado para a otimização *peephole*, o escalonamento das instruções e a alocação de registradores serão tratados em uma próxima oportunidade, o leitor interessado pode consultar um dos textos avançados de projeto e construção e compiladores (Muchnick, 1997), (Morgan, 1998) e (Cooper; Torczon, 2004).

Neste capítulo se descreveu o processo de compilação para uma arquitetura monoprocessada hipotética até a fase de otimização do código-objeto. Em seguida, conceituou-se a otimização *peephole* e se apresentou o perfil de alguns de seus algoritmos, descreveu-se como se pode gerar automaticamente as regras de otimização através do emprego de técnicas de análise de fluxo e se apresentou o problema do conflito entre regras de otimização igualmente aplicáveis e como alguns algoritmos de otimização *peephole* tentam resolvê-lo. No capítulo seguinte vemos como um otimizador *peephole* adaptativo resolve este mesmo problema por meio da busca em profundidade de seqüências de regras de otimização.

[55] A solução encontrada é uma entre várias que possivelmente compõem o espaço de soluções do problema. Portanto, deve-se encará-la como uma espécie de ótimo local, ou seja, uma solução ótima para as regras de otimização e heurística adotadas.

4. ALGORITMO DE OTIMIZAÇÃO *PEEPHOLE* ADAPTATIVO

No capítulo anterior, se mostrou como o conflito entre regras de otimização igualmente aplicáveis pode prejudicar a geração de código-objeto e algumas medidas podem ser tomadas para minimizar o problema. Neste capítulo, vamos desenvolver um otimizador *peephole* que trata automaticamente o conflito entre regras de otimização. Este algoritmo será desenvolvido a partir de um algoritmo de otimização *peephole* já conhecido. Para obtê-lo, vamos aplicar uma técnica de projetos de algoritmos descrita no capítulo 2.

Nesta técnica, após a escolha de um algoritmo já conhecido, é preciso identificar uma condição especial que estenda a solução obtida pelo algoritmo original; em seguida, deve-se selecionar um local apropriado para acoplar uma ação adaptativa que seja executada quando a condição especial for satisfeita e elaborar uma ação adaptativa que estenda a solução de um caso particular para o caso geral do problema.

Após o projeto deste algoritmo, mostra-se um exemplo simples de operação e se fazem algumas considerações a respeito das regras de otimização que afetam o funcionamento de tais algoritmos de otimização *peephole*. Finalmente, se apresenta uma análise de complexidade pessimista do algoritmo resultante.

No próximo capítulo, veremos como este algoritmo é utilizado em um projeto completo de um software de otimização *peephole*.

4.1. Projeto do algoritmo de otimização *peephole* adaptativo

Vamos aplicar o paradigma de projeto de algoritmos adaptativos ao algoritmo de otimização *peephole* desenvolvido por Lamb (1981). A condição especial de entrada que condiciona a execução da ação adaptativa é a detecção da aplicação simultânea de duas ou mais regras de otimização. A ação adaptativa associada à condição especial de entrada é a aplicação concorrente de todas as possíveis regras de otimização.

Para simular a concorrência, o algoritmo utilizará uma pilha para armazenar o contexto de cada execução em andamento, ou *thread*. A organização e a subseqüente execução dos contextos armazenados na pilha será análoga ao que ocorre em uma busca em profundidade. Para uma introdução à busca em profundidade, pode-se

consultar, por exemplo, Russell e Norvig (1995). A criação de tantos *threads* quantas forem as regras de otimização aplicáveis em um dado instante propicia a aplicação exaustiva de todas as possíveis regras de otimização sobre a seqüência de instruções do código-objeto, começando a partir da instrução corrente, até o final da lista. A figura 4.1 mostra o pseudocódigo do algoritmo de otimização *peephole* adaptativo.

Para aplicar exaustivamente todas as regras possíveis, o algoritmo deve processar todos os *threads* memorizados na pilha. Para cada *thread*, o algoritmo deve processar todas as instruções a partir do apontador de instruções corrente até o final da lista de instruções. No início, existe um único *thread* presente na pilha, sendo que o seu apontador de instruções indica a primeira instrução do programa (José Neto, 2001).

À medida que novas instruções são incluídas na lista de instruções do *thread* atual, determina-se o novo conjunto de regras de otimização que a ela se aplicam. Caso esse conjunto seja unitário, o algoritmo efetua a otimização correspondente e determina o surgimento de outras eventuais possibilidades de otimização. Caso novas possibilidades de otimização não surjam, o algoritmo avança o apontador de instruções e efetua a leitura da próxima instrução. Caso ocorram, o algoritmo determina o conjunto de regras que a ele se aplicam. Caso se aplique uma única regra, o algoritmo repete o procedimento descrito anteriormente. Caso o algoritmo detecte a possibilidade de aplicação simultânea de duas ou mais regras de otimização, o algoritmo cria tantos *threads* quantas forem as regras de otimização aplicáveis, insere os novos *threads* no topo da pilha e retoma o processamento do *thread* corrente com a realização da otimização associada ao mesmo, a determinação de outras possibilidades de otimização, e assim por diante, até que toda a lista de instruções do programa se esgote. Quando isso ocorre, o algoritmo memoriza a lista de instruções armazenada pelo *thread* corrente, desempilha o próximo *thread* e repete o procedimento descrito acima para todas as instruções do código-objeto, iniciando-se pela instrução que estava sendo apontada na ocasião em que o *thread* foi criado.

Algoritmo de otimização *peephole* adaptativo

Entradas: (1) uma lista ligada c/as regras de otimização e uma lista duplamente ligada c/as instruções do código objeto; (2) pilha de controle auxiliar capaz de armazenar o contexto de cada *thread* em execução. No início, existe um único *thread* no topo da pilha de controle e o seu apontador de instruções aponta para a primeira instrução do programa; (3) variável local 'fim'.

Saídas: pelo menos uma seqüência de instruções otimizadas.

```
início
  Prepara a pilha de controle
  enquanto existe thread a processar no topo da pilha de controle faça
    Aponta a instrução corrente do código objeto
    enquanto existem instruções do código objeto a serem processadas faça
      se há regra de otimização ainda não aplicada então
        Executa a regra de otimização
      senão
        Empilha instrução na pilha de instruções do thread
      fim se
      Determina o número de regras de otimização aplicáveis
      se no. de regras de otimização aplicáveis > 0 então
        se no. de regras de otimização aplicáveis > 1 então
          -- Procedimento de clonagem
          Atribui ao thread corrente a 1a. regra de otimização aplicável
          para no. de regras de otimização aplicáveis - 1 faça
            Copia o thread atual
            Atribui ao novo thread a próxima regra de otimiz. aplicável
            Insere o novo thread no topo da pilha de controle
            Aponta a próxima regra de otimização aplicável
          fim para
        senão
          Atribui ao thread corrente a regra de otimização aplicável
        fim se
      senão
        Não existe regra de otimização aplicável
      fim se
      repita
         posiciona fim := VERDADEIRO
         se no. de regras de otimização aplicáveis > 0 então
           Executa a regra de otimização
           Determina o número de regras de otimização aplicáveis
           se no. de regras de otimização aplicáveis > 0 então
             set fim := FALSO
             se no. de regras de otimização aplicáveis > 1 então
               -- Repete o procedimento de clonagem
             senão
               Atribui ao thread corrente a regra de otimização aplicável
             fim se
           fim se
         fim se
       até (fim ? VERDADEIRO)
      Aponta a próxima instrução do código objeto
    fim enquanto
    Imprime a lista de instruções armazenadas e descarta thread atual
  fim enquanto
fim.
```

Figura 4.1 - Algoritmo de otimização *peephole* adaptativo

O algoritmo de otimização *peephole* adaptativo é capaz de resolver os conflitos resultantes entre regras de otimização igualmente aplicáveis, através de uma busca em profundidade de seqüências de regras de otimização. Se, por exemplo, uma função objetivo externa que seleciona o código-objeto mais reduzido for acoplada ao algoritmo adaptativo, o algoritmo resultante é capaz de encontrar a melhor taxa de redução do código. Caso a função objetivo externa selecione o código-objeto com o

menor tempo de execução, o algoritmo resultante é capaz de encontrar o código-objeto com tempo de execução mais rápido e assim por diante.

4.2. Exemplo de operação

A figura 4.2 mostra algumas regras de otimização para a arquitetura x86 empregadas pelo algoritmo de otimização *peephole* adaptativo. A regra da figura 4.2a substitui uma instrução de armazenamento de zero no operando de destino por uma instrução de reposicionamento do operando de destino; a regra da figura 4.2b elimina uma instrução que soma zero ao operando de destino; a regra da figura 4.2c substitui uma seqüência de salto em igualdade e salto incondicional por uma seqüência de salto em desigualdade; a regra da figura 4.2d substitui duas instruções que efetuam armazenamento indireto por uma única instrução de armazenamento indireto, sendo a instrução de soma de constante a registrador preservada em virtude do valor do registrador poder ser necessário no processamento subseqüente; a regra da figura 4.2e elimina o processamento desnecessário de instruções antes da realização de um salto e a regra da figura 4.2f substitui uma seqüência de instruções de movimentação de valores dos operandos de origem para os operandos de destino por uma única instrução de movimentação direta de valores, sendo a instrução que realiza a primeira movimentação preservada pela mesma razão que a regra da figura 4.2d.

mov e%00,0
%01
=
xor e%00,e%00
%01
	add %00,0
	=
		jeq %00
		jmp %01
		%00:
		=
		jne %01
		%00:
			add %00,%01
			mov %02,%03[%00]
			=
			mov %02,%03(%01)[%00]
			add %00,%01
				%00 e%01,%02
				j%03
				=
				j%03
					mov %01,%00
					mov %02,%01
					=
					mov %02,%00
					mov %01,%00

(a)	(b)	(c)	(d)	(e)	(f)

Figura 4.2 - Exemplo de algumas regras empregadas pelo otimizador

Antes do início do processamento do código não-otimizado, o otimizador efetua a leitura de um arquivo que armazena as regras de otimização mostradas na figura 4.2. A figura 4.3a mostra um fragmento de código não-otimizado e as figuras 4.3b, 4.3c e 4.3d mostram três fragmentos de código produzidos pelo algoritmo de otimização *peephole* adaptativo a partir do código não-otimizado da figura 4.3a.

1.mov eax,0 2.mov dword ptr(_fp),eax 3.add eax,0 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: (a)	xor eax,eax mov dword ptr(_fp),eax jne L2 L1: (b)	mov dword ptr(_fp),0 jne L2 L1: (c)	mov dword ptr(_fp),0 jne L2 L1: (d)

Figura 4.3 - Dados de entrada e dados de saída do otimizador: (a) fragmento de código não-otimizado; (b), (c) e (d) fragmentos produzidos a partir do código não-otimizado

A primeira versão de código otimizado da figura 4.3b foi obtida pela aplicação sucessiva das seguintes regras: (1) regra 4.2a às instruções das linhas 1 e 2; (2) regra 4.2b à instrução da linha 3; (3) regra 4.2d às instruções das linhas 4 e 5; (4) regra 4.2e ao resultado de (3) e à instrução da linha 6; (5) regra 4.2e ao resultado de (3) e à instrução da linha 6; (6) regra 4.2c ao resultado de (5) e às instruções das linhas 7 e 8. A figura 4.4 mostra as sucessivos passos de transformação do fragmento de código não-otimizado da figura 4.3a no código da figura 4.3b.

1.xor eax,eax 2.mov dword ptr(_fp),eax 3.add eax,0 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (1)	1.xor eax,eax 2.mov dword ptr(_fp),eax 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (2)	1.xor eax,eax 2.mov dword ptr(_fp),eax 5.mov eax,(4)[eax] 4.add eax,4 6.jeq L1 7.jmp L2 8.L1: Código após o passo (3)	1.xor eax,eax 2.mov dword ptr(_fp),eax 5.mov eax,(4)[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (4)
1.xor eax,eax 2.mov dword ptr(_fp),eax 6.jeq L1 7.jmp L2 8.L1: Código após o passo (5)	1.xor eax,eax 2.mov dword ptr(_fp),eax 7.jne L2 8.L1: Código após o passo (6)		

Figura 4.4 - Passos de transformação do fragmento não-otimizado (1)

A segunda versão de código otimizado da figura 4.3c foi obtida pela aplicação sucessiva das seguintes regras: (1) regra 4.2f às instruções das linhas 1 e 2; (2) regra 4.2a ao resultado de (1) e à instrução da linha 3; (3) regra 4.2b à instrução da linha 3; (4) regra 4.2d às instruções das linhas 4 e 5; (5) regra 4.2e ao resultado de (4) e à instrução da linha 6; (6) regra 4.2e ao resultado de (4) e de (5); (7) regra 4.2e ao resultado de (2) e (6) e, finalmente, (8) regra 4.2c ao resultado de (7) e às instruções das linhas 7 e 8. A figura 4.5 mostra as sucessivos passos de transformação do fragmento de código não-otimizado da figura 4.3a no código da figura 4.3c.

2.mov dword ptr(_fp),0 1.mov eax,0 3.add eax,0 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (1)	2.mov dword ptr(_fp),0 1.xor eax,eax 3.add eax,0 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (2)	2.mov dword ptr(_fp),0 1.xor eax,eax 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (3)	2.mov dword ptr(_fp),0 1.xor eax,eax 5. mov eax,(4)[eax] 4. add eax,4 6.jeq L1 7.jmp L2 8.L1: Código após o passo (4)
2.mov dword ptr(_fp),0 1.xor eax,eax 5. mov eax,(4)[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (5)	2.mov dword ptr(_fp),0 1.xor eax,eax 6.jeq L1 7.jmp L2 8.L1: Código após o passo (6)	2.mov dword ptr(_fp),0 6.jeq L1 7.jmp L2 8.L1: Código após o passo (7)	2.mov dword ptr(_fp),0 7.jne L2 8.L1: Código após o passo (8)

Figura 4.5 - Passos de transformação do fragmento não-otimizado (2)

A terceira versão de código otimizado da figura 4.3d foi obtida pela aplicação sucessiva das seguintes regras: (1) regra 4.2f às instruções das linhas 1 e 2; (2) regra 4.2b à instrução da linha 3; (3) regra 4.2a ao resultado de (1) e à instrução da linha 4; (4) regra 4.2d às instruções das linhas 4 e 5; (5) regra 4.2e ao resultado de (4) e à instrução da linha 6; (6) regra 4.2e ao resultado de (4) e de (5); (7) regra 4.2c ao resultado de (2) e de (6) e, finalmente, (8) regra 4.2c ao resultado de (7) e às instruções das linhas 7 e 8. A figura 4.6 mostra as sucessivos passos de transformação do fragmento de código não-otimizado da figura 4.3d.

2.mov dword ptr(_fp),0 1.mov eax,0 3.add eax,0 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (1)	2.mov dword ptr(_fp),0 1.mov eax,0 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (2)	2.mov dword ptr(_fp),0 1.xor eax,eax 4.add eax,4 5.mov eax,[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (3)	2.mov dword ptr(_fp),0 1.xor eax,eax 5. mov eax,(4)[eax] 4. add eax,4 6.jeq L1 7.jmp L2 8.L1: Código após o passo (4)
2.mov dword ptr(_fp),0 1.xor eax,eax 5. mov eax,(4)[eax] 6.jeq L1 7.jmp L2 8.L1: Código após o passo (5)	2.mov dword ptr(_fp),0 1.xor eax,eax 6.jeq L1 7.jmp L2 8.L1: Código após o passo (6)	2.mov dword ptr(_fp),0 6.jeq L1 7.jmp L2 8.L1: Código após o passo (7)	2.mov dword ptr(_fp),0 7.jne L2 8.L1: Código após o passo (8)

Figura 4.6 - Passos de transformação do fragmento não-otimizado (3)

O conflito entre as regras de otimização igualmente aplicáveis 4.2a e 4.2f às linhas 1 e 2 do código não-otimizado proporcionou a derivação de uma primeira e uma segunda versão de código otimizado e um conflito entre as regras 4.2a e 4.2b proporcionou a derivação de uma terceira versão de código otimizado a partir da segunda versão. Caso as regras da figura 4.2 fossem empregadas por um otimizador *peephole* convencional, não se poderia garantir a obtenção de uma versão ótima de código otimizado, uma vez que a ordem de análise das regras de otimização

influencia decisivamente o resultado final, conforme mencionado anteriormente.

4.3. Considerações sobre o projeto das regras de otimização

Consideremos as seguintes regras de otimização hipotéticas mostradas na figura 4.7. A regra da figura 4.7a substitui a instrução A por B, ou, abreviadamente, A→B; a regra da figura 4.7b substitui a instrução B por C, ou B→C, e a regra da figura 4.7c substitui a instrução C por A, ou C→A.

Em função de o algoritmo de otimização *peephole* (seja o algoritmo de Lamb ou o algoritmo adaptativo) realizar vários passos de otimização sobre um trecho localizado do código, o algoritmo de otimização *peephole* entra em um enlace infinito ao encontrar a instrução A, em virtude de aplicar de forma recorrente as regras A→B→C→A e assim por diante. Portanto, ao projetar as regras de otimização, o usuário estar atento para a eventual ocorrência de casos de referências circulares.

(a)	(b)	(c)
A = B	B = C	C = A

Figura 4.7 - Exemplo de referência circular entre regras

Existe a possibilidade de se criticar automaticamente o conjunto de regras em busca de referências circulares. Contudo, isto é dificultado pela presença de variáveis nas regras de otimização. Uma variável armazena uma cadeia de caracteres até encontrar um novo átomo da regra de otimização (veja o apêndice C). Isto permite a omissão de informações que se tornam disponíveis somente em tempo de execução, o que dificulta a detecção estática de referências circulares. Por outro lado, a utilização das variáveis permite que o número de regras de otimização seja menor, já que uma variável evita a enumeração de todas as condições necessárias para se aplicar uma regra.

Consideremos em seguida as regras de otimização hipotéticas, mostradas na figura 4.8. A regra da figura 4.8a substitui a seqüência de instruções ABC por X, ou ABC→X; a regra da figura 4.8b substitui a seqüência de instruções BCD por Y, ou BCD→Y. Evidentemente ocorre uma sobreposição das regras da figuras 4.8a e 4.8b. Caso o algoritmo de otimização *peephole* adaptativo se depare com a seqüência de instruções ABCD e faça uso das regras das figuras 4.8a e 4.8b, o algoritmo efetuará a

substituição ABC→X, deixando de aplicar a regra da figura 4.8b. O algoritmo adaptativo agirá desta maneira porque o usuário determinou uma janela de otimização com apenas 3 instruções. Caso o usuário desejasse tirar proveito da sobreposição existente entre as regras, poderia estender a janela de otimização para 4 instruções conforme mostram as regras das figuras 4.8c e 4.8d. Da forma como foi empregada, a variável '%00' representa uma instrução qualquer que efetivamente estende a janela de otimização e torna possível a aplicação simultânea das regras das figuras 4.8c e 4.8d pelo algoritmo de otimização *peephole* adaptativo.

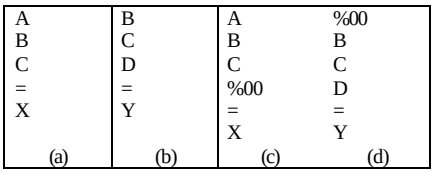

Figura 4.8 - Exemplo de indução de sobreposição de regras

A presença de variáveis nas regras de otimização também dificulta a detecção automática de conflitos entre regras de otimização sobrepostas, pois as variáveis omitem informações que só se tornam disponíveis em tempo de execução, o que dificulta a detecção estática de conflitos. Além desta dificuldade, a padronização do tamanho da janela de otimização também não é feita de maneira uniforme. Sejam os três exemplos mostrados abaixo.

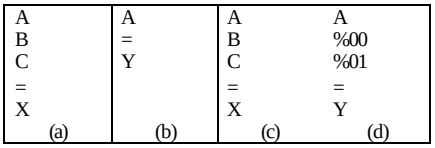

Figura 4.9 - Exemplo de sobreposição não uniforme de regras (1)

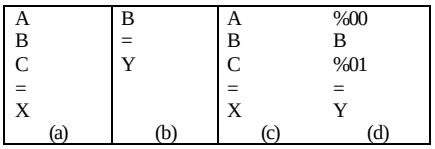

Figura 4.10 - Exemplo de sobreposição não uniforme de regras (2)

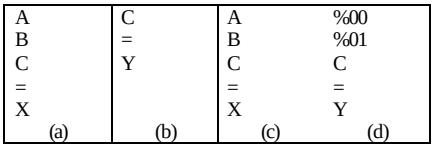

Figura 4.11 - Exemplo de sobreposição não uniforme de regras (3)

As figuras 4.9, 4.10 e 4.11 mostram que cada caso requer um tratamento específico visando a padronização do tamanho da janela de otimização.

Ainda que se tenha automatizado a geração de regras de otimização, conforme se mostrou no capítulo anterior, as considerações acima mostram que a elaboração de um conjunto de regras de otimização sem a ocorrência de referências circulares e com regras de otimização sobrepostas não parece ser trivial. O projeto das regras de otimização, especialmente com relação à automatização da análise e da padronização do tamanho da janela de otimização, talvez necessite de um estudo mais aprofundado.

4.4. Análise de complexidade pessimista

Como o algoritmo adaptativo realiza uma busca exaustiva, o algoritmo requer mais memória e tempo para execução em relação a um algoritmo de otimização *peephole* convencional. A complexidade de tempo pessimista de uma busca em profundidade clássica é $O(r^p)$, sendo r o fator de ramificação e p a profundidade da busca (Russell ; Norvig, 1995). Como o algoritmo adaptativo também efetua uma busca em profundidade, a sua complexidade de tempo pessimista também deverá ser função do fator de ramificação e da profundidade da busca. No caso pessimista, vamos considerar que o fator de ramificação seja igual ao maior número de regras de otimização igualmente aplicáveis e que aconteça uma nova ramificação a cada leitura de uma nova instrução de forma que a profundidade da busca seja igual ao número total de instruções do código objeto. Supondo-se, ainda, que as regras de otimização apenas permutem as instruções, sem efetuar eliminações nem substituições, tal que o número de instruções acumuladas a cada ramificação seja acrescido de um, então, nestas condições, a figura 4.12 representa a árvore de busca do algoritmo adaptativo.

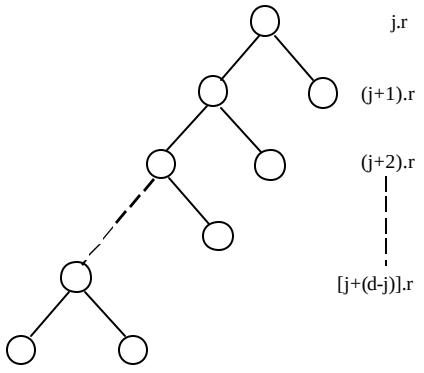

Figura 4.12 - Árvore de busca do algoritmo adaptativo

Nessa análise, j é o número de instruções necessárias para preencher uma janela de otimização. Admitindo-se que a profundidade da busca p, ou seja, o número total de instruções do código objeto, seja muito maior que o tamanho da janela de otimização j, pode-se calcular, a partir da figura 4.12, a complexidade de tempo pessimista, C_t, e a complexidade de espaço pessimista, C_s.

C_t deve ser proporcional à produtória do esforço necessário para expandir todos os nós da árvore de busca, ou seja para executar todos os procedimentos de clonagem. Como o tempo para executar este procedimento é proporcional ao número de instruções armazenadas em cada nó da árvore de busca, o esforço requerido para executar uma ramificação aumenta à medida que a profundidade da árvore aumenta. Em outras palavras, o custo de expansão dos nós é proporcional ao número de instruções armazenadas. Portanto, o valor de C_t é igual a:

$$C_t = jr \times (j+1)r \times \ldots \times [j+(p-j)]r = \{j \times (j+1) \times \ldots \times [j+(p-j)]\}r^{(p-j+1)} =$$
$$= [p!/(j-1)!]r^{(p-j+1)} = O(p!\, r^p)$$

Como o algoritmo não expande todos os nós da árvore de busca quando promove uma ramificação, C_s deve ser proporcional à somatória do número de instruções armazenadas em cada nó da árvore de busca. Assim, C_s é igual a:

$$C_s = jr + (j+1)r + \ldots + [j+(p-j)]r = \{j + (j+1) + \ldots + [j+(p-j)]\}r =$$
$$= [(j+p)(p-j+1)/2]r = O(rp^2)$$

É difícil de imaginar, nestas condições, que o algoritmo esteja aprimorando o código objeto. Na verdade, o algoritmo está efetuando todas as permutações possíveis resultantes do conflito de regras igualmente aplicáveis. Caso as regras não se limitassem apenas a permutar as instruções, mas, pelo contrário, promovessem o acréscimo de instruções, o resultado seria uma explosão combinatória de espaço e tempo. Contudo, dada a natureza da otimização *peephole*, a condição de que as regras de otimização efetuem, no máximo, uma permutação de instruções é suficiente para limitar C_t e C_s aos valores calculados. Mesmo assim, os valores teóricos calculados para C_t e C_s são preocupantes, principalmente C_t. Neste caso, a adoção de alguma forma de heurística pode contornar a situação. Uma possível heurística seria a adoção de um mecanismo de aprendizado para avaliar a efetividade das regras de otimização igualmente aplicáveis contra a função objetivo externa (Mitchell, 1997). As regras que contribuíssem para a satisfação da função objetivo externa seriam

recompensadas com a atribuição de créditos, sendo a atribuição de crédito negada para aquelas que não contribuíssem. Após a execução de um conjunto representativo de programas de treinamento, algumas regras de otimização eventualmente terão mais crédito que as suas contrapartes igualmente aplicáveis, sendo, então, as regras utilizadas na versão final de produção. Nesta situação, C_t e C_s se tornariam proporcionais ao número total de instruções do código objeto.

Nos testes de validação do software de otimização *peephole* apresentado no próximo capítulo, veremos que o desempenho do algoritmo de otimização *peephole* adaptativo na prática é melhor do que os cálculos de complexidade pessimista sugerem. Os testes mostrarão que quando não existem, ou quase não existem, conflitos entre as regras de otimização, o otimizador *peephole* adaptativo funciona da mesma forma que um otimizador *peephole* convencional e produz os mesmos resultados do que aquele.

5. ASPECTOS DE IMPLEMENTAÇÃO

O algoritmo de otimização *peephole* adaptativo irá otimizar código em linguagem de montagem da arquitetura x86 produzido pelo compilador LCC 4.x desenvolvido por Fraser e Hanson (1991; 1995 e 2001). Para uma definição do conjunto completo de regras empregadas, ver apêndice C. O compilador LCC é adequado como ferramenta de desenvolvimento por que: (a) embora implemente a compilação de uma linguagem completa, o ANSI C, e seja um compilador redirecionável, o acréscimo de passos adicionais de otimização é simples, sendo feito pelo encadeamento de novos programas de tratamento do código em linguagem de montagem produzido pelo compilador, conforme mostra a figura 5.1, neste sentido o processo de geração de código é similar àquele mostrado na figura 3.7; (b) mesmo sendo um compilador de produção, o código produzido pelo compilador não é muito otimizado (Fraser; Hanson, 1995), o que estimula o acréscimo de passos adicionais de otimização; (c) está disponível para uma série de plataformas de desenvolvimento de custo acessível (Fraser; Hanson, 2004) como, por exemplo, MS Visual Studio p/ MS-Windows 2000 (Microsoft, 1998), (Microsoft, 2000a) e (Microsoft, 2000b), ou Linux/GCC e (d) é distribuído gratuitamente pela Internet (Fraser; Hanson, 2004).

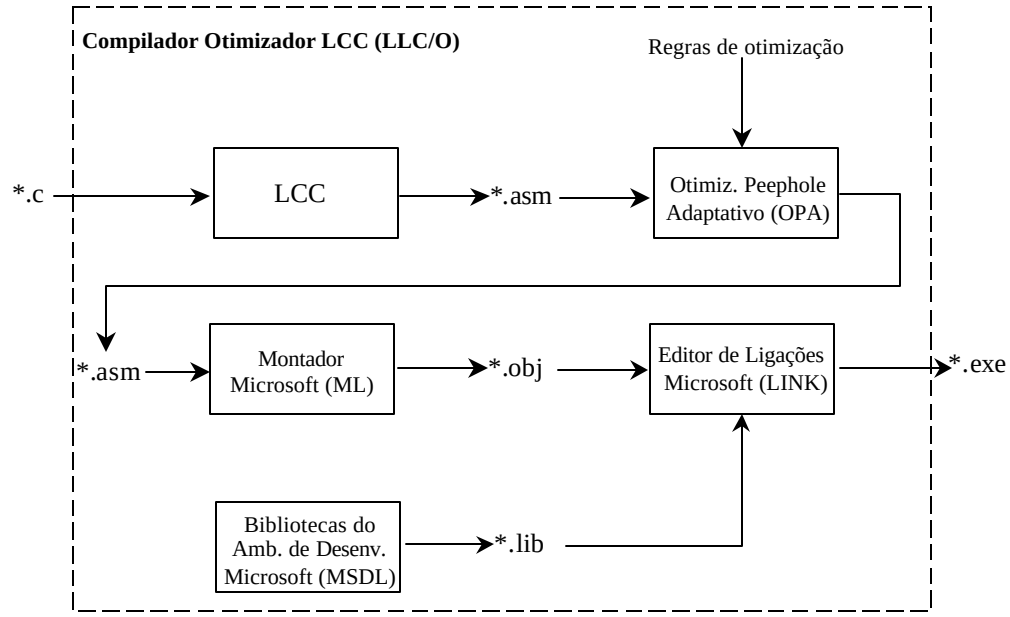

Figura 5.1 - Geração do código executável

5.1. Arquitetura do software

O algoritmo proposto no capítulo 4 foi implementado em uma versão do algoritmo de Lamb desenvolvida por Fraser em linguagem C (Fraser; Hanson, 1995). O local selecionado para acoplar uma ação adaptativa, que possibilite a execução de vários *threads* pelo otimizador resultante quando um conflito de regras for detectado, foi o enlace principal do otimizador *peephole* original. A figura 5.2 mostra a arquitetura de alto nível do otimizador *peephole* adaptativo e a figura 5.3 o pseudocódigo do otimizador *peephole* adaptativo.

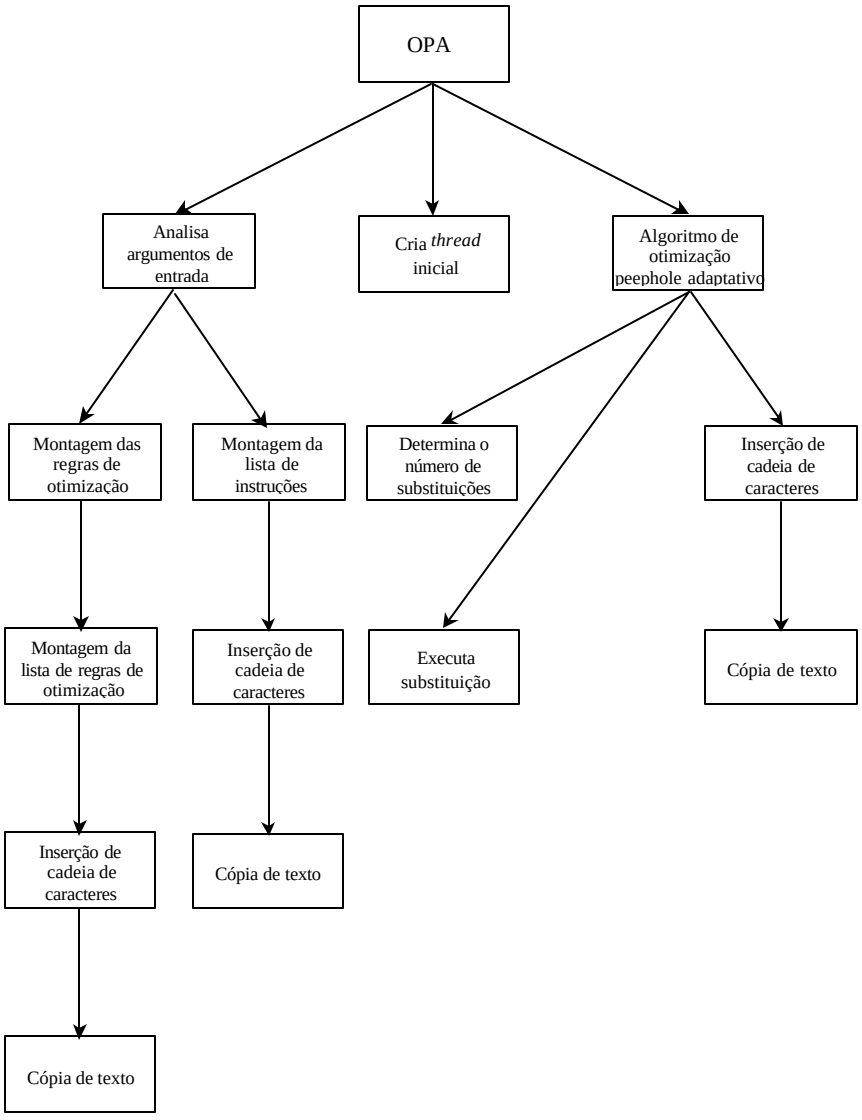

Figura 5.2 - Arquitetura de alto nível do otimizador *peephole* adaptativo

```
Programa otimizador peephole adaptativo

Entradas: argumentos de entrada contendo as opções de funcionamento e o nome
dos arquivos de regras de otimização e de instruções do código objeto.

Saídas: pelo menos uma seqüência de instruções otimizadas.

início
  chama Algoritmo de análise dos argumentos de entrada
  Cria thread inicial
  chama Algoritmo de otimização peephole adaptativo
fim.
```

Figura 5.3 - Otimizador *peephole* adaptativo

Como o algoritmo de otimização *peephole* adaptativo foi descrito no capítulo 4, vamos descrever o algoritmo de análise dos argumentos de entrada que cria os dados de entrada necessário para o correto funcionamento do algoritmo adaptativo. A figura 5.4 mostra o pseudocódigo deste algoritmo.

```
Algoritmo de análise dos argumentos de entrada

Entradas: argumentos de entrada contendo as opções de funcionamento e o nome
dos arquivos de regras de otimização e de instruções do código objeto.

Saídas: uma lista ligada contendo as regras de otimização e uma lista
duplamente ligada contendo as instruções do código objeto.

início
  Lê argumentos de entrada
  Posiciona indicadores
  Abre arquivo de regras de otimização
  chama Algoritmo de montagem das regras de otimização
  Fecha arquivo de regras de otimização
  Abre arquivo contendo as instruções do código objeto
  chama Algoritmo de montagem de lista
  Fecha arquivo contendo as instruções do código objeto
fim.
```

Figura 5.4 - Algoritmo de análise dos dados de entrada

O algoritmo de análise dos argumentos de entrada é auxiliado na tarefa de análise dos argumentos de entrada por dois algoritmos: (1) o algoritmo de montagem das regras de otimização e (2) o algoritmo de montagem de lista mostrados na figura 5.5.

```
Algoritmo de montagem das regras de otimização

Entradas: apontador do arquivo contendo as regras de otimização.

Saídas: lista ligada contendo as regras de otimização.

início
  Inicia lista ligada de regras de otimização
  enquanto não encontrou o fim do arquivo faça
    Cria um novo nó da lista ligada
    -- Monta uma pequena lista dupl. ligada com a seqüência de busca
    chama Algoritmo de montagem de lista
    se existe condição então
      Analisa condição
    fim se
    -- Monta uma pequena lista dupl. ligada com a seqüência de substituição
    chama Algoritmo de montagem de lista
    Analisa ganho
    Insere o novo nó na lista ligada
  fim enquanto
fim.

Algoritmo de montagem de lista

Entradas: apontador do arquivo, condição de parada, nó inicial e nó final da
lista duplamente ligada.

Saídas: lista duplamente ligada.

início
  Conecta o nó inicial e o nó final da lista dupl. ligada
  Lê uma linha do arquivo
  enquanto linha do arquivo ¹ condição de parada faça
    -- Insere cadeia na lista dupl. ligada
    chama Algoritmo de inserção de cadeia
    Lê uma linha do arquivo
  fim enquanto
fim.

Algoritmo de inserção de cadeia

Entradas: cadeia de texto e nó final da lista duplamente ligada.

Saídas: -

início
  Cria um novo nó da lista dupl. ligada
  chama Algoritmo de cópia de texto
  Insere o novo nó antes do último nó da lista dupl. ligada
fim.

Algoritmo de cópia de texto

Entradas: cadeia de texto.

Saídas: apontador da tabela de hash.

início
  Calcula função de hash
  Efetua uma busca da cadeia de texto na tabela de hash
  se não encontrou cadeia de texto na tabela de hash então
    Insere cadeia de texto na tabela de hash
  fim se
  Retorna apontador da tabela de hash p/a cadeia de texto
fim.
```

Figura 5.5 - Algoritmo de montagem das regras de otimização e de lista

A figura 5.6 mostra mais um passo de refinamento do algoritmo de otimização *peephole* adaptativo.

Algoritmo de otimização *peephole* adaptativo

Entradas: (1) uma lista ligada c/as regras de otimização e uma lista duplamente ligada c/as instruções do código objeto; (2) pilha de controle auxiliar capaz de armazenar o contexto de cada *thread* em execução. No início, existe um único *thread* no topo da pilha de controle e o seu apontador de instruções aponta para a primeira instrução do programa; (3) variáveis locais 'fim' e 'subs_poss'.

Saídas: pelo menos uma seqüência de instruções otimizadas.

```
início
  Prepara a pilha de controle
  enquanto existe thread a processar no topo da pilha de controle faça
    Aponta a instrução corrente do código objeto
    enquanto existem instruções do código objeto a serem processadas faça
      se há regra de otimização ainda não aplicada então
        chama Algoritmo de substituição
      senão
        chama Algoritmo de inserção de cadeia
      fim se
      chama Algoritmo de determinação do no. de substituições
      posiciona subs_poss := no. de substituições encontradas
      se subs_poss > 0 então
        se subs_poss > 1 então
          -- Procedimento de clonagem
          Atribui ao thread corrente a 1a. regra de otimização aplicável
          para subs_poss - 1 faça
            Copia o thread atual
            Atribui ao novo thread a próxima regra de otimiz. aplicável
            Insere o novo thread no topo da pilha de controle
            Aponta a próxima regra de otimização aplicável
          fim para
        senão
          Atribui ao thread corrente a regra de otimização aplicável
        fim se
      senão
        Não existe regra de otimização aplicável
      fim se
      repita
         posiciona fim := VERDADEIRO
         se subs_poss > 0 então
           chama Algoritmo de substituição
           chama Algoritmo de determinação do no. de substituições
           posiciona subs_poss := no. de substituições encontradas
           se subs_poss > 0 então
             posiciona fim := FALSO
             se subs_poss > 1 então
               -- Procedimento de clonagem
               Atribui ao thread corrente a 1a. regra de otimização aplicável
               para subs_poss - 1 faça
                 Copia o thread atual
                 Atribui ao novo thread a próxima regra de otimiz. aplicável
                  Insere o novo thread no topo da pilha de controle
                 Aponta a próxima regra de otimização aplicável
               fim para
             senão
               Atribui ao thread corrente a regra de otimização aplicável
             fim se
           fim se
         fim se
      até (fim ? VERDADEIRO)
      Aponta a próxima instrução do código objeto
    fim enquanto
    Imprime a lista de instruções armazenadas e descarta thread atual
  fim enquanto
fim.
```

Figura 5.6 - Algoritmo de otimização *peephole* adaptativo refinado

A figura 5.7 mostra o algoritmo de determinação do número de substituições e o algoritmo de substituição empregados pelo algoritmo de otimização *peephole* adaptativo e a listagem completa do otimizador *peephole* adaptativo se encontra no apêndice D.

```
Algoritmo de determinação do no. de substituições

Entradas: apontador da instrução corrente da lista dupl. ligada.

Saídas: número de substituições encontradas.

início
  Aponta 1a. regra de otimização
  Reposiciona contador de substituições
  enquanto não encontrou o fim da lista ligada de regras de otimização faça
    Compara seq. de busca da regra c/a seq. de instruções
    se encontrou seq. de busca na seq. de instruções então
      Incrementa contador de substituições
    fim se
    Aponta próxima regra de otimização
  fim enquanto
  Retorna o valor do contador de substituições
fim.

Algoritmo de substituição

Entradas: apontador da instrução corrente da lista dupl. ligada e o número da
substituição.

Saídas: ganho proporcionado.

início
  Aponta 1a. regra de otimização
  Reposiciona contador de substituições
  enquanto não encontrou o fim da lista ligada de regras de otimização faça
    Compara seq. de busca da regra c/a seq. de instruções
    se encontrou seq. de busca na seq. de instruções então
      Incrementa contador de substituições
    fim se
    se contador de substituições igual ao no. da substituição então
      Efetua troca da seq. de instruções pela seq. de substituição da regra
      Retorna o ganho proporcionado
    senão
      Aponta próxima regra de otimização
    fim se
  fim enquanto
  Não encontrou o número da substituição solicitada
fim.
```

Figura 5.7 - Algoritmo de determinação do número de substituições

5.2. Plano de validação do software

O acréscimo do algoritmo de otimização *peephole* adaptativo ao compilador original produz um compilador com otimização adaptativa. A fim de aferir os efeitos proporcionados ao sistema resultante, o desempenho do compilador com otimização adaptativa é agora comparado com o desempenho do compilador original e do compilador com otimização *peephole* convencional.

Além dessas avaliações de desempenho, o compilador com otimização adaptativa deve compilar o programa-fonte do compilador original e produzir uma nova versão executável do mesmo[56]. O novo programa executável do compilador deverá compilar os programas de *benchmark* agregados ao programa-fonte do compilador original e produzir os mesmos resultados que a versão original executável do compilador, conforme mostra figura 5.8.

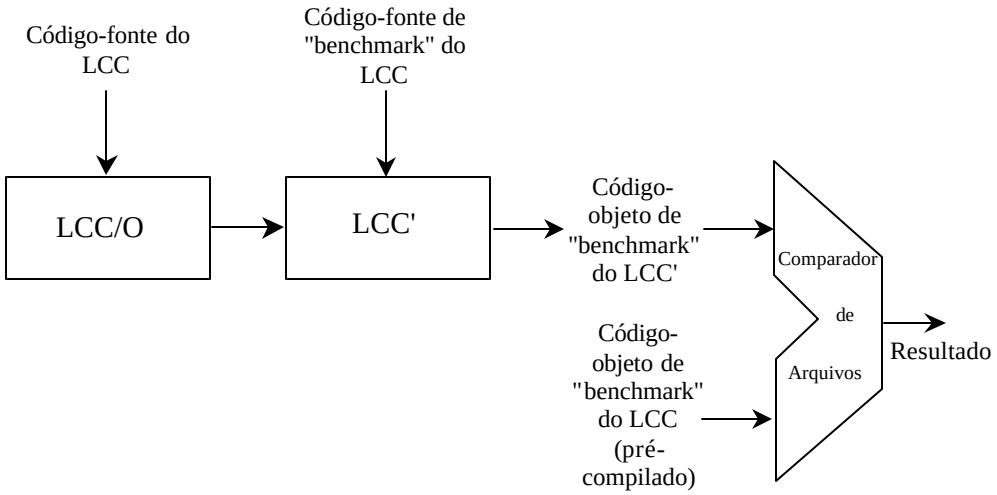

Figura 5.8 - *Bootstrapping* do LCC

[56] Esta técnica é denominada de *bootstrapping*

5.3. Resultados obtidos

O LCC trabalha com o apoio do ambiente de programação do compilador Visual C/C++ 6.0 da suíte MS Visual Studio. O LCC utiliza os arquivos de cabeçalho, as bibliotecas e as ferramentas de linha de comando para gerar os programas executáveis. A instalação do LCC no ambiente MS-Windows 2000 requer o compilador Visual C/C++, a ferramenta make da Microsoft, *nmake.exe*, e o interpretador de comandos do Windows, já que o arquivo MAKEFILE.NT incluído na distribuição do LCC foi projetado para usar somente o *nmake*. Assim, após a instalação do compilador Visual C/C++ 6.0 da suíte MS Visual Studio, copiar em um subdiretório qualquer o conteúdo do arquivo compactado que contém os arquivos de distribuição do LCC, LCC42.ZIP. Em seguida, executar o interpretador de comandos no subdiretório onde se copiaram os arquivos de distribuição do LCC e seguir as instruções para instalação do ambiente LCC apresentadas no apêndice E.

Após o teste e depuração do otimizador *peephole* adaptativo, o arquivo de *nmake*, MAKEFILE.NT, para geração e teste das ferramentas LCC, ou seja, do pré-processador, compilador, bibliotecas e outros utilitários, foi modificado para executar o processo representado na figura 5.8. Com isto, automatizou-se o procedimento de *bootstrapping* do LCC. O arquivo de *nmake* resultante, MAKELCCO.NT, está listado no apêndice G.

Com o procedimento de *bootstrapping* do LCC à disposição, foi possível validar o conjunto completo de regras de otimização descrito no apêndice C. Para tanto, criou-se um novo subdiretório para se armazenar uma versão do LCC gerada pelo LCC/O, ou LCC', e executaram-se os passos de geração e teste das ferramentas LCC com o arquivo MAKELCCO.NT.

O conjunto completo de regras de otimização originalmente submetido à validação[57] não passou no teste, pois se constatou que algumas regras alteraram a semântica do compilador original. Da inspeção das listagens de código em linguagem de montagem, descobriu-se que as regras de otimização seguintes eram as responsáveis pela modificação da semântica do compilador original:

---

[57] O conjunto apresentado no apêndice C, excetuando-se as regras 37, 38, 39 40, 41 e 42, juntamente com as regras mostradas nas tabelas 5.1 e 5.2.

mov %00,%01
push %00
%03 %00,%02
?
is_not_strstr(%02,%00);
=
push %01
%03 %00,%02
->
1
	mov %00,%01
	push %00
	call %02
	add esp,%03
	%04 %00,%05
	?
	is_not_strstr(%05,%00);
	=
	push %01
	call %02
	add esp,%03
	%04 %00,%05
	->
	1
		mov %00,%01
		push %00
		call %02
		add esp,%03
		%04:
		%05 %00,%06
		?
		is_not_strstr(%06,%00);
		=
		push %01
		call %02
		add esp,%03
		%04:
		%05 %00,%06
		->
		1

Tabela 5.1 - Regras que alteraram a semântica do compilador original (1)

O trecho '%03 %00,%02', da primeira regra, o trecho '%04 %00,%05', da segunda regra, e o trecho '%05 %00,%06', da terceira regra foram identificados como os responsáveis pela modificação da semântica. Por exemplo, consideremos a seguinte seqüência de instruções:

```
mov edi, dword ptr (-4)[ebp]
push edi
add edi,4
```

Caso se aplique a esta seqüência de instruções a primeira regra de otimização, o valor do registrador EDI se torna indeterminado, o mesmo acontecendo com o resultado da operação de adição. Neste caso, a exposição de mais contexto é a solução que torna segura a aplicação de qualquer uma das regras, mantendo-as, ao mesmo tempo, no conjunto de regras de otimização. As regras 37, 38, 39 40, 41 e 42 descritas no apêndice C são o resultado deste tratamento.

Contudo, a simples modificação das regras da figura 5.8 não foi suficiente para validar o conjunto de regras de otimização. Outras regras ainda estavam alterando a semântica do compilador original e com um agravante, dado o volume de listagens produzidas, a simples inspeção das listagens de código em linguagem de montagem foi insuficiente para se localizar as regras responsáveis pela modificação da semântica. Para tanto, desenvolveu-se uma nova forma de se isolar as regras defeituosas que lembra, em muito, uma pesquisa binária.

Dividiu-se inicialmente o conjunto de regras de otimização em duas partes. Em seguida, submeteu-se, em separado, as duas partes ao procedimento de validação.

Uma das partes submetida ao procedimento de validação passou, sendo que a outra parte não passou. O mesmo procedimento descrito acima foi repetido para a parte do conjunto de regras de otimização que não passou no teste e assim por diante até que as regras de otimização defeituosas eventualmente são localizadas. No nosso caso, constatou-se que as seguintes regras de otimização alteravam a semântica do compilador original:

movzx e%00,%01
cmp e%00,%02
j%03 %04
mov e%00,%05
?
is_const(%02);
=
cmp %01,%02
j%03 %04
mov e%00,%05
->
1
	movzx e%00,%01
	cmp e%00,%02
	j%03 %04
	lea e%00,%05
	?
	is_const(%02);
	=
	cmp %01,%02
	j%03 %04
	lea e%00,%05
	->
	1
		mov e%00,%01
		cmp e%00,%02
		j%03 %04
		mov e%00,%05
		?
		is_const(%02);
		=
		cmp %01,%02
		j%03 %04
		mov e%00,%05
		->
		1
			mov e%00,%01
			cmp e%00,%02
			j%03 %04
			lea e%00,%05
			?
			is_const(%02);
			=
			cmp %01,%02
			j%03 %04
			lea e%00,%05
			->
			1

Tabela 5.2 - Regras que alteraram a semântica do compilador original (2)

Da inspeção das listagens de código em linguagem de montagem, não se descobriu, até o momento, em qual caso as regras de otimização se mostram inadequadas. Neste caso, para tornar seguro o conjunto de regras de otimização, a melhor solução encontrada foi a eliminação destas quatro regras do conjunto descrito no apêndice C.

Após a consolidação do conjunto de regras de otimização, determinou-se o tempo de execução do procedimento de *bootstrapping* do LCC sem otimização *peephole*, com otimização *peephole* adaptativa e com otimização *peephole* convencional. Para a execução do procedimento de *bootstrapping* do LCC sem otimização *peephole*, o arquivo de *nmake* MAKELCCO.NT deu origem a um novo arquivo de *nmake*, o MAKELCC.NT, no qual se suprimiu o uso do otimizador *peephole* adaptativo. Para a execução do procedimento de *bootstrapping* do LCC com otimização *peephole* convencional, acrescentou-se a chave '-n' à constante OPTIM do arquivo de *nmake* MAKELCCO.NT para desativar a característica adaptativa do otimizador durante a sua execução. Exemplo:

```
OPTIM=opa -sx86 -n
```

sendo '-sx86' uma chave que habilita a contagem de instruções para a arquitetura x86. A tabela 5.3 mostra a média dos resultados de 10 execuções do procedimento de

bootstrapping[58] e as tabelas 5.4 e 5.5 mostram as taxas de reduções de código obtidas para cada módulo componente compilado.

Sem otimização *peephole*	**Com otimização *peephole* adaptativa**	**Com otimização *peephole* convencional**
54,7s	163,6s	165,6s

Tabela 5.3 - Tempos de execução do procedimento de *bootstrapping*

Módulo	**No. de instruções processadas**	**No. de instr. obtidas com otimização adaptativa**	**No. de instr. obtidas sem otimização adaptativa**	**Taxa de redução (%)**
alloc	166	164	164	1,2
bind	12	12	12	0
dag	4436	4296	4296	3,2
decl	5925	5675	5675	4,2
enode	4704	4499	4499	4,4
error	593	571	571	3,7
event	72	69	69	4,2
expr	4464	4273	4273	4,3
gen	4154	3992	3992	3,9
init	1516	1468	1468	3,2
inits	34	29	29	14,7
input	498	464	464	6,8
lex	2889	2522	2522	12,7
list	135	135	135	0
main	1178	1140	1140	3,2
null	302	302	302	0
output	816	767	767	6
prof	1163	1107	1107	4,8
profio	1052	1009	1009	4,1
simp	4957	4747	4747	4,2
stmt	3399	3261	3261	4,1
string	229	227	227	0,9
sym	1306	1275	1275	2,4
symbolic	2832	2731	2731	3,6
bytecode	1425	1371	1371	3,8

Tabela 5.4 - Taxas de redução de código obtidas (1)

[58] Utilizou-se um microcomputador equipado com dois processadores Intel Pentium Pro 200 MHz com 256 KB de cache cada, 128 MB de RAM e 10GB de HD interface IDE e placa gráfica ATI Graphics Pro Turbo PCI c/ 2MB de memória de vídeo. Neste microcomputador se instalou o sistema operacional Windows 2000 em Português e o pacote de serviços SP3, bem como todas as ferramentas necessárias para a geração e instalação do LCC descritas no início do capítulo. Além disso, os dados foram obtidos em condições idênticas de carga de processamento.

Módulo	No. de instruções processadas	No. de instr. obtidas com otimização adaptativa	No. de instr. obtidas sem otimização adaptativa	Taxa de redução
trace	925	887	887	4,1
tree	1135	1103	1103	2,8
types	4555	4341	4341	4,7
stab	1543	1457	1457	5,6
dagcheck	4621	4381	4381	5,2
alpha	13576	13005	13005	4,2
mips	11425	10982	10982	3,9
sparc	14548	14054	14054	3,4
x86	15734	15307	15307	2,7
bprint	2061	1963	1963	4,8
win32	152	150	150	1,3
assert	65	61	61	6,1
yynull	52	49	49	5,8
bbexit	457	426	426	6,8
cpp	906	865	865	4,5
lexer	920	890	890	3,3
nlist	207	200	200	3,4
tokens	1033	990	990	4,2
macro	1695	1605	1605	5,3
eval	1117	1077	1077	3,6
include	462	438	438	5,2
hideset	311	304	304	2,2
getopt	156	154	154	1,3
unix	324	315	315	2,8
Totais	**126237**	**121110**	**121110**	**4,1**

Tabela 5.5 - Taxas de redução de código obtidas (2)

5.4. Avaliação dos resultados e sugestões para trabalhos futuros

A partir da inspeção das tabelas 5.3 e 5.4 observa-se que o acréscimo do estágio de otimização provoca um aumento expressivo do tempo de execução do procedimento de *bootstrapping* do LCC em troca de uma redução média de aproximadamente 4% do número de instruções processadas e que a característica adaptativa não teve influência no resultado final do teste de validação. Isto se deveu à minimização dos conflitos entre regras. Até o momento, as regras 1 e 28 são as únicas que entram em conflito. A decisão de minimizar os conflitos entre regras amadureceu a partir da complexidade de tempo pessimista e dos resultados obtidos em testes realizados com o próprio código em linguagem de montagem do otimizador *peephole* adaptativo, pois se constataram, dadas as mesmas condições de carga de processamento e as mesmas regras de otimização descritas no apêndice C, tempos de execução da ordem de 450 segundos com o otimizador *peephole* operando com otimização adaptativa contra tempos da ordem de 2 segundos com o otimizador *peephole* operando com otimização convencional, sendo os resultados obtidos idênticos em ambos os casos[59].

Em função destes resultados, decidiu-se investigar mais atentamente as regras de otimização 1 e 28. Com o otimizador *peephole* operando com otimização convencional, omitiu-se primeiramente a regra 1 e, em seguida, a regra 28 e constatou-se que a redução de código proporcionada em ambos os casos é menor que aquela proporcionada quando ambas operam em conjunto. A seguir inverteu-se a ordem de análise das regras de forma que a regra 28 fosse executada antes da regra 1 e constatou-se que a redução de código proporcionada é igual à redução

[59] Durante a fase de elaboração das regras de otimização, o otimizador *peephole* operando com otimização adaptativa foi experimentado com um conjunto de regras de otimização similar ao conjunto de regras com que se iniciou o teste de validação, mas com um número de regras conflitantes maior em relação a este último. Nessas sessões preliminares, deixou-se o otimizador *peephole* operando com otimização adaptativa aprimorar o seu próprio código em linguagem de montagem durante 2 ou 3 horas ininterruptas. Como o otimizador não chegava a um resultado final, a sessão era interrompida manualmente. Num futuro próximo, essas sessões de teste devem ser reproduzidas com um computador muito mais rápido que o computador ora empregado. Contudo, para este trabalho, optou-se por reduzir os conflitos entre regras com a exposição de mais contexto.

proporcionada quando se analisa primeiramente a regra 1 e depois a regra 28. Portanto, pode-se concluir que as regras entram em conflito, mas uma não consegue ser superior à outra, e que o otimizador *peephole* pode operar com otimização convencional dadas as regras de otimização descritas no apêndice C.

Os resultados obtidos com a otimização do próprio código em linguagem de montagem do otimizador *peephole* adaptativo e a análise de complexidade de tempo pessimista encorajam a adoção de heurísticas ou a exploração de outras técnicas, como busca em largura ou busca por aprofundamento relativo, que produzam algoritmos com complexidade de tempo pessimista linear. Outra possibilidade seria a elaboração de um algoritmo de otimização *peephole* adaptativo paralelo que de fato executasse simultaneamente todas as regras de otimização aplicáveis.

Embora a redução média do número de instruções processadas seja de aproximadamente 4%, é possível aprimorar o conjunto de regras de otimização descrito no apêndice C a fim de obter melhores taxas de redução de código. Embora o acréscimo de novas regras possa melhorar a taxa de redução de código, a complexidade de tempo pessimista do otimizador operando com otimização convencional deve piorar já que o número de regras a ser analisada é maior. Para resolver este problema, a elaboração de uma nova gramática de regras de otimização que tornasse a representação das regras de otimização mais compactas, reduzisse o número de regras necessárias e proporcionasse a elaboração de expressões condicionais mais elaboradas seria parte importante de um esforço visando melhorar o desempenho do otimizador *peephole* e torná-lo mais flexível.

Espera-se, também, que a aplicação do paradigma de projetos de algoritmos adaptativos à otimização *peephole*, estimule a sua utilização na fase de otimização global e nas demais fases de otimização do código-objeto, especialmente, na super-otimização, no escalonamento das instruções e na alocação de registradores. Embora neste trabalho tenham elas sido apenas mencionadas, estas aplicações são sérias candidatas a uma investigação mais detalhada num futuro imediato, pois compartilham problemas de busca e substituição similares aos encontrados na otimização *peephole*. Por último, podem-se aplicar os algoritmos adaptativos na busca de seqüências de transformações otimizantes que minimizem uma determinada função-objetivo.

6. CONCLUSÕES E CONTRIBUIÇÕES

As contribuições apresentadas neste trabalho podem ser classificadas em três grandes áreas: a) otimização de código em compiladores; b) tecnologia adaptativa e c) engenharia de computação.

Para a área de otimização de código em compiladores pode-se mencionar:

- a apresentação de uma nova variante dos algoritmos clássicos de otimização *peephole*;
- a inclusão de novas funcionalidades ao algoritmo clássico permitindo: (1) a possibilidade de interação do usuário[60] para facilitar a escolha de opções, (2) a possibilidade de busca exaustiva, se necessário, no espaço de soluções resultante dos conflitos entre regras e (3) a possibilidade de obtenção de combinações ótimas e sub-ótimas, sob controle do usuário e
- o estudo de complexidade desenvolvido para o algoritmo implementado.

Para a área de tecnologia adaptativa pode-se mencionar:

- uma nova aplicação de tecnologia adaptativa à área de software básico;
- comprovação da aplicabilidade de formalismos adaptativos à otimização de código orientada por regras;
- a instanciação de algoritmos complexos, de base adaptativa, em um caso prático de grande porte;
- o exercício de programação de algoritmos adaptativos em linguagem de programação convencional;
- a demonstração, na prática, da viabilidade da programação adaptativa;
- a criação de um substrato para a implementação futura de um sistema de substituição mais complexo e
- a constatação de que o método utilizado é comparável em seus resultados a outros normalmente adotados, como a programação dinâmica e similares.

Para a área de engenharia de computação pode-se mencionar:

- a aproximação da área de software básico da área de tecnologia adaptativa;
- a inclusão de alternativa tecnológica genuinamente nacional no projeto de

[60] Neste contexto, o usuário deve ser entendido como o projetista do sistema de programação.

sistemas de programação;

- a disponibilização de um conjunto importante de ferramentas para o desenvolvimento de outros programas de sistemas baseados em sistemas de substituições condicionais, paramétricas e dependentes de contexto e
- a indicação de outras áreas da engenharia da computação nas quais a tecnologia desenvolvida se aplica.

LISTA DE REFERÊNCIAS

AHO, A.V.; ULLMAN, J. D. **Principles of compiler design**. Reading, MA: Addison-Wesley Publ. Co., 1979.

AHO, A.V.; ULLMAN, J. D. **The theory of parsing, translation and compiling**. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1986. v.1.

AHO, A.V.; SETHI, R.; ULLMAN, J. D. **Compilers: principles, techniques and tools**. Reading, MA: Addison-Wesley Publ. Co., 1986.

AHO, A.V.; GANAPATHI, M.; TJIANG, S. W. K. Code generation using tree matching and dynamic programming. **ACM Transactions on Programming Languages and Systems**, New York, v.11, n.4, p.491-516, Oct.1989.

ALLEN, F. E.; COCKE, J. A catalogue of optimizing transformations. In: RUSTIN, R. (Ed.) **Design and optimization of compilers**. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1972.

BACON, D. F.; GRAHAM, S. L.; SHARP, O. J. Compiler transformations for high-performance computing. **ACM Computing Surveys**, New York, v.26, n.4, p.345-420, Dec. 1994.

BACKUS, J. et al. The FORTRAN automatic coding system. In: ROSEN, S. (Ed.) **Programming systems and languages**. New York, NY: McGraw-Hill Book Co., 1967.

BACKUS, J. Programming in America in the 1950s - some personal recollections. In: METROPOLIS, N.; HOWLETT, J.; ROTA, G. (Eds.) **A history of computing in the twentieth century**. New York, NY: Academic Press, 1980.

BENTLEY, J. **Programming pearls** 2nd. ed. New York, NY: ACM Press Books, 2000.

BREY, B. B. **The INTEL microprocessors 8086/8088, 80186/80188, 80286, 80386, 80486, Pentium and Pentium Pro processor, Pentium II, Pentium III, Pentium 4: architecture, programming and interfacing**. 6th. ed. Upper Saddle River, NJ: Prentice-Hall, 2002.

CERUZZI, P. E. **A history of modern computing**. Cambridge, MA: MIT Press, 1998.

COOPER, K. D.; MCINTOSH, N. Enhanced code compression for embedded RISC processors. **ACM SIGPLAN Notices**, v.34, n.5, p.139-149, May 1999. / Apresentado a ACM SIGPLAN'99 Conference on Programming Language Design and Implementation, Atlanta, GA, USA, 1999 /

COOPER, K. D.; SCHIELKE, P. J.; SUBRAMANIAN D. Optimizing for reduced code space using genetic algorithms. **ACM SIGPLAN Notices**, New York, v.34,

n.7, p.1-9, July 1999.

COOPER, K. D.; SUBRAMANIAN D.; TORCZON, L. Adaptive optimizing compilers for the 21st. century. **The Journal of Supercomputing**, Hingham, v.23, n.1, p.7-22, Aug. 2002.

COOPER, K. D.; TORCZON, L. **Engineering a Compiler**. San Francisco, CA: Morgan Kaufmann Publishers, 2004.

CORMEN, T. H. et al. **Introduction to algorithms**. 2nd. ed. Cambridge, MA: MIT Press, 2001.

CYTRON, R. et al. Efficiently computing static single assignment form and the control dependence graph. **ACM Transactions on Programming Languages and Systems**, New York, v.13, n.4, p.451-490, Oct.1991.

DAVIDSON, J. W.; FRASER, C. W. The design and application of a retargetable peephole optimizer. **ACM Transactions on Programming Languages and Systems**, New York, v.2, n.2, p.191-202, Apr.1980.

DAVIDSON, J. W.; FRASER, C. W. Automatic generation of peephole optimizations. **ACM SIGPLAN Notices**, v.19, n.6, p.111-116, June 1984a. / Apresentado a ACM SIGPLAN'84 Symposium on Compiler Construction, Montreal, Canada, 1984 /

DAVIDSON, J. W.; FRASER, C. W. Register allocation and exhaustive peephole optimization. **Software -Practice and Experience**, New York, v.14, n.9, p.857-865, Sept.1984b.

DAVIDSON, J. W.; FRASER, C. W. Code selection through object code optimization. **ACM Transactions on Programming Languages and Systems**, New York, v.6, n.4, p.505-526, Oct.1984c.

DAVIDSON, J. W.; FRASER, C. W. Automatic inference and fast interpretation of peephole optimization rules. **Software -Practice and Experience**, New York, v.17, n.11, p.801-812, Nov.1987.

DAVIDSON, J. W.; WHALLEY, D. B. Quick compilers using peephole optimization. **Software -Practice and Experience**, New York, v.19, n.1, p.79-97, Jan.1989.

DE SUTTER, B.; DE BOSSCHERE, K. Software techniques for code compaction. **Communications of the ACM**, New York, v.46, n.8, p.33-34, Aug. 2003.

DEBRAY, S. K. et al. Compiler techniques for code compaction. **ACM Transactions on Programming Languages and Systems**, New York, v.22, n.2, p.378-415, Mar. 2000.

DRINIC, M.; KIROVSKI, D.; VO, H. Code optimization for code compression. In: International Symposium on Code Generation and Optimization, 1st, San Francisco, CA, USA, Mar. 2003. **Proceedings**. Los Alamitos, CA: IEEE Computer Society Press, 2003, p.315-324.

ERNST, J. et al. Code compression. **ACM SIGPLAN Notices**, v.32, n.5, p.358-365, May 1997. / Apresentado a ACM SIGPLAN'97 Conference on Programming Language Design and Implementation, Las Vegas, NV, USA, 1997 /

FISCHER, C. N.; LeBLANC, R. J. **Crafting a compiler with C**. Redwood City, CA: The Benjamin/Cummings Publ. Co., Inc., 1991.

FRANZ, M.; KISTLER, T. Slim Binaries. **Communications of the ACM**, New York, v.40, n.12, p.87-94, Dec. 1997.

FRASER, C. W.; MYERS, E. W.; WENDT, A. L. Analyzing and compressing assembly code. **ACM SIGPLAN Notices**, v.19, n.6, p.117-121, June 1984. / Apresentado a ACM SIGPLAN'84 Symposium on Compiler Construction, Montreal, Canada, 1984 /

FRASER, C. W.; HANSON, D. R. A retargetable compiler for ANSI C. **ACM SIGPLAN Notices**, New York, v.26, n.10, p.29-43, Oct. 1991.

FRASER, C. W.; HANSON, D. R. **A retargetable C compiler: design and implementation**. Menlo Park, CA: Addison-Wesley Publ. Co., 1995.

FRASER, C. W.; HANSON, D. R. **The LCC 4.x code-generation interface**. Redmond, WA: Microsoft Research, 2001 (Technical Report MSR-TR-2001-64).

FRASER, C. W.; HANSON, D. R. LCC: a retargetable C compiler. Princeton, NJ. Sept. 2004. Distribuição do código-fonte para geração do LCC e instruções para instalação e execução do LCC em uma série de plataformas. Disponível em <http://www.cs.princeton.edu/software/lcc> Acesso em 04 de set. 2004.

FRASER, C. W.; WENDT, A. L. Integrating code generation and optimization. **ACM SIGPLAN Notices**, v.21, n.7, p.242-248, July 1986. / Apresentado a ACM SIGPLAN'86 Symposium on Compiler Construction, Palo Alto, CA, USA, 1986 /

FRASER, C. W.; WENDT, A. L. Automatic generation of fast optimizing code generators. **ACM SIGPLAN Notices**, v.23, n.7, p.79-84, July 1988. / Apresentado a ACM SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, GA, USA, 1988 /

GIEGERICH, R. A formal framework for the derivation of machine-specific optimizers. **ACM Transactions on Programming Languages and Systems**, New York, v.5, n.3, p.478-498, July 1983.

GILL, A. A novel approach towards peephole optimisations. In: Annual Glasgow Workshop on Functional Programming, 4th., Portree, Isle of Skye, Scotland, Aug. 1991. **Functional Programming, Glasgow 1991**. Berlin: Springer-Verlag, 1992, p.100-111.

GOLDBERG, D. What every computer scientist should know about floating-point arithmetic. **ACM Computing Surveys**, New York, v.23, n.1, p.5-48, Mar. 1991.

GRAHAM, S. L. Code generation and optimization. In: LORHO, B. (Ed.) **Methods and tools for compiler construction**. Cambridge: Cambridge University Press, 1984.

GRANLUND, T.; KENNER, R. Eliminating branches using a superoptimizer and the GNU C compiler. **ACM SIGPLAN Notices**, v.27, n.2, p.341-352, July 1992. / Apresentado a ACM SIGPLAN'92 Conference on Programming Languages Design and Implementation, San Francisco, CA, USA, 1992 /

GRUNE, D.; BAL, H. E.; JACOBS, C. J. H.; LANGENDOEN, K. G. **Modern compiler design**. Chichester: John Wiley & Sons Ltd., 2000.

IEEE. **IEEE Standard for binary floating-point arithmetic ANSI/IEEE Std. 754-1985**. New York, NY: IEEE, 1985.

INTEL CORP. **ASM386 Assembly Language Reference**. Mount Prospect, IL: Intel Corp., 1995.

INTEL CORP. **Intel architecture software developer's manual volume 1: basic architecture**. Mount Prospect, IL: Intel Corp., 1997a.

INTEL CORP. **Intel architecture software developer's manual volume 2: instruction set reference**. Mount Prospect, IL: Intel Corp., 1997b.

INTEL CORP. **Intel architecture software developer's manual volume 3: system programming guide**. Mount Prospect, IL: Intel Corp., 1997c.

INTEL CORP. **Intel architecture optimization manual**. Mount Prospect, IL: Intel Corp., 1997d.

JOHNSON, S. C. **YACC: yet another compiler-compiler**. Murray Hill, NJ: Bell Labs., 1975 (Computing Science Technical Report no. 32).

JOSÉ NETO, J. **Introdução à Compilação**. São Paulo, SP: LTC, 1987.

JOSÉ NETO, J. **Contribuições à metodologia de construção de compiladores.** São Paulo, 1993, 272p. Tese (Livre-Docência) Escola Politécnica, Universidade de São Paulo.

JOSÉ NETO, J. Adaptive automata for context-dependent languages. **ACM SIGPLAN Notices**, New York, v.29, n.9, p.115-124, Sept. 1994.

JOSÉ NETO, J. Solving complex problems efficiently with adaptive automata. In: International Conference on Implementation and Application of Automata - CIAA 2000, 5th., London, Ontario, Canada, July 2000. **Lectures Notes on Computer Science 2088**. Berlin: Springer-Verlag, 2001, p.340-342.

JOSÉ NETO, J. Adaptive rule-driven devices - general formulation and case study. In: International Conference on Implementation and Application of Automata - CIAA 2001, 6th., Pretoria, South Africa, July 2001. **Lectures Notes on Computer Science 2494**. Berlin: Springer-Verlag, 2002, p.234-250.

JOSHI, R.; NELSON, G.; RANDALL, K. DENALI: a goal directed superoptimizer. **ACM SIGPLAN Notices**, v.37, n.5, p.304-314, May 2002. / Apresentado a ACM SIGPLAN'02 Conference on Programming Language Design and Implementation, Berlin, Germany, 2002 /

KENNEDY, K. A survey of data flow analysis techniques. In: MUCHNICK, S. S.; JONES, N.D. (Eds.) **Program flow analysis: theory and applications**. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1981.

KESSLER, P. B. Discovering machine-specific code improvements. **ACM SIGPLAN Notices**, v.21, n.7, p.249-254, July 1986. / Apresentado a ACM SIGPLAN'86 Symposium on Compiler Construction, Palo Alto, CA, USA, 1986 /

KESSLER, R. R. Peep-an architectural description driven peephole optimizer. **ACM SIGPLAN Notices**, v.19, n.6, p.106-110, June 1984. / Apresentado a ACM SIGPLAN'84 Symposium on Compiler Construction, Montreal, Canada, 1984 /

KIM, D. H.; LEE, H. J. Iterative procedural abstraction for code size reduction. In: International Conference on Compilers, Architecture, and Synthesis for Embedded Systems - CASES'02, 3rd., Grenoble, France, Oct. 2002. **Proceedings**. New York, NY: ACM Press, 2002, p.277-279.

KIM, J.; OH, S. EM-code optimization algorithm using tree pattern matching. In: 1997 International Conference on Information, Communications and Signal Processing - ICICS'97, Singapore, Sept. 1997. **Proceedings**. Singapore: IEEE, 1997, v.2, p.917-923.

KNUTH, D. The early development of programming languages. In: METROPOLIS, N.; HOWLETT, J.; ROTA, G. (Eds.) **A history of computing in the twentieth century**. New York, NY: Academic Press, 1980.

LAMB, D. A. Construction of a peephole optimizer. **Software-Practice and Experience** , New York, v.11, n.6, p.639-647, June 1981.

LESK, M. E.; SCHMIDT, E. **LEX - a lexical analyzer generator**. Murray Hill, NJ: Bell Labs., 1975 (Computing Science Technical Report no. 39).

LEVERETT, B. W. et al. An overview of the production-quality compiler-compiler project, **IEEE Computer**, v.13, n.8, p.38-49, Aug. 1980.

LEWIS, D.W. **Fundamentals of embedded software: where C and assembly meet**. Upper Saddle River, NJ: Prentice-Hall, 2002.

LEWIS, H. R.; PAPADIMITRIOU, C. H. **Elements of the Theory of Computation.** Englewood Cliffs, NJ, Prentice-Hall, 1981.

LORHO, B. (Ed.) **Methods and tools for compiler construction: an advanced course**. Cambridge: Cambridge University Press, 1984.

MASSALIN, H. Superoptimizer - a look at the smallest program. **ACM SIGPLAN Notices**, v.22, n.10, p.122-126, Oct. 1987. / Apresentado a 2nd. International Conference on Architectural Support for Programming Languages and Operating Systems, Palo Alto, CA, USA, 1987 /

McKEEMAN, W. M. Peephole optimization. **Communications of the ACM**, New York, v.8, n.7, p.443-444, July 1965.

McKENZIE, B. J. Fast peephole optimization techniques. **Software-Practice and Experience**, New York, v.19, n.12, p.1151-1162, Dec.1989.

MICROSOFT **Visual Studio, version 6.0.** [s.l.]: Microsoft Corp., 1998. Conjunto de programas. 1 CD-ROM.

MICROSOFT CORP. **Visual C++ 6.0 Processor Pack.** [s.l.]. Nov.2000a. Versão 6.15 do montador Microsoft MASM. Disponível em <http://msdn.microsoft.com/vstudio/downloads/tools/ppack/default.asp> Acesso em 04 de set. 2004.

MICROSOFT CORP. **Visual Studio 6.0 Service Pack 5.** [s.l.]. Nov.2000b. Últimas atualizações p/o sistema de desenvolvimento visual studio 6.0. Disponível em <http://msdn.microsoft.com/vstudio/downloads/updates/sp/vs6/sp5/default.asp> Acesso em 04 de set. 2004.

MITCHELL, T. M. **Machine learning**. Singapore: McGraw-Hill, 1997.

MOREL, E. Data flow analysis and global optimization. In: LORHO, B. (Ed.) **Methods and tools for compiler construction**. Cambridge: Cambridge University Press, 1984.

MORGAN, R. **Building an optimizing compiler**. Woburn, MA: Butterworth-Heinemann, 1998.

MUCHNICK, S. S. **Advanced compiler design and implementation**. San Francisco, CA: Morgan Kaufmann Publ. Co., 1997.

NAVIA, J. **Lcc-win32: a compiler system for windows**. Paris. Sept. 2002. Distribuição da documentação técnica do sistema de programação LCC para o ambiente windows. Disponível em <http://www.cs.princeton.edu/~lcc-win32/> Acesso em 11 de set. 2004.

NYSTRÖM, S. O.; RUNESON, J.; SJÖDIN, J. Optimizing code size through procedural abstraction. In: ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems - LCTES'2000, Vancouver, BC, Canada, June 2000. **Lectures Notes on Computer Science 1985**. Berlin: Springer-Verlag, 2001, p.204-205.

PELEGRÍ-LLOPART, E.; GRAHAM, S. L. Optimal code generation for expression trees: an application of BURS theory. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 15th., San Diego, CA, USA, Jan. 1988. **Proceedings**. New York, NY: ACM Press, 1988, p.294-308.

PISTORI, H. **Tecnologia adaptativa em engenharia de computação: estado da arte e aplicações.** São Paulo, 2003, 172p. Tese (Doutorado) Escola Politécnica, Universidade de São Paulo.

PROEBSTING, T. A. Optimizing an ANSI C interpreter with superoperators. In: ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, 22nd., San Francisco, CA, USA, Jan. 1995. **Proceedings**. New York, NY: ACM Press, 1995, p.322-332.

REIF, J. H.; LEWIS, H. R. Symbolic evaluation and the global value graph. In: ACM SIGACT-SIGPLAN Symposium of Programming Languages, 4th., Los Angeles, CA, USA, Jan. 1977. **Proceedings**. New York, NY: ACM Press, 1977, p.104-118.

RUNESON, J. **Code compression through procedural abstraction before register allocation.** Uppsala, 2000, 21p. Thesis (Master's) Computer Science Dept., Uppsala University.

RUSSELL, S.; NORVIG, P. **Artificial Intelligence: a modern approach**. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1995.

SAMMET, J. E. **Programming languages: history and fundamentals**. Englewood Cliffs, N. J.: Prentice-Hall, Inc., 1969.

SAMMET, J. E. Programming languages: history and future. **Communications of the ACM**, New York, v.15, n.7, p.601-610, Jul. 1972.

SKIENA, S. S. **The algorithm design manual**. New York, NY: Springer-Verlag New York, Inc., 1998.

SPINELLIS, D. Declarative peephole optimization using string pattern matching. **ACM SIGPLAN Notices**, New York, v.34, n.2, p.47-51, Feb.1999.

TANENBAUM, A. S.; VAN STAVEREN, H.; STEVENSON, J. W. Using peephole optimization on intermediate code. **ACM Transactions on Programming Languages and Systems**, New York, v.4, n.1, p.21-36, Jan.1982.

TREMBLAY, J. P.; SORENSON, P. G. **The theory and practice of compiler writing**. New York, NY: McGraw-Hill Book Co., 1985.

VAN DE WIEL, R. **The code compaction bibliography**, [s.l]. June 2004. Este sítio compila uma bibliografia de trabalhos sobre a redução do espaço ocupado por softwares de sistemas embutidos. Disponível em <http://www.extra.research.philips.com/ccb> Acesso em 04 de set. 2004.

WATSON, D. **High-level languages and their compilers**. Wokingham, England: Addison-Wesley Publ. Co., 1989.

WATT, D. A. Contextual constraints. In: LORHO, B. (Ed.) **Methods and tools for compiler construction**. Cambridge, England: Cambridge University Press, 1984.

WULF, W. et al. **The design of an optimizing compiler**. New York, NY: American Elsevier Publ. Co., Inc., 1975.

GLOSSÁRIO

Abstract syntax tree
ver 'árvore sintática abstrata'.

Annotated abstract syntax tree
ver 'árvore sintática abstrata decorada'.

Abstração de procedimento
Técnica de otimização do código-objeto, ou de formas intermediárias de representação do programa, que procura seqüências repetidas de instruções e as substitui por chamadas de um único procedimento.

Alfabeto
É um conjunto finito de símbolos, em geral letras, números e símbolos tipográficos, usados para compor as sentenças de uma linguagem de programação.

Alocação de registradores
Passo de otimização do código-objeto que efetua um mapeamento das variáveis temporárias utilizadas no código intermediário para os registradores do computador.

Ambiente de execução
Para que o código-objeto seja corretamente executado pelo processador é necessária a associação de ações que ocorrem apenas em tempo de execução. São exemplo de tais ações: inserção e retirada dos registros de ativação da pilha de controle, alocação e liberação de memória dinâmica, ativação de rotinas de biblioteca, comunicação com o sistema operacional, identificação do método correspondente durante a ativação de um método em uma linguagem orientada a objetos, comunicação entre módulos paralelos e concorrentes em linguagens paralelas e distribuídas, ativação de rotinas de unificação de parâmetros em linguagens lógicas, etc. O ambiente responsável pela execução do conjunto de todas estas ações é chamado ambiente de execução. As ações implementadas pelo ambiente de execução são carregadas com o código-objeto durante a síntese do código-objeto e/ou durante a edição do programa pelo editor de ligações.

Análise do fluxo de dados
Técnica de extração de informações sobre o fluxo de valores em tempo de execução a partir do grafo de fluxo de controle do código intermediário. Ver 'grafo de fluxo de controle'.

Análise léxica
Fase do processo de compilação responsável pela leitura dos caracteres do programa-fonte e o agrupamento e classificação de determinadas seqüências de caracteres do programa-fonte que se denominam átomos. A análise léxica produz como saída um fluxo de átomos para a fase de análise sintática.

Análise semântica
Fase do processo de compilação que verifica se os elementos da árvore abstrata satisfazem os aspectos dependentes de contexto especificados para a linguagem de

programação, ou as regras semânticas da linguagem, e registra informações essenciais para a síntese do código-objeto. A estrutura sintática resultante da análise semântica é conhecida por árvore sintática abstrata decorada e é a forma de representação intermediária pela qual se inicia a síntese do código-objeto.

Análise sintática
Fase do processo de compilação que verifica se o fluxo de átomos proveniente do passo de análise léxica produz estruturas sintaticamente válidas da linguagem de programação. Nos compiladores otimizadores, estas estruturas são agrupadas geralmente em uma estrutura de mais alto nível que representa o programa, a árvore sintática abstrata. Nem todas as implementações constróem fisicamente a árvore sintática abstrata, contudo, o princípio de construção da mesma se manifesta através da seqüência de derivações ou reduções que ativam rotinas de análise semântica que verificam aspectos dependentes de contexto especificados para a linguagem de programação e geram o código-objeto.

Aplicação embutida
Software que controla um sistema microcontrolado embutido. Ver também 'sistema microcontrolado embutido'.

Arquitetura de computador
Designa a estrutura geral, os componentes lógicos e o inter-relacionamento das partes constituintes de um processador. Ver também 'processador'.

Árvore sintática abstrata
Ver 'análise sintática'.

Árvore sintática abstrata decorada
Ver 'análise semântica'.

Átomo
Ver 'análise léxica'.

Basic block
Ver 'bloco básico'.

Bloco básico
Seqüência de instruções consecutivas que sempre são executadas do início até o fim.

Code generation
Ver 'geração de código'.

Código da operação
Elemento obrigatório de uma instrução de máquina, o código da operação descreve o tipo da operação a ser executada pelo processador. Algumas instruções necessitam de elementos adicionais denominados operandos. Ver também 'operação', 'operando' e 'processador'.

Código de máquina
Ver 'código-objeto'.

Código-fonte
Ver 'programa-fonte'.

Código intermediário
Ver 'representação intermediária'.

Código-objeto
Seqüência de operações executadas pelo processador. Também denominado de código de máquina. Ver 'operação' e 'processador'.

Comando
Ver 'sentença'.

Compilador
É um programa que analisa um programa-fonte e o transforma em código-objeto caso reconheça o programa-fonte como uma sentença válida da linguagem de programação. Ver também 'análise léxica', 'análise sintática', 'análise semântica', 'expansão de código', 'otimização' e 'geração de código'.

Compilador redirecionável
É um compilador capaz de gerar código-objeto para diferentes arquiteturas de computadores.

Compiler-compiler
Ver 'gerador de compiladores'.

Conjunto de instruções
Ver 'processador'.

Control-flow graph
Ver 'grafo de fluxo'.

Data-flow analysis
Ver 'análise do fluxo de dados'.

Editor de ligações (ou *linkage editor*)
É um programa que permite a criação de um único programa a partir de vários arquivos de código-objeto. Esses arquivos resultam da compilação em separado de vários arquivos de programas-fonte.

Embedded application
Ver 'aplicação embutida'.

Embedded microcontrolled system
Ver 'sistema microcontrolado embutido'.

Escalonamento das instruções
Passo de otimização do código-objeto que altera a ordem de execução das instruções do código-objeto a fim de utilizar a linha de montagem de instruções da forma mais eficiente possível.

Expansão de código
Fase, opcional, do processo de compilação em que se traduzem as construções específicas da linguagem representadas na árvore sintática abstrata decorada, em construções gerais para certas classes de arquitetura de computadores, que se implementam na forma de um código intermediário.

Fase
Em um compilador típico, o processo de compilação se desenvolve através de fases bem definidas. Em cada fase, ou se extraem informações do fluxo de dados de entrada para uso nas fases posteriores do processo de compilação, ou se efetua uma transformação do fluxo de dados de entrada para uma forma intermediária, mais conveniente para a fase subseqüente. Simplificadamente, pode-se dizer que as fases iniciais, até a análise semântica, efetuam o reconhecimento do programa-fonte, e as fases remanescentes se encarregam da geração do código-objeto equivalente ao programa-fonte. Não confundir com 'passo'. Ver 'passo'.

Fresta
Ver 'janela'.

Função
É uma declaração que associa um identificador e um enunciado. O identificador é o nome da função, sendo que ao nome da função se associa uma informação do tipo de valor retornado pela função após o término da sua execução. O enunciado é o corpo da função, sendo o enunciado formado por sentenças válidas da linguagem de programação.

Geração de código
Fase do processo de compilação que realiza a geração do código-objeto, ou seja, o texto de saída, denotado na linguagem de programação da máquina. Nesta fase, selecionam-se as instruções da máquina que correspondem às instruções do código intermediário ou cria-se o código-objeto diretamente, como decorrência do tratamento da varredura da árvore sintática abstrata decorada.

Geração do código intermediário
Ver 'expansão de código'.

Gerador de compiladores (ou *compiler-compiler*)
É um programa, ou um conjunto de ferramentas de programação, que processa uma especificação de uma linguagem de programação, ou uma especificação da arquitetura do computador, bem como um conjunto de regras de produção que descrevem o tipo de tradução que se deseja efetuar, e produz um programa capaz de executar alguma das fases do processo de compilação.

Grafo de fluxo de controle
Estrutura de dados que modela o fluxo de execução do programa-fonte. É empregada na fase de otimização global dos compiladores otimizadores.

Gramática
Formalismo empregado na descrição das sentenças válidas de uma linguagem a partir

do seu alfabeto.

Instrução
Ver 'operação'.

Instrução de máquina
Ver 'processador'.

Intermediate code
Ver 'código intermediário'.

Intermediate representation
Ver 'representação intermediária'.

Janela (ou fresta)
Seqüência de instruções consecutivas do código-objeto.

Ligação
É o ato de combinar vários arquivos de código-objeto em um único arquivo com o programa a ser executado.

Linguagem de montagem
Representação simbólica das instruções executáveis pelo processador, para facilitar a leitura e a compreensão do código-objeto pelos humanos.

Linguagem de programação
É toda linguagem que possui as seguintes características: (1) para utilizá-la, o usuário não precisa ter conhecimento do código de máquina, (2) pode ser convertida para código de máquina de diversos computadores, (3) origina, em geral, para cada comando da linguagem, uma seqüência de instruções em código-objeto e (4) tem uma notação mais próxima do domínio dos problemas a serem resolvidos do que a linguagem de montagem. Ver 'gramática'.

Linguagem de programação de alto nível
Linguagem de programação cujos comandos são mais próximos do domínio da aplicação que da linguagem de máquina. Ver 'linguagem de programação'.

Linha de montagem (ou *pipeline*)
Arquitetura de processadores que aumenta o número médio de instruções executadas em um dado intervalo de tempo através da paralelização, em cascata, de múltiplas operações elementares.

Link
Ver 'ligação'.

Linker , Linkage editor
Ver 'editor de ligações'.

Máquina real
Ver 'processador'.

Máquina virtual
Modelo idealizado da máquina responsável pela execução do código-objeto. A máquina virtual combina instruções, ou seqüência de instruções, em alguma forma de representação intermediária e interpreta-as acionando rotinas de apoio do ambiente de execução que provocam a execução dos comandos do programa-fonte na máquina.

Microcontrolador
O avanço da tecnologia de integração em grande escala de circuitos semicondutores possibilitou a integração de diversos dispositivos eletrônicos digitais em um único componente e deu origem, entre outros, aos microcontroladores. Um microcontrolador é o componente resultante da integração de um microprocessador e outros dispositivos auxiliares. Pode ser visto como um componente que agrega todo o hardware necessário para executar uma aplicação particular.

Microprocessador
Ver 'processador'.

Opcode
Ver 'código da operação'.

Operação
Indica o código da operação e os operandos necessários para a execução de uma dada tarefa em uma unidade funcional do processador. Ver também 'código da operação', 'operando' e 'processador'.

Operando
Quando presente em uma instrução, denota um elemento sobre o qual atua o código da operação, ou *opcode*. Ver também 'código da operação', 'operação' e 'processador'.

Otimização
Fase, opcional, do processo de compilação, que promove a geração de versões mais eficientes de código-objeto. Uma versão mais eficiente alcança os mesmos resultados que uma versão não otimizada, porém de forma mais rápida, ou usando menos memória, ou ambos. Algumas otimizações são independentes da arquitetura do computador e podem ser realizadas como transformações das formas intermediárias de representação do programa. Outras otimizações dependem da arquitetura do computador e devem ser realizadas como transformações do código-objeto. Assim, a otimização aparece como duas fases adicionais do processo de compilação, uma imediatamente após a fase de expansão de código, denominada de otimização global, e outra imediatamente após a geração do código-objeto, denominada de otimização local, ou do código-objeto.

Otimização do código objeto
Ver 'otimização'.

Otimização global
Ver 'otimização'.

Otimização local
Ver 'otimização'.

Otimização *peephole*
Técnica de otimização local que se aplica em uma vizinhança limitada do código-objeto, ou de formas intermediárias de representação do programa. Essa vizinhança vai sendo deslocada de forma que todo o código seja tratado. Nesta técnica se efetua a eliminação de instruções redundantes e a substituição de seqüências de instruções ineficientes por outras mais eficientes.

Passo
Um passo é composto por uma fase, ou um conjunto de fases, do processo de compilação, sendo que a execução dos passos em uma dada seqüência leva à compilação completa do programa-fonte. Os passos podem ser implementados como programas em separado com a realização da leitura dos dados de entrada de um arquivo e a gravação dos dados de saída em um outro arquivo distinto do primeiro. Até meados da década de 80 do século passado, os compiladores eram divididos em passos visando a máxima reutilização da memória do computador que executava a compilação do programa-fonte.

Peephole
Ver 'janela'.

Pilha de controle
É uma região especial da memória na qual se armazenam os registros de ativação. Os registros de ativação são armazenados na mesma ordem em que são inseridos, mas são retirados na ordem reversa. Em função deste modo de operação, as pilhas são chamadas de estruturas de dados do tipo LIFO, ou *Last-In, First-Out* ou último a entrar, primeiro a sair. O próximo registro de ativação a ser retirado está no local conhecido como topo da pilha. Normalmente, não há necessidade de se especificar um endereço para acessar os dados da pilha de controle, porque a arquitetura do computador provê um registrador que aponta para o topo da pilha de controle, denominado de apontador da pilha, bem como instruções especiais para a inserção e retirada de dados da pilha. Ver também 'ambiente de execução'.

Pipeline
Ver 'linha de montagem'.

Procedimento
É uma declaração que associa um identificador a um enunciado. O identificador é o nome do procedimento e o enunciado é o corpo do procedimento, sendo o enunciado formado por sentenças válidas da linguagem de programação. Em muitas linguagens, os procedimentos que retornam valores são chamados de funções. Pode-se considerar um programa completo como um procedimento.

Procedural abstraction
Ver 'abstração de procedimento'.

Processador
Circuito lógico digital construído com materiais semicondutores que executa as instruções que dirigem um computador, ou instruções de máquina. O processador executa operações aritméticas e lógicas utilizando as unidades funcionais disponíveis e um conjunto de dispositivos para armazenamento temporário de valores denominados registradores. O conjunto de todas as operações efetuadas pelo processador é denominado de conjunto de instruções do processador. O conjunto de instruções do processador e a organização dos seus registradores são elementos que definem a arquitetura do computador. O processador dos computadores pessoais ou de pequenos dispositivos eletrônicos é também chamado de microprocessador. Ver também 'operação' e 'unidade funcional'.

Programa-fonte
Codificar um programa-fonte é a forma usual de elaboração do software. O programa-fonte consiste em uma relação de comandos do programa, denotados em alguma linguagem de programação. Uma vez pronto, o programa é submetido ao compilador para análise e geração do código-objeto.

RAM
Read access memory, ou memória de acesso aleatório.

Reconhecedor
Dispositivo capaz de classificar como válidas todas as sentenças de uma linguagem, e nada mais.

Registrador
Ver 'processador'.

Registro de ativação
Estrutura de dados composta por uma área reservada para os objetos declarados no interior da rotina, outra para os parâmetros de chamada da rotina e outra relativa a informações de controle, como: endereço de retorno, apontadores para outros registros de ativação que a rotina pode acessar e valores de retorno. Ver também 'ambiente de execução'.

Regras de escopo
Regras semânticas que disciplinam o acesso a objetos do programa de acordo com o seu escopo de declaração e execução. Quando estas regras dependem apenas do exame isolado do programa-fonte, são chamadas de regras de escopo estático, ou léxico. Quando dependem de considerações em tempo de execução, são chamadas de regras de escopo dinâmico.

Representação intermediária
Forma interna de representação do programa-fonte utilizada nas diversas fases de análise e geração de um compilador.

Retargetable compiler
Ver 'compilador redirecionável'.

ROM
Read only memory, ou memória de apenas leitura.

Rotina
Ver 'procedimento'.

Run-time system
Ver 'ambiente de execução'.

Sentença
É uma seqüência finita de símbolos do alfabeto da linguagem e que obedece às regras que regem a estrutura das sentenças da linguagem.

Sistema microcontrolado embutido
É todo equipamento, ou sistema, eletrônico projetado e construído com microcontroladores. A utilização de microcontroladores visa simplificar o projeto e dar flexibilidade ao sistema, já que a incorporação de um dispositivo microprocessador permite, em geral, que a solução de problemas, a realização de modificações e o acréscimo de novas facilidades seja feito com a substituição do software que controla o equipamento. Os sistemas microcontrolados embutidos normalmente não possuem disco para armazenamento do software de controle, sendo o mesmo armazenado em algum tipo de memória não volátil, como ROM, EPROM, EEPROM, etc., o que requer a substituição ou reprogramação do componente.

Super-otimização
Técnica de otimização do código-objeto que visa determinar a seqüência de instruções mais eficiente possível que seja equivalente a uma seqüência que realiza uma dada função. A determinação da seqüência mais eficiente de instruções é feita por algum tipo de busca exaustiva.

Token
Ver 'átomo'.

Transformação otimizante
Ver 'otimização'.

Unidade funcional
Parte do processador que executa determinadas operações. Ver também 'operação' e 'processador'.

APÊNDICE A - Introdução à Teoria dos Autômatos Adaptativos

Na sua forma mais geral, uma transição adaptativa pode ser expressa da seguinte forma:

$(q, \sigma\alpha, g\gamma), A(\pi_1,...,\pi_n) \rightarrow (q', \sigma'\alpha, g'\gamma), D(\pi'_1,...,\pi'_{n'})$

onde α e γ são meta-símbolos que representam o conteúdo da cadeia de entrada e da pilha ignorados pelo autômato e não afetam a execução da transição. q é o estado do autômato antes da transição e q' o estado do autômato após a execução da transição. σ é o símbolo da cadeia de entrada consumido pelo autômato e σ' o símbolo depositado na cadeia de entrada pelo autômato após a execução da transição. g e g' representam o conteúdo do topo da pilha antes e depois da execução da transição respectivamente. σ', g, g' são opcionais e representam a cadeia vazia quando omitidos.

A e *D* são chamadas, opcionais, de funções que executam as ações adaptativas, ou *funções adaptativas*, com argumentos de entrada $\pi_1,...,\pi_n$ e $\pi'_1,...,\pi'_{n'}$ respectivamente. *A* é uma função adaptativa executada *antes* de efetuada a transição e *D* é uma função adaptativa executada *depois* de efetuada a transição.

Portanto, no caso em que $\sigma' = g = g' = \varepsilon$ e *A* e *D* não se aplicam, a transição adaptativa assume uma forma particular, operando como uma transição convencional de um autômato finito:

$(q, \sigma) \rightarrow q'$

Uma função adaptativa *F* com *parâmetros* de entrada $\rho_1,...,\rho_n$ é declarada da seguinte maneira:

$F(\rho_1,...,\rho_n)$ = {<declaração de nomes> : <declaração de ações>}

onde <declaração de nomes> é uma lista de identificadores, opcional, que representa objetos no interior do corpo da função e <declaração de ações> é uma lista de *ações adaptativas elementares* que pode ser precedida pela chamada, opcional, de uma função adaptativa denominada de *ação adaptativa inicial* e seguida pela chamada de uma outra função adaptativa denominada de *ação adaptativa final.*

Os parâmetros recebem os valores atribuídos aos argumentos antes do início da execução da função. Cada parâmetro ρ_i assume o valor de π_i, onde i representa a posição do argumento na lista de chamada, e permanece invariável até o final da

execução da função.

A <declaração de nomes> assume a seguinte forma: $v_1,...,v_n$, $g_1*,...,g_n*$ onde cada identificador seguido por asterisco denota um *gerador*, enquanto os demais denotam *variáveis*. Os valores dos geradores são atribuídos uma única vez ao início da execução da função adaptativa e permanecem invariáveis até o seu final. Os valores das variáveis são atribuídos como efeito da chamada das ações adaptativas elementares de inspeção e eliminação, conceituadas a seguir.

A ação adaptativa inicial é executada logo após a passagem dos parâmetros e antes do início da execução da função adaptativa. A ação adaptativa inicial pode consultar os valores dos parâmetros, mas não tem acesso aos valores das variáveis e dos geradores, cujos valores ainda não estão definidos.

A ação adaptativa final é executada imediatamente antes do término da execução da função e pode consultar os valores dos parâmetros, das variáveis e dos geradores.

As ações adaptativas elementares têm a seguinte forma:

prefixo[regra de produção]

onde prefixo é uma das três ações adaptativas elementares de: inspeção (?), eliminação (-) ou inserção (+) e regra de produção é a regra à qual se aplica a ação adaptativa elementar.

Uma regra de produção assume uma das seguintes formas:

$(q, \sigma\alpha, g\gamma), A(\pi_1,...,\pi_n) \rightarrow (q', \sigma'\alpha, g'\gamma), D(\pi'_1,...,\pi'_{n'})$ ou

$(q, \sigma\alpha, g\gamma), A(\pi_1,...,\pi_n) \rightarrow (q', \sigma'\alpha, g'\gamma)$ ou

$(q, \sigma\alpha, g\gamma) \rightarrow (q', \sigma'\alpha, g'\gamma), D(\pi'_1,...,\pi'_{n'})$ ou

$(q, \sigma\alpha, g\gamma) \rightarrow (q', \sigma'\alpha, g'\gamma)$ ou

$(q, \sigma), A(\pi_1,...,\pi_n) \rightarrow q', D(\pi'_1,...,\pi'_{n'})$ ou

$(q, \sigma), A(\pi_1,...,\pi_n) \rightarrow q'$ ou

$(q, \sigma) \rightarrow q', D(\pi'_1,...,\pi'_{n'})$ ou

$(q, \sigma) \rightarrow q'$

onde q, σ, g, q', σ', g', bem como o nome e os argumentos das funções adaptativas A e D podem ser constantes ou variáveis.

A ação adaptativa elementar de inspeção (?) procura regras de produção da forma apresentada entre colchetes. Com exceção das variáveis, os demais elementos

parametrizáveis da regra de produção, ou seja, as constantes, os parâmetros e os geradores, estão todos definidos no instante da aplicação da ação de inspeção. Por este mecanismo, as variáveis utilizadas na descrição da regra de produção recebem o valor correspondente ao da regra encontrada. Caso não seja encontrada nenhuma regra no conjunto de produções, as variáveis permanecem com valores indefinidos.

A ação adaptativa elementar de eliminação (-) efetua uma procura de regras de produção da forma apresentada entre colchetes e as elimina do conjunto de produções do autômato. Todos os elementos parametrizáveis da regra de produção devem estar definidos no instante da sua aplicação, do contrário a ação será ignorada.

A ação adaptativa elementar de inserção (+) efetua uma inclusão de regras de produção da forma apresentada entre colchetes no conjunto de produções do autômato. Da mesma forma que no caso anterior, todos os elementos parametrizáveis da regra de produção devem estar definidos no instante da sua aplicação para que a ação adaptativa elementar surta efeito.

Define-se uma configuração t_k do autômato adaptativo no instante $k \geq 0$ como uma quádrupla $(E_k, q_k, \omega_k, \gamma_k)$, onde:

E_k é a máquina de estados que implementa o autômato adaptativo no instante k;

q_k é o estado atual de E_k no instante k;

ω_k é a cadeia de entrada a ser processada por E_k no instante k e

γ_k é o conteúdo da pilha no instante k.

Para k = 0, a configuração inicial do autômato é $t_0 = (E_0, q_0, \omega_0, \gamma_0)$, onde:

E_0 é a máquina de estados inicial do autômato adaptativo;

q_0 é o estado inicial de E_0 e do autômato adaptativo;

$\omega_0=\omega$, a cadeia de entrada completa que o autômato adaptativo irá processar e

$\gamma_0=Z_0$, ou m vazia.

Da mesma forma define-se $t_f = (E_f, q_f, \omega_f, \gamma_f)$, onde:

E_f é a máquina de estados do autômato adaptativo após consumo completo da cadeia de entrada ω;

q_f é um dos estados finais de E_f se e somente se ω é uma sentença aceita pelo autômato;

$\omega_f=\varepsilon$, em outras palavras a cadeia de entrada foi consumida completamente pelo autômato adaptativo e

$\gamma_f = Z_0$, ou marcador de pilha vazia.

Finalmente, diz-se que o autômato reconhece uma sentença válida da linguagem quando, partindo de t_0, o autômato atinge a configuração t_f após $|\omega|$ passos de reconhecimento, onde $|\omega|$ é o comprimento da cadeia de entrada.

APÊNDICE B - Arquitetura x86

Introdução

A família de processadores da arquitetura x86, encabeçada pelo microprocessador Intel 386 e incluindo os seus sucessores, 486, Pentium, Pentium II, Pentium III, Pentium IV, etc., implementa o mesmo conjunto de instruções básico, mas, freqüentemente, de formas diferentes, a fim de obter reduções significativas do tempo de execução de instrução em relação aos seus antecessores na família. A arquitetura é restringida pela necessidade de um novo membro ser compatível com os microprocessadores primeiramente desenvolvidos, como o 8086, que manipulam dados com apenas 8 ou 16 bits de comprimento e possuem um esquema peculiar de endereçamento por segmentos (Muchnick, 1997).

Registradores

Nos processadores desta família existem oito registradores com 32 bits de comprimento para a realização de operações lógicas e aritméticas com números inteiros denominados EAX, EBX, ECX, EDX, EBP, ESP, ESI e EDI. Os 16 bits menos significativos de cada registrador recebem um segunda denominação, e são usados pelo subconjunto de instruções derivado do 8086 que faz uso de registradores com 16 bits de comprimento. A denominação dos registradores com 16 bits de comprimento é igual à denominação dos registradores com 32 bits de comprimento, mas sem o prefixo 'E', ou AX, BX, CX, DX, BP, SP, SI e DI. Além disso, os quatro primeiros registradores com 16 bits de comprimento são subdivididos em dois registradores com 8 bits de comprimento, como AH e AL que compreendem, respectivamente, o byte mais significativo do registrador AX e o byte menos significativo do mesmo registrador. Alguns registradores têm uma função específica, como EBP e ESP, que apontam, respectivamente, para o registro de ativação corrente e o topo da pilha de controle, enquanto outros são utilizados em certas classes de instruções, como ECX, ESI e EDI durante a execução de instruções de manipulação de cadeias de caracteres.

A arquitetura x86 provê seis registradores de segmento para o endereçamento de memória, denominados CS, DS, SS, ES, FS e GS, um apontador de instruções e

um registrador de bits indicadores[61] denominados, respectivamente, EIP e EFlags, no 386 e posteriores, e IP e Flags, no 8086 e assemelhados, para a realização de desvios condicionais[62]. Um endereço de memória é formado por um registrador de segmento selecionado, em muitos casos, pelo tipo da instrução, um registrador base e/ou um registrador de índice (multiplicado, opcionalmente, no modo protegido, por um fator de escala de 2, 3 ou 4) e/ou um deslocamento de 8 a 32 bits, sendo a presença de, pelo menos, um deles obrigatória no endereço. A realização de desvios condicionais é feita pela execução de instruções de desvio que consultam o estado de acionamento dos bits indicadores, que resultam, por sua vez, da execução de instruções que afetam os mesmos, principalmente, operações aritméticas e lógicas. A figura B.1 mostra o conjunto de registradores da arquitetura x86.

---

[61] *Flags*

[62] Os principais bits indicadores são: OF, DF, IF, TF, SF, ZF, AF, PF e CF. Os indicadores OF (*overflow flag*), SF (*signal flag*), ZF (*zero flag*), AF (*auxiliary flag*), PF (*parity flag*) e CF (*carry flag*) são os indicadores de estado. São atualizados durante a execução de instruções aritméticas e lógicas de acordo com o resultado produzido. DF (*direction flag*), IF (*interrupt enable flag*) e TF (*trap flag*) são indicadores de controle que afetam o comportamento do processador. Por exemplo, o armazenamento de 1 no IF, habilita o atendimento de interrupções de hardware.

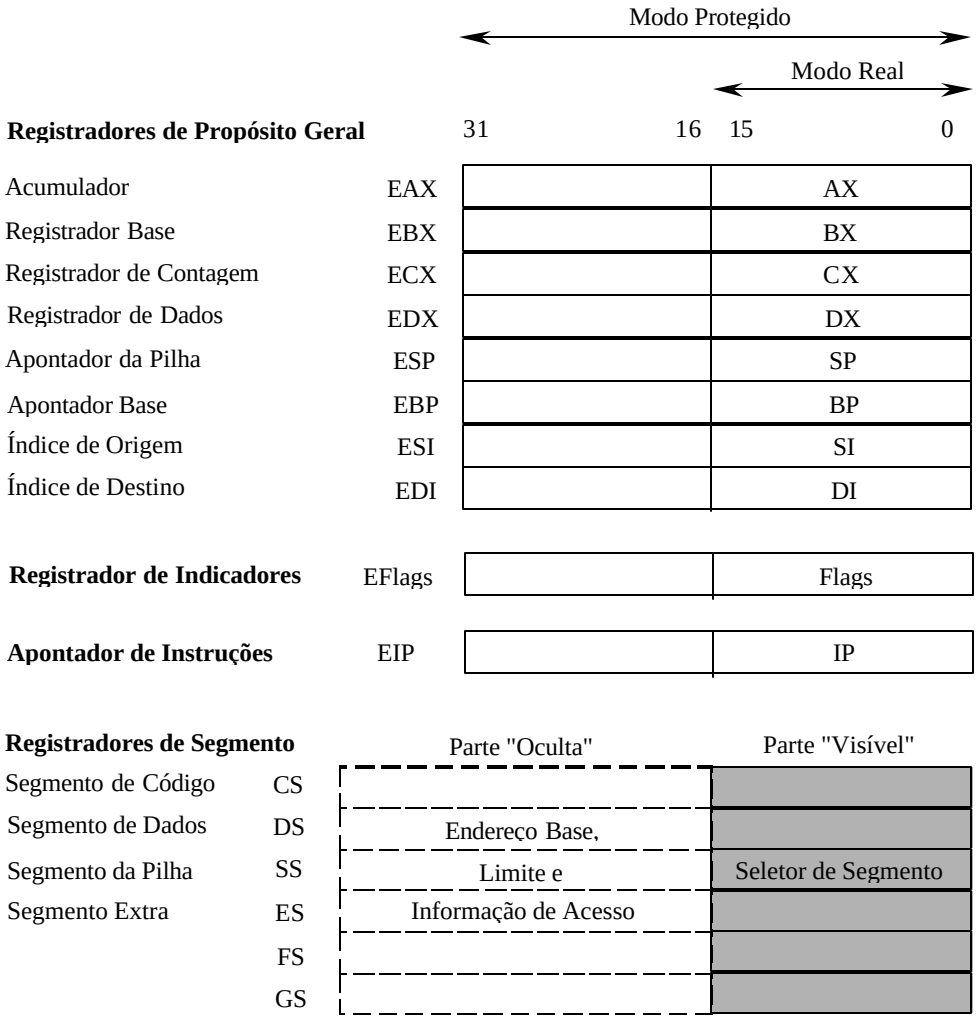

Figura B.1 - Conjunto de registradores da arquitetura x86
Adaptada de Lewis (2002)

Formato das Instruções

Na linguagem de montagem da arquitetura x86, uma instrução é composta por um campo opcional de rótulo, terminado por dois pontos, um campo de código da instrução, ou *opcode*, obrigatório, um campo de operandos, opcional ou obrigatório dependendo da instrução associada, sendo o campo de operandos separado do campo de instrução por, ao menos, um espaço em branco e os operandos individuais separados por vírgulas, e um campo opcional de comentário, iniciado por ponto-e-vírgula. Grande parte das instruções possuem dois operandos associados, embora haja uma quantidade significativa de instruções com nenhum, ou com um único operando associado. Nas instruções com dois operandos, o *opcode* atua sobre o operando de origem (o segundo operando da instrução) e, quando necessário, sobre o

operando de destino (o primeiro operando da instrução) e armazena o resultado no operando de destino. Nas instruções com um único operando, considera-se o operando da instrução simultaneamente como operando de origem e operando de destino.

Tipos de Operandos

Permite-se a utilização de quatro tipos de operandos nas instruções da linguagem de montagem: (1) uma constante, (2) o conteúdo de um registrador, (3) o conteúdo de uma posição de memória ou (4) o conteúdo de um dispositivo de entrada/saída. Pode-se usar como operando de origem qualquer um dos quatro tipos de operandos, mas somente os tipos (2), (3) e (4) como operandos de destino. O conteúdo de uma posição de memória é acessível após a determinação do endereço da posição de memória. A arquitetura fornece vários métodos de endereçamento da memória, também denominados 'modos de endereçamento', discutidos mais adiante.

Restrições Aplicáveis aos Operandos

Nas instruções com dois operandos, a arquitetura estabelece algumas restrições quanto às combinações permitidas dos operandos de origem e destino. A primeira restrição estabelece que os operandos de origem e destino devem possuir o mesmo comprimento, ou seja, ambos devem possuir 8, ou 16, ou 32 bits de comprimento. A segunda restrição estabelece que tanto o operando de origem, quanto o operando de destino podem referenciar o conteúdo de uma posição de memória, mas não ambos simultaneamente. E a terceira restrição estabelece que o uso dos registradores de segmento, excetuando-se CS, (DS, SS, ES, FS ou GS) é limitado às instruções de armazenamento de dados, sendo permitido, neste caso, o uso de um registrador de segmento como operando de origem ou operando de destino, mas não ambos simultaneamente e, caso o operando de destino seja um registrador de segmento, não é permitido o uso de uma constante como operando de origem.

Além dessas restrições, a arquitetura não efetua automaticamente a compatibilização de operandos de comprimentos distintos, fornece, contudo, instruções que estendem operandos de comprimento menor para os operandos de comprimento maior, ou seja, de operandos de origem com comprimento de 8 bits para operandos de destino com 16, ou 32, bits e de operandos de origem com comprimento de 16 bits para operandos de destino com 32 bits.

Estas restrições são aplicadas mais adiante durante o estudo do 'conjunto de instruções' da arquitetura.

Modo Real e Modo Protegido

A fim de assegurar a execução do código objeto originalmente desenvolvido para os primeiros microprocessadores 8086 pelos novos processadores da arquitetura x86, incluiu-se em todos os processadores da arquitetura x86 a capacidade de emular o funcionamento e as limitações do microprocessador 8086 original que se denominou de operação em modo real.

A característica mais peculiar da operação em modo real é o método de endereçamento da memória por meio da combinação de dois números de 16 bits de comprimento, um chamado de *segmento* e outro de *deslocamento*, usando um esquema denominado de endereçamento por segmentos. O valor do segmento é fornecido por um dos registradores de segmento (CS, DS, SS, ES), enquanto o valor do deslocamento é determinado por um dos modos de endereçamento no modo real. Isto permitiu que o microprocessador de 16 bits 8086 pudesse endereçar até 1.048.576 posições de memória com 8 bits de comprimento, ou 1 MB, com um barramento de dados de 20 bits, em oposição às 65.536 posições de memória com 8 bits de comprimento, ou 64 KB, possíveis com um barramento de dados de 16 bits. A figura B.2 ilustra o esquema de endereçamento por segmentos no modo real.

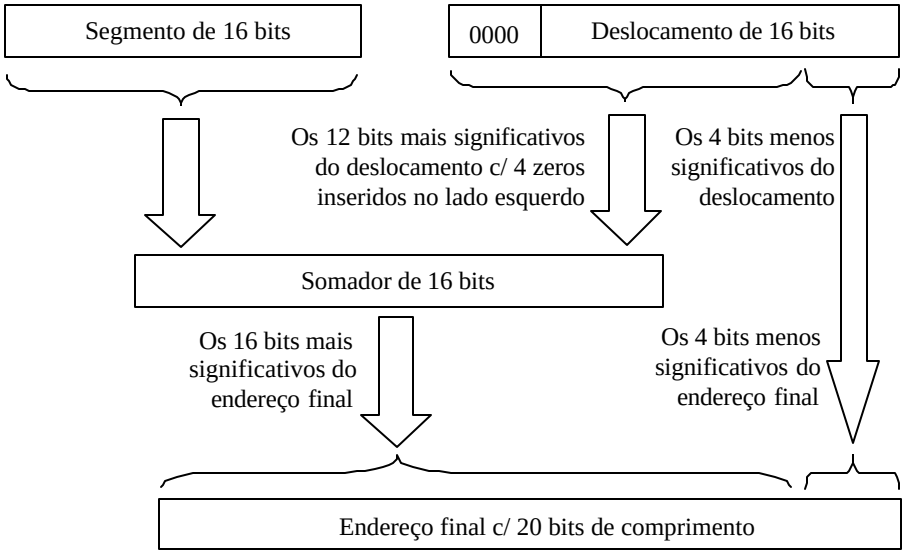

Figura B.2 - Esquema de endereçamento por segmentos no modo real
Adaptada de Lewis (2002)

Por este esquema, é possível endereçar até 64 KB do espaço total de endereçamento de 1 MB mantendo-se inalterado o valor do registrador de segmento, sendo o endereço da posição inicial da memória igual ao valor do registrador de segmento multiplicado por 16 adicionado a um deslocamento zero, ou "segmento:0000h" na notação "<segmento>:<deslocamento>", e o endereço final igual ao valor do registrador de segmento multiplicado por 16 adicionado a um deslocamento FFFFh, ou "segmento:FFFFh", variando-se o valor do registrador de segmento pode-se acessar o espaço total de endereçamento de 1 MB. A escolha do registrador de segmento depende normalmente do contexto de execução da instrução. Por exemplo, o registrador CS é sempre combinado com o deslocamento presente no registrador IP durante o ciclo de busca da instrução; o registrador SS é sempre combinado com o deslocamento presente no registrador SP durante a inserção e retirada de dados da pilha de controle; as demais operações de acesso à memória utilizam o registrador DS com uma exceção: se o registrador BP é usado no cálculo do deslocamento, utiliza-se o registrador de segmento SS na composição do endereço da memória. No caso particular do registrador DS, pode-se utilizar um prefixo de substituição de segmento para se trocar o registrador de segmento default, DS, usado na composição do endereço por um outro registrador de segmento, como, CS, ES, ou SS, por exemplo.

Livre das limitações impostas pela operação em modo real, a operação em modo protegido permite o endereçamento de 4.294.967.296 posições de memória com 8 bits de comprimento, ou 4 GB, com um barramento de dados de 32 bits. No modo protegido, os registradores de segmento são divididos em duas partes: a parte visível e a parte oculta, conforme mostrado na figura B.1. A parte visível, também denominada de seletor de segmento, é acessível para programação e possui 16 bits de comprimento. A parte oculta não é acessível para programação e possui 32 bits de comprimento. O conteúdo do seletor de segmento é usado como um deslocamento para um vetor de descritores de segmento, residente na memória principal, denominado tabela de descritores de segmento, ou GDT[63]. Cada descritor contém informações sobre uma região da memória denominada segmento, incluindo-se o seu

[63] *Global descriptor table*

endereço inicial, o seu comprimento e direitos de acesso. A inclusão de informações de comprimento e direitos de acesso aos descritores de segmento permite que o processador proteja o segmento contra acessos indevidos, daí a denominação de modo protegido, e assegure uma operação mais confiável. Diferentemente do modo real, os segmentos não estão limitados a 64 KB, podendo abranger até 4 GB. Sempre que um registrador de segmento tem armazenado um novo valor no seu seletor de segmento, o processador efetua a leitura da posição da GDT indicada pelo seletor de segmento e armazena o endereço inicial do segmento na parte oculta do registrador de segmento. Combinando o endereço encontrado na parte oculta do registrador de segmento com um deslocamento de 32 bits produzido por um dos modos de endereçamento no modo protegido, determina-se a posição de memória desejada. A escolha do registrador de segmento no modo protegido é análoga ao modo real. A figura B.3 ilustra o esquema de endereçamento da memória no modo protegido.

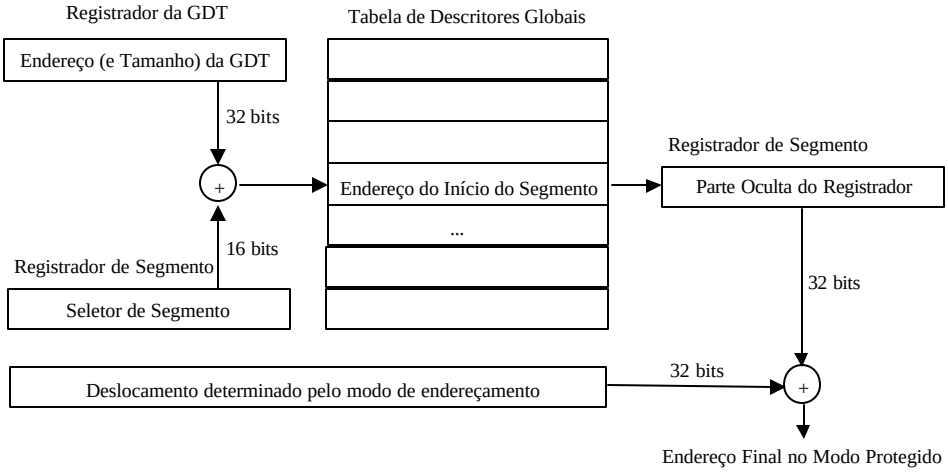

Figura B.3 - Esquema de endereçamento da memória no modo protegido
Adaptada de Lewis (2002)

Modos de Endereçamento no Modo Real

No modo real, calculam-se os deslocamentos por meio de uma combinação de ao menos um dos seguintes valores: (1) um valor residente em um registrador base, ou seja, BX ou BP. Quando se utiliza o registrador BP, assume-se temporariamente que o registrador de segmento default é SS no lugar de DS; (2) um valor residente em um registrador índice, ou seja, SI ou DI; (3) uma constante com 8, ou 16, bits de comprimento armazenada no interior da instrução. A figura B.4 mostra o cálculo do deslocamento no modo real e a tabela B.1 apresenta os modos de endereçamento

resultantes.

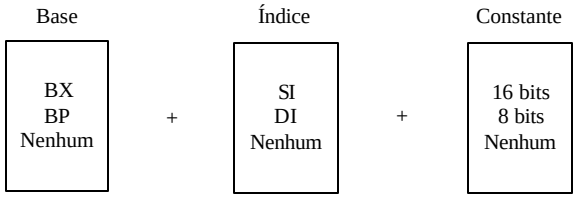

Figura B.4 - Cálculo do deslocamento no modo real
Adaptada de Lewis (2002)

Modo de Endereçamento	Deslocamento calculado a partir de:
Direto	Uma constante armazenada no interior da instrução
Indireto por Registrador	Um valor de um registrador base (BX, BP) ou índice (SI, DI)
Base	Um valor de um registrador base (BX, BP) + uma constante
Indexado	Um valor de um registrador índice (SI, DI) + uma constante
Base-Indexado	Um valor de um registrador base (BX, BP) + um valor de um registrador índice (SI, DI) + uma constante

Tabela B.1 - Modos de endereçamento no modo real
Adaptada de Lewis (2002)

Modos de Endereçamento no Modo Protegido

No modo protegido, o cálculo do deslocamento foi estendido com a permissão para utilização de quase todos os registradores como registrador base ou índice, sendo que o valor do registrador índice pode ser multiplicado por um fator de escala de 1, 2, 3 ou 4, para facilitar o endereçamento de vetores contendo elementos com 1, 2, 3 ou 4 bytes de comprimento. Assim, calculam-se os deslocamentos por meio de uma combinação de ao menos um dos seguintes valores: (1) um valor residente em qualquer um dos registradores base, ou seja, EAX, EBX, ECX, EDX, EDI, ESI, EBP, ou ESP. Quando se utilizam os registradores EBP ou ESP, assume-se temporariamente que o seletor de segmento default é SS no lugar de DS; (2) um valor residente em qualquer um dos registradores índice, excetuando-se ESP, ou seja, EAX, EBX, ECX, EDX, EDI, ESI, ou EBP, multiplicado por um fator de escala de 1, 2, 3 ou 4; (3) uma constante com 8, ou 16, ou 32, bits de comprimento armazenada no interior da instrução. A figura B.5 mostra o cálculo do deslocamento no modo protegido.

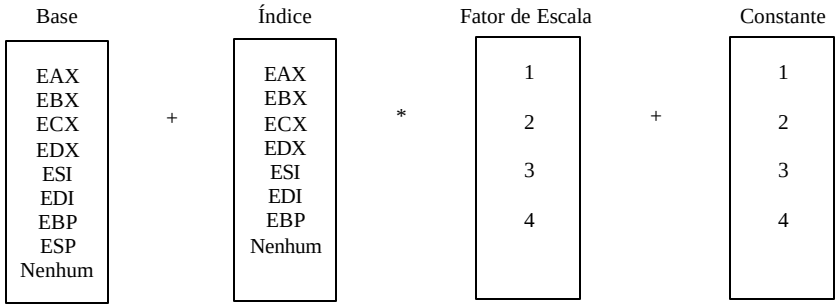

Figura B.5 - Cálculo do deslocamento no modo protegido
Adaptada de Lewis (2002)

Conjunto de Instruções

O conjunto de instruções da arquitetura possui instruções de armazenamento de dados (entre dois registradores ou entre um registrador e uma posição de memória), operações lógicas e aritméticas e de deslocamento e rotação de bits de números inteiros, instruções para a realização de desvios condicionais e incondicionais, instruções para a inserção e retirada de dados da pilha de controle, instruções para a ativação e o término da execução de procedimentos, instruções para o controle de execução de enlaces e a manipulação de cadeias de caracteres, instruções para a realização de entrada e saída e instruções para controle da operação do processador.

As operações matemáticas com números de ponto flutuante são realizadas por um co-processador matemático, externo no 386 e interno no 486 e seguintes. A arquitetura do co-processador matemático é organizada em torno de uma pilha de oito registradores de ponto flutuante com 80 bits de comprimento. Todas as operações são realizadas no formato de precisão estendida recomendado pela norma IEEE 754 (IEEE, 1985), (Goldberg, 1991), o que requer a conversão de números de ponto flutuante com 32 bits de comprimento no formato de precisão simples e de números de ponto flutuante com 64 bits de comprimento no formato de precisão dupla para o formato de precisão estendida e vice-versa. Além das instruções para o armazenamento de dados entre o co-processador e a memória e para a realização de operações aritméticas, de comparação e de conversão de números de ponto flutuante, a arquitetura fornece instruções para o armazenamento de constantes notáveis (+0.0, +1.0, π, $\log_2 10$, $\log_2 e$, $\log_{10} 2$, $\log_e 2$) e a realização de operações trigonométricas, exponenciais e logarítmicas (Brey, 2002).

A apresentação do conjunto completo de instruções da arquitetura está além do

escopo deste trabalho. Contudo, as tabelas B.2 a B.8 descrevem um subconjunto de instruções da arquitetura necessário para a compreensão das regras de otimização empregadas pelo otimizador *peephole* adaptativo em linguagem de montagem[64]. Entre as instruções excluídas deste anexo, estão as instruções para a realização de entrada e saída, as instruções para controle da operação do processador, as operações matemáticas com números de ponto flutuante e as instruções de apoio às extensões multimídia, ou MMX, de *multimedia extensions*. Para uma descrição mais detalhada da arquitetura x86 e a apresentação do conjunto de instruções completo, recomenda-se a leitura dos manuais que descrevem a arquitetura (Intel, 1997a), (Intel, 1997b), (Intel, 1997c). Sobre a linguagem de montagem, recomenda-se a leitura dos manuais das ferramentas de desenvolvimento (Intel, 1995) ou de bons livros-textos que abordam o tema (Brey, 2002) e (Lewis, 2002).

Instruções para Armazenamento de Dados

Instrução		Operação	Flags afetados
MOV	dst,org	dst ← org	-
MOVZX	reg_{16},org_8	reg ← org com extensão de zeros	-
MOVZX	reg_{32},org_8		
MOVZX	reg_{32},org_{16}		
MOVSX	reg_{16},org_8	reg ← org com extensão de sinal	-
MOVSX	reg_{32},org_8		
MOVSX	reg_{32},org_{16}		
LEA	reg_{32},mem	reg_{32} ← deslocamento(mem)	-
XCHG	dst,org	temp ← dst; dst ← org; org ← temp	-

Tabela B.2 - Instruções para armazenamento de dados
Adaptada de Lewis (2002)

[64] Abreviaturas empregadas
acc:um dos registradores acumuladores AL, AX ou EAX
dst: um registrador ou posição de memória de destino
org: um registrador, posição de memória ou constante de origem
reg: qualquer registrador diferente de um registrador de segmento
regseg: parte visível de CS, DS, SS, ES, FS ou GS
con: constante
mem: posição de memória
gdt: *global descriptor table*
rótulo: rótulo (posição de memória)

Instruções para Operações Aritméticas com Inteiros

Instrução		Operação	Flags afetados
ADD	dst,org	dst←dst + org	OF, SF, ZF, AF, CF, PF
ADC	dst,org	dst←dst + org + CF	
SUB	dst,org	dst←dst - org	
SBB	dst.org	dst←dst - org - CF	
CMP[65]	dst,org	dst - org; descarta o resultado numérico	
INC	dst	dst←dst + 1	OF, SF, ZF, AF, PF
DEC	dst	dst←dst - 1	
NEG	dst	dst←-dst	OF, SF, ZF, AF, PF; CF=0 sse[66] dst = 0
MUL	org_8	AX←AL x org_8	Posiciona CF e OF em caso de *overflow*
IMUL[67]	org_8		
MUL	org_{16}	DX.AX←AX x org_{16}	Posiciona CF e OF em caso de *overflow*
IMUL	org_{16}		
MUL	org_{32}	EDX.EAX←EAX x org_{32}	Posiciona CF e OF em caso de *overflow*
IMUL	org_{32}		
DIV	org_8	AL←quociente(AX÷org_8)	OF, SF, ZF, AF, CF, PF indefinidos
IDIV[68]	org_8	AH←resto(AX÷org_8)	
DIV	org_{16}	AX←quociente(DX.AX÷org_{16})	OF, SF, ZF, AF, CF, PF indefinidos
IDIV	org_{16}	DX←resto(DX.AX÷org_{16})	
DIV	org_{32}	EAX←quociente(EDX.EAX÷org_{32})	OF, SF, ZF, AF, CF, PF indefinidos
IDIV	org_{32}	EDX←resto(EDX.EAX÷org_{32})	
CBW		AX←AL com extensão de sinal	-
CWD		DX.AX←AX com extensão de sinal	
CDQ		EDX.EAX←EAX com extensão de sinal	
CWDE		EAX←AX com extensão de sinal	

Tabela B.3 - Instruções para operações aritméticas com inteiros
Adaptada de Lewis (2002)

[65] Instrução normalmente usada antes de uma instrução de salto condicional

[66] sse: se e somente se

[67] MUL é usada para operandos não assinalados. IMUL é usada para operandos assinalados. Estes são os formatos mais comuns das instruções. Outras variantes são possíveis.

[68] DIV é usada para operandos não assinalados. IDIV é usada para operandos assinalados. Estes são os formatos mais comuns das instruções. Outras variantes são possíveis.

Instruções para Operações Lógicas

Instrução		Operação	Flags afetados
AND OR XOR TEST	dst,org dst,org dst,org dst.org	dst←dst & org dst←dst | org dst←dst ^ org dst & org	SF, ZF, PF; (OF e CF reposicionados, AF indefinido)
NOT	dst	dst← ~dst	-
SHL SHL	dst,con dst,CL	dst←dst << con/CL. Deslocamento lógico p/a esquerda: preenche o LSB[69] com zeros	SF, ZF, PF; CF armaz. o último bit deslocado; AF e OF indef.
SAL SAL	dst,con dst,CL	dst←dst << con/CL. Deslocamento aritmético p/a esquerda: preenche o LSB com zeros	SF, ZF, PF; CF armazena últ. bit deslocado; AF e OF indef.
ROL ROL	dst,con dst,CL	dst←dst << con/CL. Rotação à esquerda de N bits: preenche o LSB com o valor anterior do MSB [70]	CF armazena último bit deslocado; AF e OF indef.
RCL RCL	dst,con dst,CL	dst←dst << con/CL. Rotação à esquerda de N+1 bits: preenche o LSB com o valor anterior do CF	CF armazena último bit deslocado; AF e OF indef.
SHR SHR	dst,con dst,CL	dst←dst >> con/CL. Deslocamento lógico p/a direita: preenche o MSB com zeros	SF, ZF, PF; CF armaz. o último bit deslocado; AF e OF indef.
SAR SAR	dst,con dst,CL	dst←dst >> con/CL. Deslocamento aritmético p/a direita: preenche o MSB com zeros	SF, ZF, PF; CF armaz. o último bit deslocado; AF e OF indef.
ROR ROR	dst,con dst,CL	dst←dst >> con/CL. Rotação à direita de N bits: preenche o MSB com o valor anterior do LSB	CF armazena último bit deslocado; AF e OF indef.
RCR RCR	dst,con dst,CL	dst←dst >> con/CL. Rotação à direita de N+1 bits: preenche o MSB com o valor anterior do CF	CF armazena último bit deslocado; AF e OF indef.
CLC STC CMC		CF←0 CF←1 CF←~CF	-

Tabela B.4 - Instruções para operações lógicas
Adaptada de Lewis (2002)

[69] *Least significant bit*, ou o bit menos significativo.
[70] *Most significant bit*, ou o bit mais significativo.

Instruções para Inclusão e Retirada de Dados da Pilha de Controle

Instrução	Operação	Flags afetados
PUSH BYTE con_8	ESP←ESP-4; mem_{32}[ESP]←con_8 c/sinal estendido	-
PUSH WORD con_{16}	ESP←ESP-2; mem_{16}[ESP]←con_{16}	-
PUSH DWORD con_{32}	ESP←ESP-4; mem_{32}[ESP]←con_{32}	-
PUSH org_{16} PUSH org_{32}	ESP←ESP-sizeof(org); mem[ESP]←org	-
PUSH regseg	ESP←ESP-4; mem_{32}[ESP]←$regseg_{16}$ (c/ extensão de zeros)	-
POP dst_{16} POP dst_{32}	dst←mem[ESP]; ESP←ESP+sizeof(dst)	-
POP regseg	$regseg_{visível}$←mem_{32}[ESP] (descarta MSW[71]); ESP←ESP+4; $regseg_{culto}$←gdt_{64}[$regseg_{visível}$]	-
PUSHF POPF	ESP←ESP-4; mem_{32}[ESP]←EFlags EFlags←mem_{32}[ESP]; ESP←ESP+4	- -
PUSHA	Empilha EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI	-
POPA	Desempilha EDI, ESI, EBP, ESP, EBX, EDX, ECX, EAX	-
ENTER con_{16},0	Empilha EBP, EBP←ESP, ESP←ESP-con_{16}	-
LEAVE	ESP←EBP; desempilha EBP	-

Tabela B.5 - Instruções p/ inclusão e retirada de dados da pilha de controle
Adaptada de Lewis (2002)

Instruções para Ativação/Término de Chamada de Procedimentos

Instrução	Operação	Flags afetados
CALL rótulo	ESP←ESP-4; mem_{32}[ESP]←EIP; EIP←con_{32}(rótulo)	-
RET	EIP←mem_{32}[ESP]; ESP←ESP+4	-

Tabela B.6 - Instruções p/ ativação/término de chamada de procedimentos
Adaptada de Lewis (2002)

[71] *Most significant word*, a palavra mais significativa.

Instruções para Execução de Desvios Condicionais e Incondicionais

Instrução		Operação	Flags afetados
JMP	rótulo	Salta para rótulo	-
JA/JNBE	rótulo[72]	Salta qdo. maior/Salta qdo. não menor ou igual	-
JAE/JNB	rótulo	Salta qdo. maior ou igual/Salta qdo. não menor	
JBE/JNA	rótulo	Salta qdo. menor ou igual/Salta qdo. não maior	
JB/JNAE	rótulo	Salta qdo. menor/Salta qdo. não maior ou igual	
JG/JNLE	rótulo[73]	Salta qdo. maior/Salta qdo. não menor ou igual	-
JGE/JNL	rótulo	Salta qdo. maior ou igual/Salta qdo. não menor	
JLE/JNG	rótulo	Salta qdo. menor ou igual/Salta qdo. não maior	
JL/JNGE	rótulo	Salta qdo. menor/Salta qdo. não maior ou igual	
JE/JZ	rótulo[74]	Salta qdo. igual/Salta qdo. zero (ZF=1)	-
JNE/JNZ	rótulo	Salta qdo. não igual/Salta qdo. não zero (ZF=0)	
JC	rótulo	Salta qdo. CF=1	-
JNC	rótulo	Salta qdo. CF=0	
JS	rótulo	Salta qdo. SF=1	
JNS	rótulo	Salta qdo. SF=0	
JECXZ	rótulo	Salta qdo. ECX=0	
LOOP[75]	rótulo	ECX←ECX-1; Salta qdo. ECX ≠ 0	

Tabela B.7 - Instruções p/ execução de desvios condicionais e incondicionais
Adaptada de Lewis (2002)

Instruções para Execução de Enlaces com Manipulação de Cadeias

Instrução	Operação	Flags afetados
MOVSB MOVSW MOVSD	mem[EDI]←mem[ESI] Se DF=0 então ESI←ESI+1/2/4, EDI←EDI+1/2/4 Se DF=1 então ESI←ESI-1/2/4, EDI←EDI-1/2/4	-
LODSB LODSW LODSD	acc←mem[ESI] Se DF=0 então ESI←ESI+1/2/4 Se DF=1 então ESI←ESI-1/2/4	-
STOSB STOSW STOSD	mem[EDI]←acc Se DF=0 então EDI←EDI+1/2/4 Se DF=1 então EDI←EDI-1/2/4	-
CMPSB CMPSW CMPSD	mem[EDI]-mem[ESI] Se DF=0 então ESI←ESI+1/2/4, EDI←EDI+1/2/4 Se DF=1 então ESI←ESI-1/2/4, EDI←EDI-1/2/4	OF, SF, ZF, AF, CF, PF
SCASB SCASW SCASD	acc-mem[ESI] Se DF=0 então ESI←ESI+1/2/4 Se DF=1 então ESI←ESI-1/2/4	OF, SF, ZF, AF, CF, PF
CLD STD	DF←0 (ativa auto incremento) DF←1 (ativa auto decremento)	-
REP REPE/REPZ REPNE/REPNZ	Exec. inst.ant;ECX←ECX-1;Fim se ECX=0 Exec. inst.ant;ECX←ECX-1;Fim se ECX=0/ZF=0 Exec. inst.ant;ECX←ECX-1;Fim se ECX=0/ZF=1	-

Tabela B.8 - Instruções p/ execução de enlaces c/ manipulação de cadeias
Adaptada de Lewis (2002)

---

[72] Instruções usadas para a comparação de operandos não assinalados.
[73] Instruções usadas para a comparação de operandos assinalados.
[74] Instruções usadas para comparação de igualdade.
[75] Por default, a instrução LOOP usa o registrador ECX para manter o contador de iterações em modo protegido e o registrador CX em modo real.

APÊNDICE C - Conjunto Completo de Regras de Otimização

A figura C.1 mostra a definição formal das regras de otimização em notação de Wirth. Para uma definição da notação de Wirth, veja, por exemplo, (José Neto, 1987).

```
CONJUNTO_DE_REGRAS = REGRA { REGRA }.

REGRA = SEQÜÊNCIA "=\n" (SEQÜÊNCIA | "\n") ["?\n" CONDIÇÃO] " ->\n" NÚM "\n\n".

SEQÜÊNCIA = ( CHAR | NÚM | VARIÁVEL )
                    { CHAR | NÚM | VARIÁVEL } "\n".

CHAR = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
           "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
           "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
           "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" |
           "Z" | "Á" | "Á" | "É" | "É" | "Í" | "Í" | "Ó" | "Ó" | "Ú" | "Ú" |
           "À" | "À" | "Ü" | "Ü" | "Â" | "Â" | "Ê" | "Ê" | "Ô" | "Ô" |
           "Ã" | "Ã" | "Õ" | "Õ" | "Ç" | "Ç" |
           "`" | "~" | "!" | "@" | "#" | "$" | "%%" | "^" | "&" | "*" |
           "*" | "(" | ")" | "-" | "_" | "=" | "+" | "[" | "{" | "]" | "}" |
           "\" | "|" | ";" | ":" | "'" | '"' | "," | "<" | "." | ">" | "/" | "?" |
           " " | "\f" | "\r" | "\t".

NÚM = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".

VARIÁVEL = "%" NÚM NÚM.

CONDIÇÃO = (COND1 | COND2) "\n".

COND1 = FUNC1 "(" VARIÁVEL ");".

COND2 = FUNC2 "(" VARIÁVEL "," VARIÁVEL ");".

FUNC1 = ("is_const" | "is_not_const").

FUNC2 = ("is_strstr" | "is_not_strstr").
```

Figura C.1 - Definição formal das regras de otimização

Um conjunto de regras é formado ao menos por uma regra de otimização. Uma regra de otimização se divide em duas partes: uma seqüência de busca e uma seqüência de substituição. O estabelecimento de uma condição adicional para a aplicação da seqüência de substituição é opcional.

A seqüência de busca é separada da seqüência de substituição pela cadeia de terminais "=\n", sendo "\n" o delimitador de fim de linha. O término da seqüência de substituição é sinalizado pelo terminal "\n". Quando se deseja eliminar a seqüência de busca basta especificar o terminal "\n" após "=\n".

A cadeia de terminais "?\n" indica a presença de uma condição adicional para a aplicação da seqüência de substituição. O término da condição é sinalizado pelo terminal "\n". A cadeia de terminais "->\n" indica o ganho proporcionado pela

aplicação da seqüência de substituição[76]. O ganho é seguido pela cadeia de terminais "\n\n" que separam duas regras de otimização.

Uma seqüência de busca, ou de substituição, é uma expressão regular formada ao menos por um caractere, um número ou uma variável finalizada pelo terminal "\n". Uma variável é formada pela cadeia "%nn", sendo n um número de 0 a 9. Uma variável armazena uma cadeia de caracteres até encontrar um novo átomo da regra de otimização. Um caractere é qualquer símbolo diferente de um número e dos terminais "\n" (separador de uso geral) e "%" (delimitador de variáveis). Caso a regra de otimização necessite do símbolo "%", o mesmo será representado por "%%". A regra que define um caractere enumera, sem a pretensão de ser exaustiva, os símbolos mais comuns utilizados na edição de arquivos de textos nos idiomas português e inglês, bem como alguns caracteres especiais e de controle sem representação gráfica, como " " (espaço em branco), "\f" (*form feed*[77]), "\r" (*carriage return*[78]) e "\t" (*horizontal tab*[79]).

Uma condição adicional para a aplicação da seqüência de substituição ocasiona a execução de uma função que verifica se as variáveis de entrada satisfazem ou não a uma dada condição específica. Caso as variáveis de entrada satisfaçam a condição, o otimizador aplica a seqüência de substituição. Do contrário, deixa de aplicar a seqüência de substituição. Uma função pode ter uma ou duas variáveis de entrada. As funções "is_const" e "is_not_const" verificam, respectivamente, se uma variável é formada apenas por caracteres numéricos ou não. As funções "is_strstr" e "is_not_strstr" verificam, respectivamente, a ocorrência ou não de uma subcadeia em uma cadeia, sendo a cadeia representada pela primeira variável e a subcadeia representada pela segunda variável.

Uma vez definida a sintaxe, vamos apresentar as regras de otimização

---

[76] Embora o ganho proporcionado seja um valor conservador que não considera o tamanho efetivo das instruções, a função objetivo utiliza-o para estimar o ganho final auferido sem ter que gerar o código-objeto. Quanto mais o ganho proporcionado se aproximar do valor real, melhor será a estimativa final feita pela função objetivo. Contudo, para o presente trabalho, os ganhos utilizados são suficientes para mostrar o funcionamento do algoritmo de otimização *peephole* adaptativo.

[77] Alimentação de formulário

[78] Retorno do carro

[79] Tabulação horizontal

empregadas pelo algoritmo de otimização *peephole* adaptativo. Parte das regras foi adaptada de material encontrado na rede mundial de computador (Navia, 2002), parte foi desenvolvida a partir das referências encontradas na literatura (Davidson; Whalley, 1989), (Morgan, 1998) e parte foi desenvolvida a partir da análise do código em linguagem de montagem emitido pelo LCC. Para mais detalhes sobre as instruções da arquitetura x86, consultar o apêndice B.

Regra 1

```
%00
mov %01,%02
mov %02,%01
=
%00
mov %01,%02
->
1
```

Exemplo - A seqüência de instruções:

```
mov edi,dword ptr (-4)[ebp]
mov dword ptr (-8)[ebp],edi
mov edi, dword ptr (-8)[ebp]
```

por ser substituída por

```
mov edi,dword ptr (-4)[ebp]
mov dword ptr (-8)[ebp],edi
```

Comentário - Elimina movimentações cruzadas entre os operandos de origem e destino. Para diminuir o número de conflitos entre regras, a janela de otimização foi ampliada de 2 para 3 instruções através do armazenamento de uma instrução completa na variável %00. Desta forma, a regra entra em conflito apenas com a regra 32.

Regra 2

```
mov %00x,%01x
mov %02,%00l
=
mov %02,%01l
->
1
```

Exemplo - A seqüência de instruções

```
mov ebx,eax
mov byte ptr (-8)[ebp],bl
```

pode ser substituída por

```
mov byte ptr (-8)[ebp],al
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O LCC gera as seqüências de instruções acima durante a movimentação de bytes. No exemplo acima, o valor do registrador BL não deve ser usado pelas instruções seguintes. Na verdade, o LCC nunca faz uso do seu conteúdo. Portanto, a aplicação da regra é segura.

Regra 3

```
mov e%00,%01
mov e%02x,e%00
mov %03,%02l
=
mov e%02x,%01
mov %03,%02l
->
1
```

Exemplo - A seqüência de instruções

```
mov esi,dword ptr (-16)[ebp]
mov eax,esi
mov byte ptr (-1)[ebp],al
```

pode ser substituída por

```
mov eax,dword ptr (-16)[ebp]
mov byte ptr (-1)[ebp],al
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 2 também se aplica a esta regra.

Regra 4

```
mov e%00,%01
mov e%02x,e%00
movzx e%03x,%02l
mov %04,%03l
=
mov e%02x,%01
mov %04,%02l
->
2
```

Exemplo - A seqüência de instruções

```
mov esi,dword ptr (-16)[ebp]
mov ebx,esi
movzx eax,bl
mov byte ptr (-1)[ebp],al
```

pode ser substituída por

```
mov ebx,dword ptr (-16)[ebp]
mov byte ptr (-1)[ebp],bl
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 2 também se aplica a esta regra.

Regra 5

```
mov e%00,0
%01
%02
=
xor e%00,e%00
%01
%02
->
0
```

Exemplo - A seqüência de instruções

```
mov eax,0
L439:
mov esp,ebp
```

pode ser substituída por

```
xor eax,eax
L439:
mov esp,ebp
```

Comentário - Elimina o armazenamento direto de 0 em registradores por uma operação que reposiciona o valor do registrador. Embora não haja ganho de instruções, esta regra substitui uma instrução mais longa (5 bytes) por uma instrução mais curta (2 bytes).

Regra 6

```
%00
mov e%01,0
add e%01,%02
=
%00
mov e%01,%02
->
1
```

Exemplo - A seqüência de instruções

```
L250:
mov eax,0
add eax,5
```

pode ser substituída por

```
L250:
mov eax,5
```

Comentário - Elimina a soma de uma constante em um registrador cujo valor é nulo pelo armazenamento direto do valor no registrador.

Regra 7

```
add %00,0
%02
=
%02
->
1
```

Exemplo - A seqüência de instruções

```
add esp,0
mov dword ptr (-12)[ebp],eax
```

pode ser substituída por

```
mov dword ptr (-12)[ebp],eax
```

Comentário - Elimina soma desnecessária de 0 ao valor de um registrador.

Regra 8

```
add e%00,1
%02
=
inc e%00
%02
->
0
```

Exemplo - A seqüência de instruções

```
add edi,1
cmp dword ptr (-260)[ebp],edi
```

pode ser substituída por

```
inc edi
cmp dword ptr (-260)[ebp],edi
```

Comentário - Substitui a soma de 1 ao valor de um registrador por um incremento do valor do registrador. Embora não haja ganho de instruções, esta regra substitui uma instrução mais longa (3 bytes) por uma instrução mais curta (1 byte).

Regra 9

```
sub %00,0
%02
=
%02
->
1
```

Exemplo - A seqüência de instruções

```
sub ecx,0
cmp ecx,5
```

pode ser substituída por

```
cmp ecx,5
```

Comentário - Elimina subtração desnecessária de 0 ao valor de um registrador.

Regra 10

```
sub e%00,1
%02
=
dec e%00
%02
->
0
```

Exemplo - A seqüência de instruções

```
sub edi,1
cmp dword ptr (-260)[ebp],edi
```

pode ser substituída por

```
dec edi
cmp dword ptr (-260)[ebp],edi
```

Comentário - Substitui a subtração de 1 ao valor de um registrador por um decremento do valor do registrador. Embora não haja ganho de instruções, esta regra substitui uma instrução mais longa (3 bytes) por uma instrução mais curta (1 byte).

Regra 11

```
%00mul e%01,e%01,0
%02
=
mov e%01,0
%02
->
0
```

Exemplo - A seqüência de instruções

```
imul eax,eax,0
mov dword ptr (-1)[ebp],eax
```

pode ser substituída por

```
mov eax,0
mov dword ptr (-1)[ebp],eax
```

Comentário - Substitui a multiplicação do valor de um registrador por 0 pelo armazenamento direto do valor 0 no registrador. Neste caso, a aplicação desta regra propicia a aplicação posterior da regra 5.

Regra 12

```
%00mul e%01,e%01,1
%02
=
%02
->
1
```

Exemplo - A seqüência de instruções

```
imul eax,eax,1
mov dword ptr (-1)[ebp],eax
```

pode ser substituída por

```
mov dword ptr (-1)[ebp],eax
```

Comentário - Elimina a multiplicação do valor de um registrador por 1.

Regra 13

```
%00div e%01,e%01,1
%02
=
%02
->
1
```

Exemplo - A seqüência de instruções

```
idiv eax,eax,1
mov dword ptr (-1)[ebp],eax
```

pode ser substituída por

```
mov dword ptr (-1)[ebp],eax
```

Comentário - Elimina a divisão do valor de um registrador por 1.

Regra 14

```
mov e%00,%01
mov e%02,e%00
inc e%00
mov %01,e%00
=
mov e%02,1
xadd %01,e%02
->
2
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-1)[ebp]
mov esi,edi
inc edi
mov dword ptr (-1)[ebp],edi
```

pode ser substituída por

```
mov esi,1
xadd esi,dword ptr (-1)[ebp]
```

Comentário - Substitui uma seqüência de instruções por uma seqüência em que o valor anterior de uma posição de memória é armazenado em ESI e o valor da posição de memória é incrementado pelo valor de ESI. Isto permite usar o valor anterior de uma posição de memória em ESI para fazer uma atribuição de valor. O valor em EDI não deve ser utilizado posteriormente, o que nunca ocorre com o código gerado pelo LCC. Portanto, a aplicação da regra é segura.

Regra 15

```
mov e%00,%01
mov e%02,e%00
add e%00,%03
mov %01,e%00
=
mov e%02,%03
xadd %01,e%02
->
2
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-1)[ebp]
mov esi,edi
add edi,2
mov dword ptr (-1)[ebp],edi
```

pode ser substituída por

```
mov esi,2
xadd esi,dword ptr (-1)[ebp]
```

Comentário - O mesmo caso da regra anterior, sendo o incremento substituído por uma soma.

Regra 16

```
xor e%00,e%00
mov %01,e%00
xor e%00,e%00
mov %02,e%00
=
xor e%00,e%00
mov %01,e%00
mov %02,e%00
->
1
```

Exemplo - A seqüência de instruções

```
xor eax,eax
mov dword ptr (-4)[ebp],eax
xor eax,eax
mov dword ptr (-8)[ebp],eax
```

pode ser substituída por

```
xor eax,eax
mov dword ptr (-4)[ebp],eax
mov dword ptr (-8)[ebp],eax
```

Comentário - O valor do registrador EAX só precisa ser inicializado uma única vez para armazenar o mesmo valor nas posições indicadas da memória.

Regra 17

```
mov e%00,e%01
%03 e%00,%02[e%00]%04
=
%03 e%00,%02[e%01]%04
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,esi
mov edi,dword ptr[edi][ebp]
```

pode ser substituída por

```
mov edi,dword ptr[esi][ebp]
```

Comentário - Elimina uma operação de armazenamento desnecessária durante uma operação de armazenamento com endereçamento indireto.

Regra 18

```
lea e%00,%01
mov e%02x,e%00
movzx e%00,%02l
=
lea e%02x,%01
movzx e%00,%02l
->
1
```

Exemplo - A seqüência de instruções

```
lea esi,[edi*2]
mov ebx,esi
movzx esi,bl
```

pode ser substituída por

```
lea ebx,[edi*2]
movzx esi,bl
```

Comentário - Elimina a utilização desnecessária de registradores intermediários.

Regra 19

```
lea e%00,%01
mov e%02x,e%00
mov %03,%02l
=
lea e%02x,%01
mov %03,%02l
->
1
```

Exemplo - A seqüência de instruções

```
lea esi,[edi*2]
mov ebx,esi
mov byte ptr (-1)[ebp],bl
```

pode ser substituída por

```
lea ebx,[edi*2]
mov byte ptr (-1)[ebp],bl
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 2 também se aplica a esta regra.

Regra 20

```
mov %00l,%01
movzx e%02,%00l
=
movzx e%02,%01
->
1
```

Exemplo - A seqüência de instruções

```
mov bl,byte ptr (-1)[ebp]
movzx edi,bl
```

pode ser substituída por

```
movzx edi,byte ptr (-1)[ebp]
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 2 também se aplica a esta regra.

Regra 21

```
movzx e%00,%01
cmp e%00,%02
j%03 %04
%05:
mov e%00,%06
?
is_const(%02);
=
cmp %01,%02
j%03 %04
%05:
mov e%00,%06
->
1
```

Exemplo - A seqüência de instruções

```
movzx edi,byte ptr (-1)[ebp]
cmp edi,0
jne L4
L5:
mov edi,dword ptr (-2)[ebp]
```

pode ser substituída por

```
cmp byte ptr (-1)[ebp],0
jne L4
L5:
mov edi,dword ptr (-2)[ebp]
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. No exemplo acima, o valor do registrador EDI não deve ser usado como parte do operando de origem nas instruções seguintes após o desvio condicional. Na verdade, o LCC nunca faz uso do seu conteúdo. Portanto, a aplicação da regra é segura.

Regra 22

```
movzx e%00,%01
cmp e%00,%02
j%03 %04
%05:
lea e%00,%06
?
is_const(%02);
=
cmp %01,%02
j%03 %04
%05:
lea e%00,%06
->
1
```

Exemplo - A seqüência de instruções

```
movzx edi,byte ptr (-1)[ebp]
cmp edi,0
jne L4
L5:
lea edi,dword ptr (1)[ebp]
```

pode ser substituída por

```
cmp byte ptr (-1)[ebp],0
jne L4
L5:
lea edi,dword ptr (1)[ebp]
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 21 também se aplica a esta regra.

Regra 23

```
movzx e%00,%01
cmp e%00,%02
j%03 %04
jmp %05
?
is_const(%02);
=
cmp %01,%02
j%03 %04
jmp %05
->
1
```

Exemplo - A seqüência de instruções

```
movzx edi,byte ptr (-1)[ebp]
cmp edi,0
jne L4
jmp L5
```

pode ser substituída por

```
cmp byte ptr (-1)[ebp],0
jne L4
jmp L5
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 21 também se aplica a esta regra.

Regra 24

```
mov e%00,%01
cmp e%00,%02
j%03 %04
%05:
mov e%00,%06
?
is_const(%02);
=
cmp %01,%02
j%03 %04
%05:
mov e%00,%06
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-1)[ebp]
cmp edi,0
jne L4
L5:
mov edi,dword ptr (-5)[ebp]
```

pode ser substituída por

```
cmp dword ptr (-1)[ebp],0
jne L4
L5:
mov edi,dword ptr (-5)[ebp]
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. No exemplo acima, o valor do registrador EDI não deve ser usado como parte do operando de origem nas instruções seguintes após o desvio condicional. Na verdade, o LCC nunca faz uso do seu conteúdo. Portanto, a aplicação da regra é segura.

Regra 25

```
mov e%00,%01
cmp e%00,%02
j%03 %04
%05:
lea e%00,%06
?
is_const(%02);
=
cmp %01,%02
j%03 %04
%05:
lea e%00,%06
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-1)[ebp]
cmp edi,0
jne L4
L5:
lea edi,dword ptr (1)[ebp]
```

pode ser substituída por

```
cmp dword ptr (-1)[ebp],0
jne L4
L5:
lea edi,dword ptr (1)[ebp]
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 24 também se aplica a esta regra.

Regra 26

```
mov e%00,%01
cmp e%00,%02
j%03 %04
jmp %05
?
is_const(%02);
=
cmp %01,%02
j%03 %04
jmp %05
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-1)[ebp]
cmp edi,0
jne L4
jmp L5
```

pode ser substituída por

```
cmp dword ptr (-1)[ebp],0
jne L4
jmp L5
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 24 também se aplica a esta regra.

Regra 27

```
add e%00,%01
mov e%02,%03[e%00]
=
mov e%02,%03(%01)[e%00]
add e%00,%01
->
0
```

Exemplo - A seqüência de instruções

```
add edi,2
mov eax,dword ptr [edi]
```

pode ser substituída por

```
mov eax,dword ptr(2)[edi]
add edi,2
```

Comentário - Reordena as instruções visando expor mais contexto para a aplicação de outras regras.

Regra 28

```
mov %00,e%01
mov %02,%00
%03 %00,%04
=
mov %02,e%01
mov %00,e%01
%03 %00,%04
->
0
```

Exemplo - A seqüência de instruções

```
mov dword ptr (-1)[ebp],eax
mov edi,dword ptr (-1)[ebp]
mov dword ptr (-1)[ebp],5
```

pode ser substituída por

```
mov edi,eax
mov dword ptr (-1)[ebp],eax
mov dword ptr (-1)[ebp],5
```

Comentário - Reordena as instruções visando expor mais contexto para a aplicação de outras regras.

Regra 29

```
mov %00,%01
mov %02,%00
%03 %00,%04
?
is_const(%01);
=
mov %02,%01
mov %00,%01
%03 %00,%04
->
0
```

Exemplo - A seqüência de instruções

```
mov dword ptr (-1)[ebp],5
mov edi,dword ptr (-1)[ebp]
mov dword ptr (-1)[ebp],eax
```

pode ser substituída por

```
mov edi,5
mov dword ptr (-1)[ebp],5
mov dword ptr (-1)[ebp],eax
```

Comentário - Reordena as instruções visando expor mais contexto para a aplicação de outras regras.

Regra 30

```
mov %00,e%01
mov e%02,%00
cmp e%02,%03
=
mov %00,e%01
mov e%02,e%01
cmp e%02,%03
->
0
```

Exemplo - A seqüência de instruções

```
mov dword ptr (-1)[ebp],eax
mov edi,dword ptr (-1)[ebp]
cmp edi,0
```

pode ser substituída por

```
mov dword ptr (-1)[ebp],eax
mov edi,eax
cmp edi,0
```

Comentário - Efetua uma simplificação visando expor mais contexto para a aplicação de outras regras.

Regra 31

```
mov %00,%01
mov e%02,%00
cmp e%02,%03
?
is_const(%01);
=
mov %00,%01
mov e%02,%01
cmp e%02,%03
->
0
```

Exemplo - A seqüência de instruções

```
mov dword ptr (-1)[ebp],5
mov edi,dword ptr (-1)[ebp]
cmp edi,ebx
```

pode ser substituída por

```
mov dword ptr (-1)[ebp],5
mov edi,5
cmp edi,ebx
```

Comentário - Efetua uma simplificação visando expor mais contexto para a aplicação de outras regras.

Regra 32

```
mov %00,%01
mov %00,%02
?
is_not_strstr(%02,%00);
=
mov %00,%02
->
1
```

Exemplo - A seqüência de instruções

```
mov dword ptr (-1)[ebp],eax
mov dword ptr (-1)[ebp],5
```

pode ser substituída por

```
mov dword ptr (-1)[ebp],5
```

Comentário - Elimina uma instrução desnecessária.

Regra 33

```
mov %00,%01
lea %00,%02
?
is_not_strstr(%02,%00);
=
lea %00,%02
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,eax
lea edi,dword ptr (1)[ebp]
```

pode ser substituída por

```
lea edi,dword ptr (1)[ebp]
```

Comentário - Elimina uma instrução desnecessária.

Regra 34

```
mov %00,%00
%01
=
%01
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,edi
cmp edi,0
```

pode ser substituída por

```
cmp edi,0
```

Comentário - Elimina uma instrução desnecessária.

Regra 35

```
mov e%00,e%01
mov %02,e%00
jmp %03
=
mov %02,e%01
jmp %03
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,eax
mov dword ptr (-1)[ebp],edi
jmp L97
```

pode ser substituída por

```
mov dword ptr (-1)[ebp],eax
jmp L97
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. No exemplo acima, o valor do registrador EDI não deve ser usado como parte do operando de origem nas instruções seguintes após o desvio incondicional. Na verdade, o LCC nunca faz uso do seu conteúdo. Portanto, a aplicação da regra é segura.

Regra 36

```
mov e%00,%01
mov %02,e%00
jmp %03
?
is_const(%01);
=
mov %02,%01
jmp %03
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,5
mov dword ptr (-1)[ebp],edi
jmp L97
```

pode ser substituída por

```
mov dword ptr (-1)[ebp],5
jmp L97
```

Comentário - Elimina a utilização desnecessária de registradores intermediários. O comentário da regra 35 também se aplica a esta regra.

Regra 37

```
mov %00,%01
push %00
mov %00,%02
?
```

```
is_not_strstr(%02,%00);
=
push %01
mov %00,%02
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (8)[edi]
push edi
mov edi,dword ptr (-4)[ebp]
```

pode ser substituída por

```
push dword ptr (8)[edi]
mov edi,dword ptr (-4)[ebp]
```

Comentário - Elimina a utilização desnecessária de registradores intermediários durante a preparação de uma chamada de procedimento.

Regra 38

```
mov %00,%01
push %00
call %02
add esp,%03
mov %00,%04
?
is_not_strstr(%04,%00);
=
push %01
call %02
add esp,%03
mov %00,%04
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-4)[ebp]
push edi
call _connect
add esp,8
mov edi,dword ptr (-4)[ebp]
```

pode ser substituída por

```
push dword ptr (-4)[ebp]
call _connect
add esp,8
mov edi,dword ptr (-4)[ebp]
```

Comentário - O comentário da regra 37 se aplica a esta regra.

Regra 39

```
mov %00,%01
push %00
call %02
add esp,%03
%04:
mov %00,%05
?
is_not_strstr(%05,%00);
=
push %01
call %02
add esp,%03
%04:
mov %00,%05
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-4)[ebp]
push edi
call _connect
add esp,8
L5:
mov edi,dword ptr (-4)[ebp]
```

pode ser substituída por

```
push dword ptr (-4)[ebp]
call _connect
add esp,8
L5:
mov edi,dword ptr (-4)[ebp]
```

Comentário - O comentário da regra 37 se aplica a esta regra.

Regra 40

```
mov %00,%01
push %00
lea %00,%02
?
is_not_strstr(%02,%00);
=
push %01
lea %00,%02
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (8)[edi]
push edi
lea edi,(L51)
```

pode ser substituída por

```
push dword ptr (8)[edi]
lea edi,(L51)
```

Comentário - O comentário da regra 37 se aplica a esta regra.

Regra 41

```
mov %00,%01
push %00
call %02
add esp,%03
lea %00,%04
```

```
?
is_not_strstr(%04,%00);
=
push %01
call %02
add esp,%03
lea %00,%04
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-4)[ebp]
push edi
call _connect
add esp,8
lea edi,(L51)
```

pode ser substituída por

```
push dword ptr (-4)[ebp]
call _connect
add esp,8
lea edi,(L51)
```

Comentário - O comentário da regra 37 se aplica a esta regra.

Regra 42

```
mov %00,%01
push %00
call %02
add esp,%03
%04:
lea %00,%05
?
is_not_strstr(%05,%00);
=
push %01
call %02
add esp,%03
%04:
lea %00,%05
->
1
```

Exemplo - A seqüência de instruções

```
mov edi,dword ptr (-4)[ebp]
push edi
call _connect
add esp,8
L5:
lea edi,(L51)
```

pode ser substituída por

```
push dword ptr (-4)[ebp]
call _connect
add esp,8
L5:
lea edi,(L51)
```

Comentário - O comentário da regra 37 se aplica a esta regra.

Regra 43

```
mov ecx,2
rep movsb
=
mov ecx,1
rep movsw
->
0
```

Comentário - Substitui uma seqüência de instruções mais lentas por uma seqüência mais rápida. Quando um registro é atribuído a outro, o LCC gera uma seqüência de instruções para a movimentação de bytes utilizando a instrução MOVSB. Caso o número de bytes a ser movimentado seja um múltiplo de 4, pode-se usar em seu lugar a instrução MOVSW. Caso o número de bytes a ser movimentado não seja um múltiplo de 4, mas seja um múltiplo de 2, pode-se usar a instrução MOVSD.

Regra 44

```
mov ecx,4
rep movsb
=
mov ecx,1
rep movsd
->
0
```

Comentário - Veja comentário da regra 43.

Regra 45

```
mov ecx,6
rep movsb
=
mov ecx,3
rep movsw
->
0
```

Comentário - Veja comentário da regra 43.

Regra 46

```
mov ecx,8
rep movsb
=
mov ecx,2
rep movsd
->
0
```

Comentário - Veja comentário da regra 43.

Regra 47

```
mov ecx,10
rep movsb
```

=
mov ecx,5
rep movsw
->
0

Comentário - Veja comentário da regra 43.

Regra 48

mov ecx,12
rep movsb
=
mov ecx,3
rep movsd
->
0

Comentário - Veja comentário da regra 43.

Regra 49

mov ecx,14
rep movsb
=
mov ecx,7
rep movsw
->
0

Comentário - Veja comentário da regra 43.

Regra 50

mov ecx,16
rep movsb
=
mov ecx,4
rep movsd
->
0

Comentário - Veja comentário da regra 43.

Regra 51

mov ecx,18
rep movsb
=
mov ecx,9
rep movsw
->
0

Comentário

Veja comentário da regra 43.

Regra 52

mov ecx,20
rep movsb
=
mov ecx,5
rep movsd
->
0

Comentário - Veja comentário da regra 43.

Regra 53

mov ecx,22
rep movsb
=
mov ecx,11
rep movsw
->
0

Comentário - Veja comentário da regra 43.

Regra 54

mov ecx,24
rep movsb
=
mov ecx, 6
rep movsd
->
0

Comentário - Veja comentário da regra 43.

Regra 55

mov ecx,26
rep movsb
=
mov ecx,13
rep movsw
->
0

Comentário - Veja comentário da regra 43.

Regra 56

mov ecx,28
rep movsb
=
mov ecx,7
rep movsd
->
0

Comentário - Veja comentário da regra 43.

Regra 57

mov ecx,30
rep movsb
=
mov ecx,15
rep movsw
->
0

Comentário - Veja comentário da regra 43.

APÊNDICE D - Listagem Completa do Otimizador *Peephole* Adaptativo

```
/* Includes */
#include <ctype.h>
#include <stdio.h>

/* Constantes */
#define COMMENT                 ";"
#define FALSE                   0
#define TRUE                    1
#define TAM_TABH                107
#define MAXLINHA                256
#define NMAX_OTM_ATIVOS         8
#define NUM_VARS                64

/* Tipos de dados */
typedef struct TipoInstrucao
{
    char                *instrucao;
    struct TipoInstrucao *anterior_p;
    struct TipoInstrucao *proxima_p;
} TIPO_INSTRUCAO;

typedef struct TipoOtimizador
{
    TIPO_INSTRUCAO    inicio;
    TIPO_INSTRUCAO    fim;
    TIPO_INSTRUCAO   *atual;
    int               cont_hab;
    unsigned int      cont_inst;
    unsigned int      cont_fila;
    unsigned int      num_subs;
} TIPO_OTIMIZADOR;

typedef struct TipoIdent
{
    TIPO_INSTRUCAO   *ident_antes;
    TIPO_INSTRUCAO   *ident_depois;
    int              (*p2FuncCond)();
    char              *p2Param;
    int               compl_cond;
    unsigned int      ident_ganho;
    unsigned int      ident_num;
    struct TipoIdent *ident_p;
} TIPO_IDENT;

struct lnode
{
    void *l_elem;
    struct lnode *l_prev, *l_next;
};

/* Prototipos */
void stoupper(char *);
void error(char *);
void connect(struct lnode *, struct lnode *);
void cstack(void);
void dstack(void);
void fstack(void);
void push(void *);
void *pop(void);

void conecta(TIPO_INSTRUCAO *, TIPO_INSTRUCAO *);
char *copia_txt(char *);
void insere(char *, TIPO_INSTRUCAO *);
int monta_lista(FILE *, char *, char *,
                TIPO_INSTRUCAO *, TIPO_INSTRUCAO *);
int IsConst(char *, char **);
char *IsStrchr(char *, int);
int IsStrstr(char *, char **);
void monta_ident(FILE *);
int compara(char *, char *, char **);
unsigned int det_subst(TIPO_INSTRUCAO *);
char *subst(char *, char **);
void troca(TIPO_INSTRUCAO *, TIPO_INSTRUCAO *,
           TIPO_INSTRUCAO *, char **);
unsigned int substitui(TIPO_INSTRUCAO *, unsigned int);

/* Variaveis Comuns */
FILE            *p2o = stdout;
FILE            *p2e = stderr;
int              debug = FALSE;
int              noadp = FALSE;
int              stx86 = FALSE;
TIPO_IDENT      *opts = 0;
TIPO_INSTRUCAO   primeira;
TIPO_INSTRUCAO   ultima;
struct lnode    *tos = NULL;
char             comment[] = COMMENT;

/* Procedimentos */

/* stoupper - transforma letras minusculas em */
/* maiusculas */
void stoupper(s) char *s;
{
  int i;

  if (s != 0)
  {
    for (i = 0; i < strlen(s); i++)
      if ((s[i] >= 'a') && (s[i] <= 'z')) s[i] -= 0x20;
  }
}

/* error - reporta erro and termina */
void error(s) char *s;
{
  fputs(s, p2e);
  exit(1);
}

/* connect - conecta p1 a p2 */
void connect(p1, p2) struct lnode *p1, *p2;
{
  if (p1 == NULL || p2 == NULL)
    error("connect: nao pode conectar ponteiro
           nulo\n");
  p1->l_next = p2;
  p2->l_prev = p1;
}

/* cstack - cria pilha */
void cstack(void)
{
  if (tos != NULL)
    error("cstack: pilha criada anteriormente\n");

  tos = (struct lnode *) malloc(sizeof *tos);
  if (tos == NULL)
    error("cstack: falha de alocacao de memoria\n");

  tos->l_elem = NULL;
  tos->l_next = NULL;
  tos->l_prev = NULL;
}

/* dstack - destroi pilha */
void dstack(void)
{
  fstack();
  free(tos);
  tos = NULL;
}

/* fstack - libera pilha */
void fstack(void)
{
  TIPO_INSTRUCAO *prox, *ti;
  TIPO_OTIMIZADOR *temp;

  while ((tos->l_next) != NULL)
  {
    temp = (TIPO_OTIMIZADOR *)pop();
    prox = temp->inicio.proxima_p;
    while(prox != &temp->fim)
    {
      ti = prox;
      prox = ti->proxima_p;
      free(ti);
    }
    free(temp);
  }
}

/* push - insere um elemento no topo da pilha */
void push(el) void *el;
{
  struct lnode *n;

  if (tos == NULL)
    error("push: pilha nao criada\n");

  n = (struct lnode *) malloc(sizeof *n);
  if (n == NULL)
    error("push: falha de alocacao de memoria\n");

  n->l_elem = el;
  if ((tos->l_next) == NULL)
  {
    n->l_next = NULL;
    n->l_prev = tos;
    tos->l_next = n;
  }
  else
  {
    connect(n, tos->l_next);
    connect(tos, n);
  }
}
```

```
/* pop - retira um elemento do topo da pilha */
void *pop(void)
{
  struct lnode *temp;
  void *el;

  temp = tos->l_next;
  if (temp == NULL)
    return(NULL);

  el = temp->l_elem;

  tos->l_next = temp->l_next;
  if ((tos->l_next) != NULL)
      (tos->l_next)->l_prev = tos;
  free(temp);
  return(el);
}

/* conecta - conecta dois nos de uma lista */
/* duplamente ligada */
void conecta(p1, p2) TIPO_INSTRUCAO *p1, *p2;
{
  if ((p1 == NULL) || (p2 == NULL))
    error("conecta: nao pode conectar ponteiro
           nulo\n");

  p1->proxima_p  = p2;
  p2->anterior_p = p1;
}

/* copia_txt - copia uma cadeia de caracteres em */
/* uma tabela de hash */
char *copia_txt(l) char *l;
{
  register struct TabhElem *p;
  register char *p1, *p2, *s;
  register int   i;
  static struct TabhElem
  {
    char *tabh_elem;
    struct TabhElem *tabh_p;
  } *tabh[TAM_TABH] = { 0 };

  /* calcula funcao de hash */

  s = l;
  for (i = 0; *s; i += *s++)
    ;
  i %= TAM_TABH;

  /* procura elemento na tabela: */
  /* se existe, retorna apontador do elemento */
  /* senao, insere elemento na tabela */

  for (p = tabh[i]; p; p = p->tabh_p)
  {
    for (p1=l, p2=p->tabh_elem;
         (*p1 != '\0') && (*p2 != '\0');
         p1++, p2++)
      if (*p1 != *p2) break;

    if ((*p1 == '\0') && (*p2 == '\0'))
      return(p->tabh_elem);
  }

  p = (struct TabhElem *)malloc(sizeof *p);
  if (p == NULL)
    error("copia_txt: falha de alocacao de memoria
           (1)\n");

  p->tabh_elem = (char *)malloc((s-l)+1);
  if (p->tabh_elem == NULL)
    error("copia_txt: falha de alocacao de memoria
           (2)\n");

  strcpy(p->tabh_elem, l);
  p->tabh_p = tabh[i];
  tabh[i] = p;
  return(p->tabh_elem);
}

/* insere - insere uma instrucao antes da instrucao */
/* atual */
void insere(l, p) char *l; TIPO_INSTRUCAO *p;
{
  TIPO_INSTRUCAO *n;

  n = (TIPO_INSTRUCAO *)malloc(sizeof *n);
  if (n == NULL)
    error("insere: falha de alocacao de memoria\n");

  n->instrucao = copia_txt(l);
  conecta(p->anterior_p, n);
  conecta(n, p);
}
```

```
/* monta_lista - insere as linhas do arquivo em uma */
/* lista duplamente ligada */
int monta_lista(fp, fim1, fim2, p1, p2)
  FILE *fp; char *fim1, *fim2; TIPO_INSTRUCAO *p1, *p2;
{
  char linha[MAXLINHA];

  conecta(p1, p2);
  if ((fim1 != 0) && (fim2 != 0))
  {
    while (fgets(linha, MAXLINHA, fp) != NULL &&
           strcmp(linha, fim1) && strcmp(linha, fim2))
      insere(linha, p2);
    if (strcmp(linha, fim2) == 0)
      return(TRUE);
    else
      return(FALSE);
  }
  else if ((fim1 != 0) && (fim2 == 0))
  {
    while (fgets(linha, MAXLINHA, fp) != NULL &&
           strcmp(linha, fim1))
      insere(linha, p2);
    return(FALSE);
  }
  else if ((fim1 == 0) && (fim2 != 0))
  {
    while (fgets(linha, MAXLINHA, fp) != NULL &&
           strcmp(linha, fim2))
      insere(linha, p2);
    if (strcmp(linha, fim2) == 0)
      return(TRUE);
    else
      return(FALSE);
  }
  else
  {
    error("monta_lista: um dos terminadores deve ser
           nao nulo\n");
    return(FALSE);
  }
}

/* IsConst - determina se a cadeia de entrada eh */
/* formada por numeros */
int IsConst(par, vars) char *par, **vars;
{
  char *p;

  p = vars[(10*(par[1]-'0')) + (par[2]-'0')];

  if (p != 0)
  {
    for (; (*p != 0); p++)
      if (! isdigit(*p)) { return(FALSE); }
    return(TRUE);
  }
  else
  {
    return(FALSE);
  }
}

/* IsStrchr - encontra a primeira ocorrencia de um */
/* caractere em uma cadeia */
char *IsStrchr(s, c) char *s; int c;
{
  char ch = c;

  for (; *s != ch; ++s)
    if (*s == '\0') { return(0); }

  return((char *)s);
}

/* IsStrstr - determina a ocorrencia de uma subcadeia*/
/* em uma cadeia */
int IsStrstr(par, vars) char *par, **vars;
{
  char *s1, *s2, *sc1, *sc2;

  s1 = vars[(10*(par[1]-'0')) + (par[2]-'0')];
  s2 = vars[(10*(par[5]-'0')) + (par[6]-'0')];

  if (s2 == 0)
    return(TRUE);

  if (IsConst(par, vars))
    return(FALSE);

  for (; (s1 = IsStrchr(s1, *s2)) != 0; ++s1)
  {
    for (sc1 = s1, sc2 = s2; ; )
      {
        if (*++sc2 == '\0')
          return(TRUE);
        else if (*++sc1 != *sc2)
          break;
```

```
    }
  }

  return(FALSE);
}

/* monta_ident - monta lista de identidades de */
/* otimizacao */
void monta_ident(fp) FILE *fp;
{
  char *eolf, *p2str, *p2sub;
  char linha[MAXLINHA];
  char sFunc[MAXLINHA];
  char sPar[MAXLINHA];
  int i, resp, retc;
  unsigned int temp, cont_ident;
  TIPO_INSTRUCAO inic, term;
  TIPO_IDENT *p, **prox;

  cont_ident = 1;

  prox = &opts;
  while (*prox)
     prox = &((*prox)->ident_p);

  while(!feof(fp))
  {
    p = (TIPO_IDENT *)malloc(sizeof *p);
    if (p == NULL)
      error("monta_ident: falha de alocacao de
             memoria\n");

    resp = monta_lista(fp, "=\n", "?\n", &inic, &term);
    inic.proxima_p->anterior_p = 0;
    if (term.anterior_p)
      term.anterior_p->proxima_p = 0;
    p->ident_antes = term.anterior_p;

    if (resp)
    {
     if (fgets(linha, MAXLINHA, fp) != NULL)
     {
       sFunc[0] = sPar[0] = 0;

       for (p2str = &linha[0], p2sub = &sFunc[0];
            (*p2str != '(') && (*p2str != 0);
            p2str++, p2sub++)
         *p2sub = *p2str;
       if (*p2str == '(')
       {
         *p2sub = 0;
         p2str++;

         for (p2sub = &sPar[0];
              (*p2str != ')') && (*p2str != 0);
              p2str++, p2sub++)
           *p2sub = *p2str;
         if (*p2str == ')')
         {
           *p2sub = 0;
           p2str++;

           if (strcmp(p2str, ";\n") == 0)
           {
             stoupper(sFunc);

             if ((strcmp(sFunc, "IS_CONST") == 0) ||
                 (strcmp(sFunc, "IS_NOT_CONST") == 0))
             {
               if (strlen(sPar) == 3 &&
                   sPar[0] == '%' &&
                   isdigit(sPar[1]) &&
                   isdigit(sPar[2]))
               {
                 p->p2FuncCond = IsConst;
                 p->p2Param = copia_txt(sPar);
                 if (strcmp(sFunc, "IS_CONST") == 0)
                   p->compl_cond = FALSE;
                 else
                   p->compl_cond = TRUE;
               }
               else
               {
                 error("monta_ident: parametro
                        invalido(1)\n");
               }
             }
             else if
               ((strcmp(sFunc,"IS_STRSTR") == 0) ||
                (strcmp(sFunc,"IS_NOT_STRSTR") == 0))
             {
               if (strlen(sPar) == 7 &&
                   sPar[0] == '%' &&
                   isdigit(sPar[1]) &&
                   isdigit(sPar[2]) &&
                   sPar[3] == ',' &&
                   sPar[4] == '%' &&
                   isdigit(sPar[5]) &&
                   isdigit(sPar[6]))
               {
                 p->p2FuncCond = IsStrstr;
                 p->p2Param = copia_txt(sPar);
                 if (strcmp(sFunc, "IS_STRSTR") == 0)
                   p->compl_cond = FALSE;
                 else
                   p->compl_cond = TRUE;
               }
               else
               {
                 error("monta_ident: falta de
                        parametros ou parametro
                        invalido (1)\n");
               }
             }
             else
             {
               error("monta_ident: condicao
                      desconhecida\n");
             }
           }
           else
           {
             error("monta_ident: falta ';' na
                    condicao\n");
           }
         }
         else
         {
           error("monta_ident: falta ')' na
                  condicao\n");
         }
       }
       else
       {
         error("monta_ident: falta '(' na
                condicao\n");
       }
     }

     if (fgets(linha, MAXLINHA, fp) != NULL)
     {
       if (strcmp(linha, "=\n"))
         error("monta_ident: falta '=' apos
                condicao\n");
     }
    }
    else
    {
      p->p2FuncCond = 0;
      p->p2Param = 0;
      p->compl_cond = FALSE;
    }

    monta_lista(fp, "->\n", 0, &inic, &term);
    term.anterior_p->proxima_p = 0;
    if (inic.proxima_p)
      inic.proxima_p->anterior_p = 0;
    p->ident_depois = inic.proxima_p;

    if (fgets(linha, MAXLINHA, fp) != NULL)
    {
      for (i = 0; i < (strlen(linha)-1); i++)
        if (!isdigit(linha[i]))
          error("monta_ident: digito nao
                 encontrado\n");

      linha[strlen(linha)-1] = 0;
      temp = (unsigned int)atoi(linha);
      p->ident_ganho = temp;
    }

    eolf = fgets(linha, MAXLINHA, fp);
    if (((eolf != NULL) &&
         (strcmp(linha, "\n") == 0)) ||
        (eolf == NULL))
    {
      p->ident_num = cont_ident++;
      *prox = p;
      prox = &p->ident_p;
    }
    else
      error("monta_ident: separador de regras nao
             encontrado\n");
  }

  *prox = 0;
}
```

```
/* compara - compara txt e ident e posiciona  */
/* indicadores */
int compara(txt, ident, vars)char *txt,*ident,**vars;
{
  char *p, linha[MAXLINHA];

  while (*txt && *ident)
    if (ident[0] == '%' &&
        isdigit(ident[1]) &&
        isdigit(ident[2]))
    {
      for (p = linha; *txt && *txt != ident[3];)
        *p++ = *txt++;
      *p = 0;

      p = copia_txt(linha);
      if (vars[(10*(ident[1]-'0')) +
          (ident[2]-'0')] == 0)
        vars[(10*(ident[1]-'0')) + (ident[2]-'0')] = p;
      else if (vars[(10*(ident[1]-'0')) +
               (ident[2]-'0')] != p)
        return 0;

      ident += 3;
    }
    else if (*ident++ != *txt++)
      return 0;

  return (*ident==*txt);
}

/* det_subst - determina o numero de substituicoes */
unsigned int det_subst(r) TIPO_INSTRUCAO *r;
{
  char *vars[NUM_VARS];
  int   i, res;
  TIPO_INSTRUCAO *c, *p;
  TIPO_IDENT *o;
  unsigned int num_subs = 0;
  unsigned int num_ident[10];

  for (i = 0; i < 10; i++)
    num_ident[i] = 0;
  for (o = opts; o; o = o->ident_p)
  {
    c = r;
    p = o->ident_antes;
    for (i = 0; i < NUM_VARS; i++)
      vars[i] = 0;
    while (p && c &&
           ((res = compara(c->instrucao,
                           p->instrucao, vars)) != 0))
    {
      p = p->anterior_p;
      c = c->anterior_p;
    }
    if ((p == 0) && res)
    {
      if (o->p2FuncCond == 0)
      {
        num_subs++;
        num_ident[num_subs-1] = o->ident_num;
      }
      else
      {
        if (! o->compl_cond)
        {
          if ((*(o->p2FuncCond))
              (o->p2Param, vars) == TRUE)
          {
            num_subs++;
            num_ident[num_subs-1] = o->ident_num;
          }
        }
        else
        {
          if ((*(o->p2FuncCond))
              (o->p2Param, vars) == FALSE)
          {
            num_subs++;
            num_ident[num_subs-1] = o->ident_num;
          }
        }
      }
    }
  }

  if (num_subs > 1)
  {
    for (i = 0; i < num_subs; i++)
      fprintf(p2e, "Regra: [%d] ", num_ident[i]);
    fprintf(p2e, "\n");
  }

  if (noadp && (num_subs > 1))
    num_subs = 1;
  return (num_subs);
}
```

```
/* subst - devolve o resultado da substituicao de */
/* vars em pat */
char *subst(pat, vars) char *pat, **vars;
{
  char linha[MAXLINHA], *s;
  int i;

  i = 0;
  for (;;)
    if (pat[0] == '%' &&
        isdigit(pat[1]) &&
        isdigit(pat[2]))
    {
      for (s = vars[(10*(pat[1]-'0')) + (pat[2]-'0')];
           i < MAXLINHA && ((linha[i] = *s++) != 0);
           i++)
        ;
      pat += 3;
    }
    else if (i >= MAXLINHA)
      error("subst: linha muito comprida\n");
    else if ((linha[i++] = *pat++) == 0)
      return (copia_txt(linha));
}

/* troca - aplica vars na seq. de substituicao e */
/* troca linhas entre p1 e p2 */
void troca(p1, p2, nova, vars)
  TIPO_INSTRUCAO *p1, *p2, *nova; char **vars;
{
  char *args[NUM_VARS];
  int   i;
  TIPO_INSTRUCAO *p, *psav;

  for (p = p1->proxima_p; p != p2; p = psav)
  {
    psav = p->proxima_p;
    if (debug)
      fprintf(p2o, "%s %s", comment, p->instrucao);
    free(p);
  }

  conecta(p1, p2);
  if (debug)
    fprintf(p2o, "%s =\n", comment);

  for (; nova; nova = nova->proxima_p)
  {
    for (i = 0; i < NUM_VARS; i++)
      args[i] = 0;
    insere(subst(nova->instrucao, vars), p2);
    if (debug)
      fprintf(p2o, "%s %s", comment,
              p2->anterior_p->instrucao);
  }

  if (debug)
    fprintf(p2o, "%s\n", comment);
}

/* substitui - executa uma sequencia de otimizacao */
unsigned int substitui(r, ns)
  TIPO_INSTRUCAO *r; unsigned int ns;
{
  char *vars[NUM_VARS];
  int   i, res;
  TIPO_INSTRUCAO *c, *p;
  TIPO_IDENT *o;
  unsigned int num_subs = 0;

  for (o = opts; o; o = o->ident_p)
  {
    c = r;
    p = o->ident_antes;
    for (i = 0; i < NUM_VARS; i++)
      vars[i] = 0;
    while (p && c &&
           (res = compara(c->instrucao,
                          p->instrucao, vars)) != 0)
    {
      p = p->anterior_p;
      c = c->anterior_p;
    }
    if ((p == 0) && res)
    {
      if (o->p2FuncCond == 0)
        num_subs++;
      else
      {
        if (! o->compl_cond)
        {
          if ((*(o->p2FuncCond))
              (o->p2Param, vars) == TRUE)
            num_subs++;
        }
        else
        {
```

```
        if ((*(o->p2FuncCond))
            (o->p2Param, vars) == FALSE)
          num_subs++;
      }
    }
  }

  if (num_subs == ns)
  {
    if (debug)
      fprintf(p2o, "%s identidade: %d\n", comment,
              o->ident_num);
    troca(c, r->proxima_p, o->ident_depois, vars);
    return(o->ident_ganho);
  }
 }

 error("substitui: substituicao nao encontrada\n");
 return(0);
}

/* IsInstrX86 - determina se eh uma instrucao x86 */
int IsInstrX86(txt) char *txt;
{
 char *p, linha[MAXLINHA];
 int   i;

 p = txt;

 /* linha vazia */
 if ((p == 0) || (*p == '\n'))
   return(FALSE);

 /* possivel rotulo */
 for (p = txt, i = 0; *p != '\n'; p++, i++)
   linha[i] = *p;
 i++;
 linha[i] = 0;

 if ((linha[0] == 'L') &&
     (linha[strlen(linha)-1] == ':'))
 {
   return(FALSE);
 }

 /* diretivas de montagem dentro do segmento */
 if ((strncmp(linha,"public ",7) == 0) ||
     (strncmp(linha,"align ",6) == 0) ||
     (strncmp(linha,"db ", 3) == 0) ||
     (strncmp(linha,"dw ", 3) == 0) ||
     (strncmp(linha,"dd ", 3) == 0) ||
     (strstr(linha, "label type") != 0))
 {
   return(FALSE);
 }

 return(TRUE);
}

/* Rotina principal */

int main(argc, argv) int argc; char **argv;
{
 char            *p2f;
 FILE            *fp;
 int              fim;
 int              i, j, k;
 TIPO_INSTRUCAO  *temp, *tip, *prox;
 TIPO_OTIMIZADOR *p2opt, *topt, *p2mco;
 unsigned int     subs_poss;

/* Processa argumentos de entrada */
 if ((argc > 2) && (argc < 7))
 {
   if (argc != 3)
   {
     for (i = 1; i <= (argc - 3); i++)
     {
       stoupper(argv[i]);

       if (strcmp(argv[i], "-D") == 0)
         debug = TRUE;
       else if (strcmp(argv[i], "-N") == 0)
         noadp = TRUE;
       else if (strcmp(argv[i], "-SX86") == 0)
         stx86 = TRUE;
       else if ((argv[i][0] == '-') &&
                (argv[i][1] == 'O'))
       {
         if ((fp = fopen(&argv[i][2], "w")) != NULL)
         {
           p2o = fp;
           p2e = stdout;
         }
         else
         {
           fprintf(p2e, "%s Otimizador Peephole
                   Adaptativo\n", comment);
           error("main: nao pode abrir arq. de
                 saida\n");
         }
       }
       else
       {
         fprintf(p2e, "%s Otimizador Peephole
                 Adaptativo\n", comment);
         error("main: argumento desconhecido\n");
       }
     }

     i = argc - 2;
   }
   else
   {
     i = 1;
   }

   /* Efetua leitura do arquivo de identidades */
   if ((fp = fopen(argv[i], "r")) == NULL)
   {
     fprintf(p2e, "%s Otimizador Peephole
             Adaptativo\n", comment);
     error("main: arquivo de regras nao
           encontrado\n");
   }
   else
   {
     monta_ident(fp);
     fclose(fp);
   }
   i++;

   /* Efetua leitura do arquivo fonte */
   if ((fp = fopen(argv[i], "r")) == NULL)
   {
     fprintf(p2e, "%s Otimizador Peephole
             Adaptativo\n", comment);
     error("main: arquivo fonte nao encontrado\n");
   }
   else
   {
     p2f=(char *)malloc(strlen(argv[i])*sizeof(*p2f));
     if (p2f == NULL)
     {
       fprintf(p2e, "%s Otimizador Peephole
               Adaptativo\n", comment);
       error("main: falha de alocacao de memoria\n");
     }
     strcpy(p2f, argv[i]);
     monta_lista(fp, "", 0, &primeira, &ultima);
     fclose(fp);
   }
 }
 else
 {
   fprintf(p2e, "%s Otimizador Peephole
           Adaptativo\n", comment);
   error("main: numero de incorreto de argumentos de
         entrada\n");
 }

 fprintf(p2e, "%s Otimizador Peephole Adaptativo\n",
         comment);

 primeira.instrucao = "";
 ultima.instrucao   = "";

 /* Inicia variaveis e estruturas */

 p2mco = NULL;

 p2opt = (TIPO_OTIMIZADOR *) malloc(sizeof *p2opt);
 if (p2opt == NULL)
   error("main: falha de alocacao de memoria (1)\n");

 if (stx86)
   p2opt->cont_hab  = FALSE;
 else
   p2opt->cont_hab  = TRUE;
 p2opt->cont_inst = 0;
 p2opt->cont_fila = 0;
 p2opt->atual = primeira.proxima_p;
 p2opt->num_subs  = 0;
 conecta(&p2opt->inicio, &p2opt->fim);
 p2opt->inicio.instrucao = "";
 p2opt->fim.instrucao    = "";
 i = 1;

 cstack();

 while (p2opt != NULL)
 {
   tip = p2opt->atual;
```

```
while(tip != &ultima)
{
  /* Processa todas as instrucoes */

  if (p2opt->num_subs > 0)
  {
    p2opt->cont_fila -=
      substitui(p2opt->fim.anterior_p,
                p2opt->num_subs);
  }
  else
  {
    if (stx86)
    {
      if (p2opt->cont_hab == FALSE)
      {
        if (strcmp(tip->instrucao,
            "_TEXT segment\n") == 0)
        {
          p2opt->cont_hab = TRUE;
        }
      }
      else
      {
        if (strcmp(tip->instrucao,
            "_TEXT ends\n") == 0)
        {
          p2opt->cont_hab = FALSE;
        }
      }

      if (p2opt->cont_hab &&
          IsInstrX86(tip->instrucao))
      {
        p2opt->cont_inst++;
        p2opt->cont_fila++;
      }

      insere(tip->instrucao, &p2opt->fim);
    }
    else
    {
      p2opt->cont_inst++;
      p2opt->cont_fila++;
      insere(tip->instrucao, &p2opt->fim);
    }
  }

  /* Determina o numero de substituicoes */
  subs_poss = det_subst(p2opt->fim.anterior_p);

  if (subs_poss > 0)
  {
    if (subs_poss > 1)
    {
      k = p2opt->num_subs = 1;
      for (j = 0; j < (subs_poss - 1); j++)
      {
        k++;

        topt = (TIPO_OTIMIZADOR *)
               malloc(sizeof *topt);
        if (topt == NULL)
          error("main: falha de alocacao de
                 memoria (2)\n");

        topt->cont_hab  = p2opt->cont_hab;
        topt->cont_inst = p2opt->cont_inst;
        topt->cont_fila = p2opt->cont_fila;
        topt->atual = tip;
        topt->num_subs = k;
        conecta(&topt->inicio, &topt->fim);
        topt->inicio.instrucao = "";
        topt->fim.instrucao    = "";

        temp = p2opt->inicio.proxima_p;
        while(temp != &p2opt->fim)
        {
          insere(temp->instrucao, &topt->fim);
          temp = temp->proxima_p;
        }

        push((void *)topt);
      }
    }
    else
    {
      p2opt->num_subs = 1;
    }
  }
  else
  {
    p2opt->num_subs = 0;
  }

  do
  {
    fim = TRUE;

    if (p2opt->num_subs > 0)
    {
      p2opt->cont_fila -=
        substitui(p2opt->fim.anterior_p,
                  p2opt->num_subs);
      subs_poss =
        det_subst(p2opt->fim.anterior_p);

      if (subs_poss > 0)
      {
        fim = FALSE;

        if (subs_poss > 1)
        {
          k = p2opt->num_subs = 1;
          for (j = 0; j < (subs_poss - 1); j++)
          {
            k++;

            topt = (TIPO_OTIMIZADOR *)
                   malloc(sizeof *topt);
            if (topt == NULL)
              error("main: falha de alocacao de
                     memoria (2)\n");

            topt->cont_hab  = p2opt->cont_hab;
            topt->cont_inst = p2opt->cont_inst;
            topt->cont_fila = p2opt->cont_fila;
            topt->atual = tip;
            topt->num_subs = k;
            conecta(&topt->inicio, &topt->fim);
            topt->inicio.instrucao = "";
            topt->fim.instrucao    = "";

            temp = p2opt->inicio.proxima_p;
            while(temp != &p2opt->fim)
            {
              insere(temp->instrucao,
                     &topt->fim);
              temp = temp->proxima_p;
            }

            push((void *)topt);
          }
        }
        else
        {
          p2opt->num_subs = 1;
        }
      }
      else
      {
        p2opt->num_subs = 0;
      }
    }
  } while(!fim);

  tip = tip->proxima_p;
}

if (p2mco == NULL)
{
  p2mco = p2opt;
}
else
{
  if (p2mco->cont_fila > p2opt->cont_fila)
  {
    prox = p2mco->inicio.proxima_p;
    while(prox != &p2mco->fim)
    {
      temp = prox;
      prox = temp->proxima_p;
      free(temp);
    }

    free(p2mco);

    p2mco = p2opt;
  }
  else
  {
    prox = p2opt->inicio.proxima_p;
    while(prox != &p2opt->fim)
    {
      temp = prox;
      prox = temp->proxima_p;
      free(temp);
    }

    free(p2opt);
  }
}

p2opt = (TIPO_OTIMIZADOR *)pop();
i++;
}
```

```
  fprintf(p2e, "%s Resultado final p/o arq:
          [%s]\n", comment, p2f);
  fprintf(p2e, "%s No. instr. inicio:[%d],
          No. instr. fim:[%d]\n",
          comment, p2mco->cont_inst, p2mco->cont_fila);
  prox = p2mco->inicio.proxima_p;
  while(prox != &p2mco->fim)
  {
    temp = prox;
    fprintf(p2o, "%s", temp->instrucao);
    prox = temp->proxima_p;
    free(temp);
  }

  free(p2mco);
  dstack();
  return(0);
}
```

APÊNDICE E – Instalação do Ambiente LCC no MS-Windows 2000

Primeiramente, criar a variável de ambiente BUILDDIR para armazenar o subdiretório onde ficará instalado o LCC e o subdiretório armazenado em BUILDDIR. Exemplo:

```
D:\lcc\4.2>set BUILDDIR=e:\arquiv~1\lcc\4.2\bin
D:\lcc\4.2>mkdir %BUILDDIR%
```

ETC\WIN32.C é o arquivo do programa acionador que contém as dependências específicas do ambiente de programação do compilador Visual C/C++ 6.0. ETC\WIN32.C assume que a variável de ambiente INCLUDE fornece a localização dos arquivos de cabeçalho e que o editor de ligações, *link.exe*, e o montador, *ml.exe*, se encontram na variável de ambiente PATH. Também assume que a variável de ambiente LCCDIR armazena o subdiretório de instalação do LCC. Se esta variável não existe, deve-se criá-la conforme mostrado abaixo:

```
D:\lcc\4.2>set LCCDIR=e:\arquiv~1\lcc\4.2\bin
```

Em seguida, gerar o programa acionador do LCC, o *lcc.exe*, a partir dos arquivos LCC.C e WIN32.C que se encontram no subdiretório ETC da distribuição do LCC. Exemplo:

```
D:\lcc\4.2>nmake -f makefile.nt HOSTFILE=etc/win32.c lcc
...
    cl -nologo -Zi -MLd -Fd%BUILDDIR%\ -c -Fo%BUILDDIR%\lcc.obj etc/lcc.c
lcc.c
    cl -nologo -Zi -MLd -Fd%BUILDDIR%\ -c -Fo%BUILDDIR%\host.obj etc/win32.c
win32.c
    cl -nologo -Zi -MLd -Fd%BUILDDIR%\ -Fe%BUILDDIR%\lcc.exe
%BUILDDIR%\lcc.obj %BUILDDIR%\host.obj
```

Em seguida, gerar o pré-processador, *cpp.exe*, o compilador propriamente dito, *rcc.exe*, bibliotecas, *liblcc.lib* e *librcc.lib*, e outros utilitários necessários, *bprint.exe* e *lburg.exe*. Exemplo:

```
D:\lcc\4.2>nmake -f makefile.nt all
```

Criar um subdiretório de teste e executar a suíte de testes. Exemplo:

```
D:\lcc\4.2>mkdir %BUILDDIR%\x86\win32\tst
D:\lcc\4.2>nmake -f makefile.nt test
```

Este comando compila cada programa no subdiretório TST, compara o código em linguagem de montagem e os diagnósticos gerados com o código em linguagem de montagem e os diagnósticos esperados, executa o programa de teste e compara os resultados gerados com os resultados esperados. A comparação dos arquivos é feito pelo utilitário *fc.exe*. A quantidade de dados produzida neste passo é grande, mas não se deve observar nenhum erro ou discrepância.

Copiar o arquivo *lcc.exe* e *bprint.exe* para o subdiretório que se encontra armazenado na variável de ambiente PATH. Exemplo:

```
D:\lcc\4.2>copy %BUILDDIR%\lcc.exe \bin
        1 arquivo(s) copiado(s)

D:\lcc\4.2>copy %BUILDDIR%\bprint.exe \bin
        1 arquivo(s) copiado(s)
```

Finalmente, apagar os arquivos temporários gerados. Exemplo:

```
D:\lcc\4.2>nmake -f makefile.nt clean
```

Este comando apaga os arquivos temporários em BUILDDIR, mas não apaga o *rcc.exe*, etc. O arquivo HTML DOC/INSTALL.HTML fornece mais detalhes sobre a distribuição e as instruções para a instalação do LCC no ambiente MS-Windows 2000 (Fraser; Hanson, 2004).

Uma vez instalado o LCC, uma série de arquivos de lote foram elaborados para automatizar o processo descrito na figura 5.1. Estes arquivos estão descritos no apêndice F. O programa executável do otimizador *peephole* adaptativo (OPA) foi produzido com o auxílio destes arquivos. Exemplo:

```
D:\projetos\c\opa>cml opa
D:\projetos\c\opa>lcc -S opa.c
...
 Assembling: opa.asm
```

APÊNDICE F - Arquivos de Lote para a Geração de Executáveis c/o LCC

Arquivo C.BAT (Compila c/ LCC)

```
@echo off
if "%1" == "" goto nenhum
if "%2" == "" goto um_arq
if "%3" == "" goto dois_arqs
if "%4" == "" goto tres_arqs
if "%5" == "" goto quat_arqs
if "%6" == "" goto cinc_arqs
if "%7" == "" goto seis_arqs
if "%8" == "" goto sete_arqs
if "%9" == "" goto oito_arqs
if "%10" == "" goto nove_arqs
if "%11" == "" goto dez_arqs
goto demais

:um_arq
@echo on
lcc -S %1.c
@echo off
goto fim

:dois_arqs
@echo on
lcc -S %1.c %2.c
@echo off
goto fim

:tres_arqs
@echo on
lcc -S %1.c %2.c %3.c
@echo off
goto fim

:quat_arqs
@echo on
lcc -S %1.c %2.c %3.c %4.c
@echo off
goto fim

:cinc_arqs
@echo on
lcc -S %1.c %2.c %3.c %4.c %5.c
@echo off
goto fim

:seis_arqs
@echo on
lcc -S %1.c %2.c %3.c %4.c %5.c %6.c
@echo off
goto fim

:sete_arqs
@echo on
lcc -S %1.c %2.c %3.c %4.c %5.c %6.c %7.c
@echo off
goto fim

:oito_arqs
@echo on
lcc -S %1.c %2.c %3.c %4.c %5.c %6.c %7.c %8.c
@echo off
goto fim

:nove_arqs
@echo on
lcc -S %1.c %2.c %3.c %4.c %5.c %6.c %7.c %8.c %9.c
@echo off
goto fim

:dez_arqs
@echo on
lcc  -S  %1.c  %2.c  %3.c  %4.c  %5.c  %6.c  %7.c  %8.c  %9.c
%10.c
@echo off
goto fim

:nenhum
echo _
echo *
echo * Erro: Especifique pelo menos um arquivo !!
echo *
goto help

:demais
echo _
echo *
echo * Erro: Excesso de arquivos especificados !!
echo *
goto help

:help
echo _
echo *
echo * Sintaxe: c arq1 [arq2] ... [arq10]
echo *
echo *          onde: arqx = arquivos *.c do projeto
echo *
echo * Exemplo: c palindro llgen llapp
echo *
goto fim

:fim
```

Arquivo M.BAT (Monta c/ ML)

```
@echo off
if "%1" == "" goto nenhum
if "%2" == "" goto um_arq
if "%3" == "" goto dois_arqs
if "%4" == "" goto tres_arqs
if "%5" == "" goto quat_arqs
if "%6" == "" goto cinc_arqs
if "%7" == "" goto seis_arqs
if "%8" == "" goto sete_arqs
if "%9" == "" goto oito_arqs
if "%10" == "" goto nove_arqs
if "%11" == "" goto dez_arqs
goto demais

:um_arq
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
goto fim

:dois_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
goto fim

:tres_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%3.asm
goto fim

:quat_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%3.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%4.asm
goto fim

:cinc_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%3.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%4.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%5.asm
goto fim

:seis_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%3.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%4.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%5.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%6.asm
goto fim

:sete_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%3.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%4.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
```

```
%5.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%6.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%7.asm
goto fim

:oito_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%3.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%4.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%5.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%6.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%7.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%8.asm
goto fim

:nove_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%3.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%4.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%5.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%6.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%7.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%8.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%9.asm
goto fim

:dez_arqs
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%1.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%2.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%3.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%4.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%5.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%6.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%7.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%8.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%9.asm
e:\arquiv~1\micros~3\vc98\bin\ml   /c   /coff   /Cp   /W0   /Zi
%10.asm
goto fim

:nenhum
echo _
echo *
echo * Erro: Especifique pelo menos um arquivo !!
echo *
goto help

:demais
echo _
echo *
echo * Erro: Excesso de arquivos especificados !!
echo *
goto help

:help
echo _
echo *
echo * Sintaxe: m arq1 [arq2] ... [arq10]
echo *
echo *          onde: arqx = arquivos *.c do projeto
echo *
echo * Exemplo: m palindro llgen llapp
echo *
goto fim

:fim
```

Arquivo L.BAT (Liga c/ *link.exe*)

```
@echo off
if "%1" == "" goto nenhum
if "%2" == "" goto um_arq
if "%3" == "" goto dois_arqs
if "%4" == "" goto tres_arqs
if "%5" == "" goto quat_arqs
if "%6" == "" goto cinc_arqs
if "%7" == "" goto seis_arqs
if "%8" == "" goto sete_arqs
if "%9" == "" goto oito_arqs
if "%10" == "" goto nove_arqs
if "%11" == "" goto dez_arqs
goto demais

:um_arq
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib %1.obj
goto fim

:dois_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib %1.obj %2.obj
goto fim

:tres_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib %1.obj %2.obj %3.obj
goto fim

:quat_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib %1.obj %2.obj %3.obj %4.obj
goto fim

:cinc_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib %1.obj %2.obj %3.obj %4.obj %5.obj
goto fim

:seis_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib %1.obj %2.obj %3.obj %4.obj %5.obj %6.obj
goto fim

:sete_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib   %1.obj   %2.obj   %3.obj   %4.obj   %5.obj   %6.obj
%7.obj
goto fim

:oito_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib   %1.obj   %2.obj   %3.obj   %4.obj   %5.obj   %6.obj
%7.obj %8.obj
goto fim

:nove_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib   %1.obj   %2.obj   %3.obj   %4.obj   %5.obj   %6.obj
%7.obj %8.obj %9.obj
goto fim

:dez_arqs
e:\arquiv~1\micros~3\vc98\bin\link.exe                    /NOLOGO
/SUBSYSTEM:CONSOLE          /INCREMENTAL:NO          /MACHINE:I386
/OUT:"%1.exe"            /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB"
libcd.lib   %1.obj   %2.obj   %3.obj   %4.obj   %5.obj   %6.obj
%7.obj %8.obj %9.obj %10.obj
goto fim

:nenhum
echo _
echo *
echo * Erro: Especifique pelo menos um arquivo !!
echo *
goto help

:demais
```

```
echo _
echo *
echo * Erro: Excesso de arquivos especificados !!
echo *
goto help

:help
echo _
echo *
echo * Sintaxe: l arq1 [arq2] ... [arq10]
echo *
echo *          onde: arqx = arquivos *.c do projeto
echo *
echo * Exemplo: l palindro llgen llapp
echo *
goto fim

:fim
```

Arquivo O.BAT (Otimiza c/ OPA)

```
@echo off
if "%1" == "" goto nenhum
if "%2" == "" goto um_arq
if "%3" == "" goto dois_arqs
if "%4" == "" goto tres_arqs
if "%5" == "" goto quat_arqs
if "%6" == "" goto cinc_arqs
if "%7" == "" goto seis_arqs
if "%8" == "" goto sete_arqs
if "%9" == "" goto oito_arqs
if "%10" == "" goto nove_arqs
if "%11" == "" goto dez_arqs
goto demais

:um_arq
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del %1.asm
ren %1.a %1.asm
goto fim

:dois_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del %1.asm %2.asm
ren %1.a %1.asm
ren %2.a %2.asm
goto fim

:tres_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
opa -sx86 -o%3.a rulesx86.v30 %3.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del %1.asm %2.asm %3.asm
ren %1.a %1.asm
ren %2.a %2.asm
ren %3.a %3.asm
goto fim

:quat_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
opa -sx86 -o%3.a rulesx86.v30 %3.asm
if errorlevel 1 goto fim
opa -sx86 -o%4.a rulesx86.v30 %4.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del %1.asm %2.asm %3.asm %4.asm
ren %1.a %1.asm
ren %2.a %2.asm
ren %3.a %3.asm
ren %4.a %4.asm
goto fim

:cinc_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
opa -sx86 -o%3.a rulesx86.v30 %3.asm
if errorlevel 1 goto fim
opa -sx86 -o%4.a rulesx86.v30 %4.asm
if errorlevel 1 goto fim
opa -sx86 -o%5.a rulesx86.v30 %5.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del %1.asm %2.asm %3.asm %4.asm %5.asm
ren %1.a %1.asm
ren %2.a %2.asm
ren %3.a %3.asm
ren %4.a %4.asm
ren %5.a %5.asm
goto fim

:seis_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
opa -sx86 -o%3.a rulesx86.v30 %3.asm
if errorlevel 1 goto fim
opa -sx86 -o%4.a rulesx86.v30 %4.asm
if errorlevel 1 goto fim
opa -sx86 -o%5.a rulesx86.v30 %5.asm
if errorlevel 1 goto fim
opa -sx86 -o%6.a rulesx86.v30 %6.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del %1.asm %2.asm %3.asm %4.asm %5.asm %6.asm
ren %1.a %1.asm
ren %2.a %2.asm
ren %3.a %3.asm
ren %4.a %4.asm
ren %5.a %5.asm
ren %6.a %6.asm
goto fim

:sete_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
opa -sx86 -o%3.a rulesx86.v30 %3.asm
if errorlevel 1 goto fim
opa -sx86 -o%4.a rulesx86.v30 %4.asm
if errorlevel 1 goto fim
opa -sx86 -o%5.a rulesx86.v30 %5.asm
if errorlevel 1 goto fim
opa -sx86 -o%6.a rulesx86.v30 %6.asm
if errorlevel 1 goto fim
opa -sx86 -o%7.a rulesx86.v30 %7.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del %1.asm %2.asm %3.asm %4.asm %5.asm %6.asm %7.asm
ren %1.a %1.asm
ren %2.a %2.asm
ren %3.a %3.asm
ren %4.a %4.asm
ren %5.a %5.asm
ren %6.a %6.asm
ren %7.a %7.asm
goto fim

:oito_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
opa -sx86 -o%3.a rulesx86.v30 %3.asm
if errorlevel 1 goto fim
opa -sx86 -o%4.a rulesx86.v30 %4.asm
if errorlevel 1 goto fim
opa -sx86 -o%5.a rulesx86.v30 %5.asm
if errorlevel 1 goto fim
opa -sx86 -o%6.a rulesx86.v30 %6.asm
if errorlevel 1 goto fim
opa -sx86 -o%7.a rulesx86.v30 %7.asm
if errorlevel 1 goto fim
opa -sx86 -o%8.a rulesx86.v30 %8.asm
if errorlevel 1 goto fim
clock >> %1.txt
```

```
@echo off
del   %1.asm   %2.asm   %3.asm   %4.asm   %5.asm   %6.asm   %7.asm
%8.asm
ren %1.a %1.asm
ren %2.a %2.asm
ren %3.a %3.asm
ren %4.a %4.asm
ren %5.a %5.asm
ren %6.a %6.asm
ren %7.a %7.asm
ren %8.a %8.asm
goto fim

:nove_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
opa -sx86 -o%3.a rulesx86.v30 %3.asm
if errorlevel 1 goto fim
opa -sx86 -o%4.a rulesx86.v30 %4.asm
if errorlevel 1 goto fim
opa -sx86 -o%5.a rulesx86.v30 %5.asm
if errorlevel 1 goto fim
opa -sx86 -o%6.a rulesx86.v30 %6.asm
if errorlevel 1 goto fim
opa -sx86 -o%7.a rulesx86.v30 %7.asm
if errorlevel 1 goto fim
opa -sx86 -o%8.a rulesx86.v30 %8.asm
if errorlevel 1 goto fim
opa -sx86 -o%9.a rulesx86.v30 %9.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del   %1.asm   %2.asm   %3.asm   %4.asm   %5.asm   %6.asm   %7.asm
%8.asm %9.asm
ren %1.a %1.asm
ren %2.a %2.asm
ren %3.a %3.asm
ren %4.a %4.asm
ren %5.a %5.asm
ren %6.a %6.asm
ren %7.a %7.asm
ren %8.a %8.asm
ren %9.a %9.asm
goto fim

:dez_arqs
@echo on
clock >> %1.txt
opa -sx86 -o%1.a rulesx86.v30 %1.asm
if errorlevel 1 goto fim
opa -sx86 -o%2.a rulesx86.v30 %2.asm
if errorlevel 1 goto fim
opa -sx86 -o%3.a rulesx86.v30 %3.asm
if errorlevel 1 goto fim
opa -sx86 -o%4.a rulesx86.v30 %4.asm
if errorlevel 1 goto fim
opa -sx86 -o%5.a rulesx86.v30 %5.asm
if errorlevel 1 goto fim
opa -sx86 -o%6.a rulesx86.v30 %6.asm
if errorlevel 1 goto fim
opa -sx86 -o%7.a rulesx86.v30 %7.asm
if errorlevel 1 goto fim
opa -sx86 -o%8.a rulesx86.v30 %8.asm
if errorlevel 1 goto fim
opa -sx86 -o%9.a rulesx86.v30 %9.asm
if errorlevel 1 goto fim
opa -sx86 -o%10.a rulesx86.v30 %10.asm
if errorlevel 1 goto fim
clock >> %1.txt
@echo off
del   %1.asm   %2.asm   %3.asm   %4.asm   %5.asm   %6.asm   %7.asm
%8.asm %9.asm %10.asm
ren %1.a %1.asm
ren %2.a %2.asm
ren %3.a %3.asm
ren %4.a %4.asm
ren %5.a %5.asm
ren %6.a %6.asm
ren %7.a %7.asm
ren %8.a %8.asm
ren %9.a %9.asm
ren %10.a %10.asm
goto fim

:nenhum
echo _
echo *
echo * Erro: Especifique pelo menos um arquivo !!
echo *
goto help

:demais
echo _
echo *
echo * Erro: Excesso de arquivos especificados !!
echo *
goto help

:help
echo _
echo *
echo * Sintaxe: o arq1 [arq2] ... [arq10]
echo *
echo *          onde: arqx = arquivos *.c do projeto
echo *
echo * Exemplo: o palindro llgen llapp
echo *
goto fim

:fim
```

Arquivo CML.BAT (Compila, Monta e Liga)

```
@echo off
cls
if "%1" == "" goto nenhum
if "%2" == "" goto um_arq
if "%3" == "" goto dois_arqs
if "%4" == "" goto tres_arqs
if "%5" == "" goto quat_arqs
if "%6" == "" goto cinc_arqs
if "%7" == "" goto seis_arqs
if "%8" == "" goto sete_arqs
if "%9" == "" goto oito_arqs
if "%10" == "" goto nove_arqs
if "%11" == "" goto dez_arqs
goto demais

:um_arq
@call c %1
if errorlevel 1 goto fim
@call m %1
if errorlevel 1 goto fim
@call l %1
goto fim

:dois_arqs
@call c %1 %2
if errorlevel 1 goto fim
@call m %1 %2
if errorlevel 1 goto fim
@call l %1 %2
goto fim

:tres_arqs
@call c %1 %2 %3
if errorlevel 1 goto fim
@call m %1 %2 %3
if errorlevel 1 goto fim
@call l %1 %2 %3
goto fim

:quat_arqs
@call c %1 %2 %3 %4
if errorlevel 1 goto fim
@call m %1 %2 %3 %4
if errorlevel 1 goto fim
@call l %1 %2 %3 %4
goto fim

:cinc_arqs
@call c %1 %2 %3 %4 %5
if errorlevel 1 goto fim
@call m %1 %2 %3 %4 %5
if errorlevel 1 goto fim
@call l %1 %2 %3 %4 %5
goto fim

:seis_arqs
@call c %1 %2 %3 %4 %5 %6
if errorlevel 1 goto fim
@call m %1 %2 %3 %4 %5 %6
if errorlevel 1 goto fim
@call l %1 %2 %3 %4 %5 %6
goto fim

:sete_arqs
@call c %1 %2 %3 %4 %5 %6 %7
if errorlevel 1 goto fim
@call m %1 %2 %3 %4 %5 %6 %7
if errorlevel 1 goto fim
@call l %1 %2 %3 %4 %5 %6 %7
goto fim

:oito_arqs
@call c %1 %2 %3 %4 %5 %6 %7 %8
if errorlevel 1 goto fim
@call m %1 %2 %3 %4 %5 %6 %7 %8
if errorlevel 1 goto fim
@call l %1 %2 %3 %4 %5 %6 %7 %8
goto fim
```

```
:nove_arqs
@call c %1 %2 %3 %4 %5 %6 %7 %8 %9
if errorlevel 1 goto fim
@call m %1 %2 %3 %4 %5 %6 %7 %8 %9
if errorlevel 1 goto fim
@call l %1 %2 %3 %4 %5 %6 %7 %8 %9
goto fim

:dez_arqs
@call c %1 %2 %3 %4 %5 %6 %7 %8 %9 %10
if errorlevel 1 goto fim
@call m %1 %2 %3 %4 %5 %6 %7 %8 %9 %10
if errorlevel 1 goto fim
@call l %1 %2 %3 %4 %5 %6 %7 %8 %9 %10
goto fim

:nenhum
echo _
echo *
echo * Erro: Especifique pelo menos um arquivo !!
echo *
goto help

:demais
echo _
echo *
echo * Erro: Excesso de arquivos especificados !!
echo *
goto help

:help
echo _
echo *
echo * Sintaxe: cml arq1 [arq2] ... [arq10]
echo *
echo *          onde: arqx = arquivos *.c do projeto
echo *
echo * Exemplo: cml palindro llgen llapp
echo *
goto fim

:fim
```

APÊNDICE G - Arquivo de *nmake* p/ Realização de *Bootstrapping* do LCC

```
# $Id: makefile.nt,v 2.3 2002/09/04 18:33:11 drh Exp $
# Modified by JCL in 2004/10/04
HOSTFILE=etc/win32.c
TARGET=x86\win32
TEMPDIR=\\temp
A=.lib
O=.obj
E=.exe
CC=lcc -S -DWIN32
CL=cl -nologo
CFLAGS=-DWIN32 -Zi -MLd -Fd$(BUILDDIR)^\
OPTIM=opa -sx86
RULES=rulesx86.v30
ML=ml
MFLAGS=/nologo /c /coff /Cp /w0 /Zi
LD=cl -nologo
LDFLAGS=-Zi -MLd -Fd$(BUILDDIR)^\
LOPT=/NOLOGO /SUBSYSTEM:CONSOLE /INCREMENTAL:NO /MACHINE:I386 /LIBPATH:"e:\arquiv~1\micros~3\vc98\LIB" libcd.lib
TSTDIR=$(BUILDDIR)\$(TARGET)\tst
B=$(BUILDDIR)^\
T=$(TSTDIR)^\
C=$Blcc -Wo-lccdir=$(BUILDDIR) -Wf-target=$(TARGET) -Iinclude\$(TARGET)

what:
  -@echo make all rcc lburg cpp lcc bprint liblcc triple clean clobber

all::  rcc lburg cpp lcc bprint liblcc

rcc:   $Brcc$E
lburg:    $Blburg$E
cpp:   $Bcpp$E
lcc:   $Blcc$E
bprint:   $Bbprint$E
liblcc:   $Bliblcc$A

RCCOBJS=$Balloc$O \
  $Bbind$O \
  $Bdag$O \
  $Bdagcheck$O \
  $Bdecl$O \
  $Benode$O \
  $Berror$O \
  $Bexpr$O \
  $Bevent$O \
  $Binit$O \
  $Binits$O \
  $Binput$O \
  $Blex$O \
  $Blist$O \
  $Bmain$O \
  $Boutput$O \
  $Bprof$O \
  $Bprofio$O \
  $Bsimp$O \
  $Bstmt$O \
  $Bstring$O \
  $Bsym$O \
  $Btrace$O \
  $Btree$O \
  $Btypes$O \
  $Bnull$O \
  $Bsymbolic$O \
  $Bgen$O \
  $Bbytecode$O \
  $Balpha$O \
  $Bmips$O \
  $Bsparc$O \
  $Bstab$O \
  $Bx86$O \
  $Bx86linux$O

$Brcc$E::   $Bmain$O $Blibrcc$A $(EXTRAOBJS)
     $(LD) $(LDFLAGS) -Fe$@ $Bmain$O $(EXTRAOBJS) $Blibrcc$A $(EXTRALIBS)

$Blibrcc$A:    $(RCCOBJS)
     lib -out:$@ $(RCCOBJS)

$(RCCOBJS):    src/c.h src/ops.h src/token.h src/config.h

$Balloc$O:     $Balloc.asm;     $(ML) $(MFLAGS) /Fo$@ $Balloc.asm
$Bbind$O:    $Bbind.asm;    $(ML) $(MFLAGS) /Fo$@ $Bbind.asm
$Bdag$O:    $Bdag.asm;    $(ML) $(MFLAGS) /Fo$@ $Bdag.asm
$Bdecl$O:    $Bdecl.asm;    $(ML) $(MFLAGS) /Fo$@ $Bdecl.asm
$Benode$O:     $Benode.asm;     $(ML) $(MFLAGS) /Fo$@ $Benode.asm
$Berror$O:     $Berror.asm;     $(ML) $(MFLAGS) /Fo$@ $Berror.asm
$Bevent$O:     $Bevent.asm;     $(ML) $(MFLAGS) /Fo$@ $Bevent.asm
$Bexpr$O:    $Bexpr.asm;    $(ML) $(MFLAGS) /Fo$@ $Bexpr.asm
$Bgen$O:    $Bgen.asm;    $(ML) $(MFLAGS) /Fo$@ $Bgen.asm
$Binit$O:    $Binit.asm;    $(ML) $(MFLAGS) /Fo$@ $Binit.asm
$Binits$O:     $Binits.asm;     $(ML) $(MFLAGS) /Fo$@ $Binits.asm
$Binput$O:     $Binput.asm;     $(ML) $(MFLAGS) /Fo$@ $Binput.asm
$Blex$O:    $Blex.asm;    $(ML) $(MFLAGS) /Fo$@ $Blex.asm
$Blist$O:    $Blist.asm;    $(ML) $(MFLAGS) /Fo$@ $Blist.asm
$Bmain$O:    $Bmain.asm;    $(ML) $(MFLAGS) /Fo$@ $Bmain.asm
$Bnull$O:    $Bnull.asm;    $(ML) $(MFLAGS) /Fo$@ $Bnull.asm
```

```
$Boutput$O:   $Boutput.asm;   $(ML) $(MFLAGS) /Fo$@ $Boutput.asm
$Bprof$O:   $Bprof.asm;   $(ML) $(MFLAGS) /Fo$@ $Bprof.asm
$Bprofio$O:   $Bprofio.asm;   $(ML) $(MFLAGS) /Fo$@ $Bprofio.asm
$Bsimp$O:   $Bsimp.asm;   $(ML) $(MFLAGS) /Fo$@ $Bsimp.asm
$Bstmt$O:   $Bstmt.asm;   $(ML) $(MFLAGS) /Fo$@ $Bstmt.asm
$Bstring$O:   $Bstring.asm;   $(ML) $(MFLAGS) /Fo$@ $Bstring.asm
$Bsym$O:   $Bsym.asm;   $(ML) $(MFLAGS) /Fo$@ $Bsym.asm
$Bsymbolic$O:   $Bsymbolic.asm;   $(ML) $(MFLAGS) /Fo$@ $Bsymbolic.asm
$Bbytecode$O:   $Bbytecode.asm;   $(ML) $(MFLAGS) /Fo$@ $Bbytecode.asm
$Btrace$O:   $Btrace.asm;   $(ML) $(MFLAGS) /Fo$@ $Btrace.asm
$Btree$O:   $Btree.asm;   $(ML) $(MFLAGS) /Fo$@ $Btree.asm
$Btypes$O:   $Btypes.asm;   $(ML) $(MFLAGS) /Fo$@ $Btypes.asm
$Bstab$O:   $Bstab.asm;   $(ML) $(MFLAGS) /Fo$@ $Bstab.asm

$Balloc.asm:   $Balloc.a;   $(OPTIM) -o$@ $(RULES) $Balloc.a
$Bbind.asm:   $Bbind.a;   $(OPTIM) -o$@  $(RULES) $Bbind.a
$Bdag.asm:   $Bdag.a;   $(OPTIM) -o$@  $(RULES) $Bdag.a
$Bdecl.asm:   $Bdecl.a;   $(OPTIM) -o$@  $(RULES) $Bdecl.a
$Benode.asm:   $Benode.a;   $(OPTIM) -o$@  $(RULES) $Benode.a
$Berror.asm:   $Berror.a;   $(OPTIM) -o$@  $(RULES) $Berror.a
$Bevent.asm:   $Bevent.a;   $(OPTIM) -o$@  $(RULES) $Bevent.a
$Bexpr.asm:   $Bexpr.a;   $(OPTIM) -o$@  $(RULES) $Bexpr.a
$Bgen.asm:   $Bgen.a;   $(OPTIM) -o$@  $(RULES) $Bgen.a
$Binit.asm:   $Binit.a;   $(OPTIM) -o$@  $(RULES) $Binit.a
$Binits.asm:   $Binits.a;   $(OPTIM) -o$@  $(RULES) $Binits.a
$Binput.asm:   $Binput.a;   $(OPTIM) -o$@  $(RULES) $Binput.a
$Blex.asm:   $Blex.a;   $(OPTIM) -o$@  $(RULES) $Blex.a
$Blist.asm:   $Blist.a;   $(OPTIM) -o$@  $(RULES) $Blist.a
$Bmain.asm:   $Bmain.a;   $(OPTIM) -o$@  $(RULES) $Bmain.a
$Bnull.asm:   $Bnull.a;   $(OPTIM) -o$@  $(RULES) $Bnull.a
$Boutput.asm:   $Boutput.a;   $(OPTIM) -o$@  $(RULES) $Boutput.a
$Bprof.asm:   $Bprof.a;   $(OPTIM) -o$@  $(RULES) $Bprof.a
$Bprofio.asm:   $Bprofio.a;   $(OPTIM) -o$@  $(RULES) $Bprofio.a
$Bsimp.asm:   $Bsimp.a;   $(OPTIM) -o$@  $(RULES) $Bsimp.a
$Bstmt.asm:   $Bstmt.a;   $(OPTIM) -o$@  $(RULES) $Bstmt.a
$Bstring.asm:   $Bstring.a;   $(OPTIM) -o$@  $(RULES) $Bstring.a
$Bsym.asm:   $Bsym.a;   $(OPTIM) -o$@  $(RULES) $Bsym.a
$Bsymbolic.asm:   $Bsymbolic.a;   $(OPTIM) -o$@  $(RULES) $Bsymbolic.a
$Bbytecode.asm:   $Bbytecode.a;   $(OPTIM) -o$@  $(RULES) $Bbytecode.a
$Btrace.asm:   $Btrace.a;   $(OPTIM) -o$@  $(RULES) $Btrace.a
$Btree.asm:   $Btree.a;  $(OPTIM) -o$@  $(RULES) $Btree.a
$Btypes.asm:   $Btypes.a;   $(OPTIM) -o$@  $(RULES) $Btypes.a
$Bstab.asm:   $Bstab.a;   $(OPTIM) -o$@  $(RULES) $Bstab.a

$Balloc.a:   src/alloc.c;   $(CC) -o $@ src/alloc.c
$Bbind.a:   src/bind.c;   $(CC) -o $@ src/bind.c
$Bdag.a:   src/dag.c;   $(CC) -o $@ src/dag.c
$Bdecl.a:   src/decl.c;   $(CC) -o $@ src/decl.c
$Benode.a:   src/enode.c;   $(CC) -o $@ src/enode.c
$Berror.a:   src/error.c;   $(CC) -o $@ src/error.c
$Bevent.a:   src/event.c;   $(CC) -o $@ src/event.c
$Bexpr.a:   src/expr.c;   $(CC) -o $@ src/expr.c
$Bgen.a:   src/gen.c;   $(CC) -o $@ src/gen.c
$Binit.a:   src/init.c;   $(CC) -o $@ src/init.c
$Binits.a:   src/inits.c;   $(CC) -o $@ src/inits.c
$Binput.a:   src/input.c;   $(CC) -o $@ src/input.c
$Blex.a:   src/lex.c;   $(CC) -o $@ src/lex.c
$Blist.a:   src/list.c;   $(CC) -o $@ src/list.c
$Bmain.a:   src/main.c;   $(CC) -o $@ src/main.c
$Bnull.a:   src/null.c;   $(CC) -o $@ src/null.c
$Boutput.a:   src/output.c;   $(CC) -o $@ src/output.c
$Bprof.a:   src/prof.c;   $(CC) -o $@ src/prof.c
$Bprofio.a:   src/profio.c;   $(CC) -o $@ src/profio.c
$Bsimp.a:   src/simp.c;   $(CC) -o $@ src/simp.c
$Bstmt.a:   src/stmt.c;   $(CC) -o $@ src/stmt.c
$Bstring.a:   src/string.c;   $(CC) -o $@ src/string.c
$Bsym.a:   src/sym.c;   $(CC) -o $@ src/sym.c
$Bsymbolic.a:   src/symbolic.c;   $(CC) -o $@ src/symbolic.c
$Bbytecode.a:   src/bytecode.c;   $(CC) -o $@ src/bytecode.c
$Btrace.a:   src/trace.c;   $(CC) -o $@ src/trace.c
$Btree.a:   src/tree.c;   $(CC) -o $@ src/tree.c
$Btypes.a:   src/types.c;   $(CC) -o $@ src/types.c
$Bstab.a:   src/stab.c src/stab.h; $(CC) -o $@ src/stab.c

$Bdagcheck$O:   $Bdagcheck.asm;   $(ML) $(MFLAGS) /Fo$@ $Bdagcheck.asm
$Balpha$O:   $Balpha.asm;   $(ML) $(MFLAGS) /Fo$@ $Balpha.asm
$Bmips$O:   $Bmips.asm;   $(ML) $(MFLAGS) /Fo$@ $Bmips.asm
$Bsparc$O:   $Bsparc.asm;   $(ML) $(MFLAGS) /Fo$@ $Bsparc.asm
$Bx86$O:   $Bx86.asm;   $(ML) $(MFLAGS) /Fo$@ $Bx86.asm
$Bx86linux$O:   $Bx86linux.c;   $(CL) $(CFLAGS) -c -Isrc -Fo$@ $Bx86linux.c

$Bdagcheck.asm:   $Bdagcheck.a;   $(OPTIM) -o$@  $(RULES) $Bdagcheck.a
$Balpha.asm:   $Balpha.a;   $(OPTIM) -o$@  $(RULES) $Balpha.a
$Bmips.asm:   $Bmips.a;   $(OPTIM) -o$@  $(RULES) $Bmips.a
$Bsparc.asm:   $Bsparc.a;   $(OPTIM) -o$@  $(RULES) $Bsparc.a
$Bx86.asm:   $Bx86.a;   $(OPTIM) -o$@  $(RULES) $Bx86.a

$Bdagcheck.a:   $Bdagcheck.c;   $(CC) -Isrc -o $@ $Bdagcheck.c
$Balpha.a:   $Balpha.c;   $(CC) -Isrc -o $@ $Balpha.c
$Bmips.a:   $Bmips.c;   $(CC) -Isrc -o $@ $Bmips.c
$Bsparc.a:   $Bsparc.c;   $(CC) -Isrc -o $@ $Bsparc.c
$Bx86.a:   $Bx86.c;   $(CC) -Isrc -o $@ $Bx86.c

$Bdagcheck.c:   $Blburg$E src/dagcheck.md; $Blburg src/dagcheck.md $@
$Balpha.c:   $Blburg$E src/alpha.md;   $Blburg src/alpha.md   $@
$Bmips.c:   $Blburg$E src/mips.md;   $Blburg src/mips.md   $@
$Bsparc.c:   $Blburg$E src/sparc.md;   $Blburg src/sparc.md   $@
$Bx86.c:   $Blburg$E src/x86.md;   $Blburg src/x86.md   $@
```

```
$Bx86linux.c:   $Blburg$E src/x86linux.md; $Blburg src/x86linux.md $@

$Bbprint$E:   $Bbprint$O;   $(LD) $(LDFLAGS) -Fe$@ $Bbprint$O -link $(LOPT)

$Bbprint$O:   $Bbprint.asm;   $(ML) $(MFLAGS) /Fo$@ $Bbprint.asm
$Bbprint.asm:   $Bbprint.a;   $(OPTIM) -o$@  $(RULES) $Bbprint.a
$Bbprint.a:   etc/bprint.c src/profio.c;       $(CC) -Isrc -o $@ etc/bprint.c

$Bops$E:    $Bops$O;    $(LD) $(LDFLAGS) -Fe$@ $Bops$O -link $(LOPT)

$Bops$O:    $Bops.asm;    $(ML) $(MFLAGS) /Fo$@ $Bops.asm
$Bops.asm:    $Bops.a;    $(OPTIM) -o$@  $(RULES) $Bops.a
$Bops.a:    etc/ops.c src/ops.h;         $(CC) -Isrc -o $@ etc/ops.c

$Blcc$E:    $Blcc$O $Bhost$O;    $(LD) $(LDFLAGS) -Fe$@ $Blcc$O $Bhost$O -link $(LOPT)

$Blcc$O:    etc/lcc.c;    $(CL) $(CFLAGS) -c -DTEMPDIR=\"$(TEMPDIR)\" -Fo$@ etc/lcc.c
$Bhost$O:   $Bwin32.asm;     $(ML) $(MFLAGS) /Fo$@ $Bwin32.asm

$Bwin32.asm:    $Bwin32.a;    $(OPTIM) -o$@  $(RULES) $Bwin32.a
$Bwin32.a:    $(HOSTFILE);     $(CC) -o $@ $(HOSTFILE)

$Bcp$E:     etc/cp.c;   $(CL) $(CFLAGS) -Fo$@ etc/cp.c

LIBOBJS=$Bassert$O $Bbbexit$O $Byynull$O

$Bliblcc$A:   $(LIBOBJS);   lib -out:$@ $Bassert$O $Bbbexit$O $Byynull$O

$Bassert$O:   $Bassert.asm;   $(ML) $(MFLAGS) /Fo$@ $Bassert.asm
$Byynull$O:   $Byynull.asm;   $(ML) $(MFLAGS) /Fo$@ $Byynull.asm
$Bbbexit$O:   $Bbbexit.asm;   $(ML) $(MFLAGS) /Fo$@ $Bbbexit.asm

$Bassert.asm:   $Bassert.a;   $(OPTIM) -o$@  $(RULES) $Bassert.a
$Byynull.asm:   $Byynull.a;   $(OPTIM) -o$@  $(RULES) $Byynull.a
$Bbbexit.asm:   $Bbbexit.a;   $(OPTIM) -o$@  $(RULES) $Bbbexit.a

$Bassert.a:   lib/assert.c;   $(CC) -o $@ lib/assert.c
$Byynull.a:   lib/yynull.c;   $(CC) -o $@ lib/yynull.c
$Bbbexit.a:   lib/bbexit.c;   $(CC) -o $@ lib/bbexit.c

$Blburg$E:    $Blburg$O $Bgram$O;    $(LD) $(LDFLAGS) -Fe$@ $Blburg$O $Bgram$O

$Blburg$O $Bgram$O:     lburg/lburg.h

$Blburg$O:     lburg/lburg.c;     $(CL) $(CFLAGS) -c -Ilburg -Fo$@ lburg/lburg.c
$Bgram$O:   lburg/gram.c;    $(CL) $(CFLAGS) -c -Ilburg -Fo$@ lburg/gram.c

CPPOBJS=$Bcpp$O $Blexer$O $Bnlist$O $Btokens$O $Bmacro$O $Beval$O \
  $Binclude$O $Bhideset$O $Bgetopt$O $Bunix$O

$Bcpp$E:    $(CPPOBJS)
     $(LD) $(LDFLAGS) -Fe$@ $(CPPOBJS) -link $(LOPT)

$(CPPOBJS):   cpp/cpp.h

$Bcpp$O:    $Bcpp.asm;    $(ML) $(MFLAGS) /Fo$@ $Bcpp.asm
$Blexer$O:    $Blexer.asm;     $(ML) $(MFLAGS) /Fo$@ $Blexer.asm
$Bnlist$O:    $Bnlist.asm;     $(ML) $(MFLAGS) /Fo$@ $Bnlist.asm
$Btokens$O:   $Btokens.asm;    $(ML) $(MFLAGS) /Fo$@ $Btokens.asm
$Bmacro$O:    $Bmacro.asm;     $(ML) $(MFLAGS) /Fo$@ $Bmacro.asm
$Beval$O:   $Beval.asm;    $(ML) $(MFLAGS) /Fo$@ $Beval.asm
$Binclude$O:     $Binclude.asm;      $(ML) $(MFLAGS) /Fo$@ $Binclude.asm
$Bhideset$O:     $Bhideset.asm;      $(ML) $(MFLAGS) /Fo$@ $Bhideset.asm
$Bgetopt$O:   $Bgetopt.asm;    $(ML) $(MFLAGS) /Fo$@ $Bgetopt.asm
$Bunix$O:   $Bunix.asm;    $(ML) $(MFLAGS) /Fo$@ $Bunix.asm

$Bcpp.asm:    $Bcpp.a;    $(OPTIM) -o$@  $(RULES) $Bcpp.a
$Blexer.asm:    $Blexer.a;     $(OPTIM) -o$@  $(RULES) $Blexer.a
$Bnlist.asm:    $Bnlist.a;     $(OPTIM) -o$@  $(RULES) $Bnlist.a
$Btokens.asm:   $Btokens.a;    $(OPTIM) -o$@  $(RULES) $Btokens.a
$Bmacro.asm:    $Bmacro.a;     $(OPTIM) -o$@  $(RULES) $Bmacro.a
$Beval.asm:   $Beval.a;    $(OPTIM) -o$@  $(RULES) $Beval.a
$Binclude.asm:     $Binclude.a;     $(OPTIM) -o$@  $(RULES) $Binclude.a
$Bhideset.asm:     $Bhideset.a;     $(OPTIM) -o$@  $(RULES) $Bhideset.a
$Bgetopt.asm:   $Bgetopt.a;    $(OPTIM) -o$@  $(RULES) $Bgetopt.a
$Bunix.asm:   $Bunix.a;    $(OPTIM) -o$@  $(RULES) $Bunix.a

$Bcpp.a:    cpp/cpp.c;    $(CC) -Icpp -o $@ cpp/cpp.c
$Blexer.a:    cpp/lex.c;    $(CC) -Icpp -o $@ cpp/lex.c
$Bnlist.a:    cpp/nlist.c;     $(CC) -Icpp -o $@ cpp/nlist.c
$Btokens.a:   cpp/tokens.c;    $(CC) -Icpp -o $@ cpp/tokens.c
$Bmacro.a:    cpp/macro.c;     $(CC) -Icpp -o $@ cpp/macro.c
$Beval.a:   cpp/eval.c;    $(CC) -Icpp -o $@ cpp/eval.c
$Binclude.a:     cpp/include.c;      $(CC) -Icpp -o $@ cpp/include.c
$Bhideset.a:     cpp/hideset.c;      $(CC) -Icpp -o $@ cpp/hideset.c
$Bgetopt.a:   cpp/getopt.c;    $(CC) -Icpp -o $@ cpp/getopt.c
$Bunix.a:   cpp/unix.c;    $(CC) -Icpp -o $@ cpp/unix.c

test:  $T8q.s \
  $Tarray.s \
  $Tcf.s \
  $Tcq.s \
  $Tcvt.s \
  $Tfields.s \
  $Tfront.s \
  $Tincr.s \
  $Tinit.s \
  $Tlimits.s \
```

```
  $Tparanoia.s \
  $Tsort.s \
  $Tspill.s \
  $Tstdarg.s \
  $Tstruct.s \
  $Tswitch.s \
  $Twf1.s \
  $Tyacc.s

$T8q.s:  tst\8q.c tst\8q.0 all
  -$C -S -Wf-errout=$T8q.2 -o $T8q.s tst/8q.c
  fc $(TARGET)\tst\8q.sbk $T8q.s
  fc $(TARGET)\tst\8q.2bk $T8q.2
  $C -o $T8q$E $T8q.s
  -$T8q$E <tst/8q.0 >$T8q.1
  fc $(TARGET)\tst\8q.1bk $T8q.1
$Tarray.s:    tst\array.c tst\array.0 all
  -$C -S -Wf-errout=$Tarray.2 -o $Tarray.s tst/array.c
  fc $(TARGET)\tst\array.sbk $Tarray.s
  fc $(TARGET)\tst\array.2bk $Tarray.2
  $C -o $Tarray$E $Tarray.s
  -$Tarray$E <tst/array.0 >$Tarray.1
  fc $(TARGET)\tst\array.1bk $Tarray.1
$Tcf.s:  tst\cf.c tst\cf.0 all
  -$C -S -Wf-errout=$Tcf.2 -o $Tcf.s tst/cf.c
  fc $(TARGET)\tst\cf.sbk $Tcf.s
  fc $(TARGET)\tst\cf.2bk $Tcf.2
  $C -o $Tcf$E $Tcf.s
  -$Tcf$E <tst/cf.0 >$Tcf.1
  fc $(TARGET)\tst\cf.1bk $Tcf.1
$Tcq.s:  tst\cq.c tst\cq.0 all
  -$C -S -Wf-errout=$Tcq.2 -o $Tcq.s tst/cq.c
  fc $(TARGET)\tst\cq.sbk $Tcq.s
  fc $(TARGET)\tst\cq.2bk $Tcq.2
  $C -o $Tcq$E $Tcq.s
  -$Tcq$E <tst/cq.0 >$Tcq.1
  fc $(TARGET)\tst\cq.1bk $Tcq.1
$Tcvt.s:    tst\cvt.c tst\cvt.0 all
  -$C -S -Wf-errout=$Tcvt.2 -o $Tcvt.s tst/cvt.c
  fc $(TARGET)\tst\cvt.sbk $Tcvt.s
  fc $(TARGET)\tst\cvt.2bk $Tcvt.2
  $C -o $Tcvt$E $Tcvt.s
  -$Tcvt$E <tst/cvt.0 >$Tcvt.1
  fc $(TARGET)\tst\cvt.1bk $Tcvt.1
$Tfields.s:   tst\fields.c tst\fields.0 all
  -$C -S -Wf-errout=$Tfields.2 -o $Tfields.s tst/fields.c
  fc $(TARGET)\tst\fields.sbk $Tfields.s
  fc $(TARGET)\tst\fields.2bk $Tfields.2
  $C -o $Tfields$E $Tfields.s
  -$Tfields$E <tst/fields.0 >$Tfields.1
  fc $(TARGET)\tst\fields.1bk $Tfields.1
$Tfront.s:    tst\front.c tst\front.0 all
  -$C -S -Wf-errout=$Tfront.2 -o $Tfront.s tst/front.c
  fc $(TARGET)\tst\front.sbk $Tfront.s
  fc $(TARGET)\tst\front.2bk $Tfront.2
$Tincr.s:  tst\incr.c tst\incr.0 all
  -$C -S -Wf-errout=$Tincr.2 -o $Tincr.s tst/incr.c
  fc $(TARGET)\tst\incr.sbk $Tincr.s
  fc $(TARGET)\tst\incr.2bk $Tincr.2
  $C -o $Tincr$E $Tincr.s
  -$Tincr$E <tst/incr.0 >$Tincr.1
  fc $(TARGET)\tst\incr.1bk $Tincr.1
$Tinit.s:  tst\init.c tst\init.0 all
  -$C -S -Wf-errout=$Tinit.2 -o $Tinit.s tst/init.c
  fc $(TARGET)\tst\init.sbk $Tinit.s
  fc $(TARGET)\tst\init.2bk $Tinit.2
  $C -o $Tinit$E $Tinit.s
  -$Tinit$E <tst/init.0 >$Tinit.1
  fc $(TARGET)\tst\init.1bk $Tinit.1
$Tlimits.s:   tst\limits.c tst\limits.0 all
  -$C -S -Wf-errout=$Tlimits.2 -o $Tlimits.s tst/limits.c
  fc $(TARGET)\tst\limits.sbk $Tlimits.s
  fc $(TARGET)\tst\limits.2bk $Tlimits.2
  $C -o $Tlimits$E $Tlimits.s
  -$Tlimits$E <tst/limits.0 >$Tlimits.1
  fc $(TARGET)\tst\limits.1bk $Tlimits.1
$Tparanoia.s:   tst\paranoia.c tst\paranoia.0 all
  -$C -S -Wf-errout=$Tparanoia.2 -o $Tparanoia.s tst/paranoia.c
  fc $(TARGET)\tst\paranoia.sbk $Tparanoia.s
  fc $(TARGET)\tst\paranoia.2bk $Tparanoia.2
  $C -o $Tparanoia$E $Tparanoia.s
  -$Tparanoia$E <tst/paranoia.0 >$Tparanoia.1
  fc $(TARGET)\tst\paranoia.1bk $Tparanoia.1
$Tsort.s:  tst\sort.c tst\sort.0 all
  -$C -S -Wf-errout=$Tsort.2 -o $Tsort.s tst/sort.c
  fc $(TARGET)\tst\sort.sbk $Tsort.s
  fc $(TARGET)\tst\sort.2bk $Tsort.2
  $C -o $Tsort$E $Tsort.s
  -$Tsort$E <tst/sort.0 >$Tsort.1
  fc $(TARGET)\tst\sort.1bk $Tsort.1
$Tspill.s:    tst\spill.c tst\spill.0 all
  -$C -S -Wf-errout=$Tspill.2 -o $Tspill.s tst/spill.c
  fc $(TARGET)\tst\spill.sbk $Tspill.s
  fc $(TARGET)\tst\spill.2bk $Tspill.2
  $C -o $Tspill$E $Tspill.s
  -$Tspill$E <tst/spill.0 >$Tspill.1
  fc $(TARGET)\tst\spill.1bk $Tspill.1
$Tstdarg.s:   tst\stdarg.c tst\stdarg.0 all
```

```
  -$C -S -Wf-errout=$Tstdarg.2 -o $Tstdarg.s tst/stdarg.c
  fc $(TARGET)\tst\stdarg.sbk $Tstdarg.s
  fc $(TARGET)\tst\stdarg.2bk $Tstdarg.2
  $C -o $Tstdarg$E $Tstdarg.s
  -$Tstdarg$E <tst/stdarg.0 >$Tstdarg.1
  fc $(TARGET)\tst\stdarg.1bk $Tstdarg.1
$Tstruct.s:   tst\struct.c tst\struct.0 all
  -$C -S -Wf-errout=$Tstruct.2 -o $Tstruct.s tst/struct.c
  fc $(TARGET)\tst\struct.sbk $Tstruct.s
  fc $(TARGET)\tst\struct.2bk $Tstruct.2
  $C -o $Tstruct$E $Tstruct.s
  -$Tstruct$E <tst/struct.0 >$Tstruct.1
  fc $(TARGET)\tst\struct.1bk $Tstruct.1
$Tswitch.s:   tst\switch.c tst\switch.0 all
  -$C -S -Wf-errout=$Tswitch.2 -o $Tswitch.s tst/switch.c
  fc $(TARGET)\tst\switch.sbk $Tswitch.s
  fc $(TARGET)\tst\switch.2bk $Tswitch.2
  $C -o $Tswitch$E $Tswitch.s
  -$Tswitch$E <tst/switch.0 >$Tswitch.1
  fc $(TARGET)\tst\switch.1bk $Tswitch.1
$Twf1.s:    tst\wf1.c tst\wf1.0 all
  -$C -S -Wf-errout=$Twf1.2 -o $Twf1.s tst/wf1.c
  fc $(TARGET)\tst\wf1.sbk $Twf1.s
  fc $(TARGET)\tst\wf1.2bk $Twf1.2
  $C -o $Twf1$E $Twf1.s
  -$Twf1$E <tst/wf1.0 >$Twf1.1
  fc $(TARGET)\tst\wf1.1bk $Twf1.1
$Tyacc.s:   tst\yacc.c tst\yacc.0 all
  -$C -S -Wf-errout=$Tyacc.2 -o $Tyacc.s tst/yacc.c
  fc $(TARGET)\tst\yacc.sbk $Tyacc.s
  fc $(TARGET)\tst\yacc.2bk $Tyacc.2
  $C -o $Tyacc$E $Tyacc.s
  -$Tyacc$E <tst/yacc.0 >$Tyacc.1
  fc $(TARGET)\tst\yacc.1bk $Tyacc.1

testclean:
  -del /q $T8q$E $T8q.s $T8q.2 $T8q.1
  -del /q $Tarray$E $Tarray.s $Tarray.2 $Tarray.1
  -del /q $Tcf$E $Tcf.s $Tcf.2 $Tcf.1
  -del /q $Tcq$E $Tcq.s $Tcq.2 $Tcq.1
  -del /q $Tcvt$E $Tcvt.s $Tcvt.2 $Tcvt.1
  -del /q $Tfields$E $Tfields.s $Tfields.2 $Tfields.1
  -del /q $Tfront$E $Tfront.s $Tfront.2 $Tfront.1
  -del /q $Tincr$E $Tincr.s $Tincr.2 $Tincr.1
  -del /q $Tinit$E $Tinit.s $Tinit.2 $Tinit.1
  -del /q $Tlimits$E $Tlimits.s $Tlimits.2 $Tlimits.1
  -del /q $Tparanoia$E $Tparanoia.s $Tparanoia.2 $Tparanoia.1
  -del /q $Tsort$E $Tsort.s $Tsort.2 $Tsort.1
  -del /q $Tspill$E $Tspill.s $Tspill.2 $Tspill.1
  -del /q $Tstdarg$E $Tstdarg.s $Tstdarg.2 $Tstdarg.1
  -del /q $Tstruct$E $Tstruct.s $Tstruct.2 $Tstruct.1
  -del /q $Tswitch$E $Tswitch.s $Tswitch.2 $Tswitch.1
  -del /q $Twf1$E $Twf1.s $Twf1.2 $Twf1.1
  -del /q $Tyacc$E $Tyacc.s $Tyacc.2 $Tyacc.1

clean::     testclean
     -del /q $B*$O
               -del /q $B*.a
               -del /q $B*.asm
     -del /q $Bdagcheck.c $Balpha.c $Bmips.c $Bx86.c $Bsparc.c $Bx86linux.c
     -del /q $Brcc1$E $Brcc1$E $B1rcc$E $B2rcc$E
     -del /q $B*.ilk

clobber::   clean
     -del /q $Brcc$E $Blburg$E $Bcpp$E $Blcc$E $Bcp$E $Bbprint$E $Bops$E $B*$A
     -del /q $B*.pdb $B*.pch

RCCSRCS=src/alloc.c \
  src/bind.c \
  src/dag.c \
  src/decl.c \
  src/enode.c \
  src/error.c \
  src/expr.c \
  src/event.c \
  src/init.c \
  src/inits.c \
  src/input.c \
  src/lex.c \
  src/list.c \
  src/main.c \
  src/output.c \
  src/prof.c \
  src/profio.c \
  src/simp.c \
  src/stmt.c \
  src/string.c \
  src/sym.c \
  src/trace.c \
  src/tree.c \
  src/types.c \
  src/null.c \
  src/symbolic.c \
  src/bytecode.c \
  src/gen.c \
  src/stab.c \
  $Bdagcheck.c \
  $Balpha.c \
```

```
  $Bmips.c \
  $Bsparc.c \
  $Bx86linux.c \
  $Bx86.c

C=$Blcc -A -d0.6 -Wo-lccdir=$(BUILDDIR) -Isrc -I$(BUILDDIR)
triple:  $B2rcc$E
  -fc /b $B1rcc$E $B2rcc$E

$B1rcc$E:   $Brcc$E $Blcc$E $Bcpp$E
     $C -o $@ -B$B $(RCCSRCS)
$B2rcc$E:   $B1rcc$E
     $C -o $@ -B$B1 $(RCCSRCS)
```