Um Compilador para uma Linguagem de Programação Orientada a Objetos

Carlos Alberto Furuti

DCC - IMECC
UNICAMP
1991

Um Compilador para uma Linguagem de Programação Orientada a Objetos

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Carlos Alberto Furuti e aprovada pela Comissão Julgadora.

Campinas, 26 de agosto de 1991

Prof. Dr. Rogério Drummond B. P. de Mello Filho

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para obtenção do Título de Mestre em Ciência da Computação

Um Compilador para uma Linguagem de Programação Orientada a Objetos

Carlos Alberto Furuti
Projeto A_HAND

Unicamp, 1991

Os nomes a seguir são marcas registradas dos respectivos proprietários:

Ada — Ada Joint Program Office (U.S. Government, Department of Defense)
CLIX, Intergraph — Intergraph Corporation
Eiffel — Interactive Software Engineering, Inc.
IBM — International Business Machines Corporation
Microsoft, MS-DOS — Microsoft Corporation
Objective C — Productivity Products International
Smalltalk — Xerox Corporation
SunOS — Sun Microsystems, Inc.
UNIX — AT&T; *UNIX* System Laboratories, Inc.

A meus pais

Agradecimentos

Expresso minha gratidão a todos que de uma forma ou outra estiveram comigo durante o Mestrado. Em especial, ao Prof. Rogério Drummond, meu orientador, e aos membros da Comissão Julgadora, Prof. Sérgio E. R. de Carvalho, Prof. Tomasz Kowaltowski e Prof. Hans K. E. Liesenberg.

À Unicamp e às instituições de amparo à pesquisa, CNPq, CAPES e FAPESP, que sucessivamente contribuíram com bolsas de estudo.

A meus colegas na Unicamp e a Fábio Queda, que primeiro trabalhou em Cm.

Quanto aos companheiros do Projeto A.HAND, certamente cometeria injustiças se citasse nomes; mas injustiça maior seria não mencionar quem mais de perto acompanhou meu trabalho: Ana C. Biazetti, André F. Vincent, Carlos A. Polanczyk, Cassius Di Cianni, Hernán P. Arias, Lídia A. R. Yamamoto, e Luiz C. D. Carneiro.

Last but not least, àqueles que já têm a dedicatória desta Dissertação, mas não podem ficar sem agradecimentos.

Conteúdo

Prólogo

Construir programas complexos requer uma linguagem de programação e ferramentas apropriadas capazes de explorar abstração, modularidade e o esforço cooperativo de várias pessoas. O Projeto A_HAND, no Departamento de Ciência da Computação da Unicamp, procura definir um ambiente integrado para criação de programas, da definição à implementação e manutenção, tendo patrocinado o desenvolvimento de Cm como a linguagem básica de programação.

Descreve-se aqui a implementação do primeiro compilador para Cm, uma extensão da linguagem de programação C oferecendo suporte à modularidade e polimorfismo paramétrico. Cm introduz um construtor *classe* para um tipo de dados abstrato com parâmetros que podem ser expressões de tipo. Portanto, uma classe pode definir um grande número de novos tipos através de diferentes parametrizações. A linguagem também oferece verificação rigorosa (estrutural) de tipos em tempo de compilação e herança múltipla de classes.

O compilador para Cm é implementado através de pré-processamento, traduzindo código Cm em código-fonte aceito por um compilador C convencional. O código gerado é portável, possibilitando a imediata programação Cm numa variedade de ambientes. Vários problemas a resolver numa tradução para outra linguagem são interessantes e diferentes dos encontrados na tradução direta para código de máquina ou objeto.

O tradutor inclui a compilação automática de um conjunto mínimo de classes a atualizar após edição, bem como a chamada ao compilador C, simplificando a organização de projetos e a manutenção de consistência de programas.

A versão implementada do compilador, escrita inicialmente nos sistemas operacionais *UNIX* e MS-DOS, foi publicada e usada em projetos de curso a nível de Mestrado na Unicamp.

Este texto tem como objetivo apresentar os princípios de programação Cm, servindo como uma introdução ao usuário, bem como discutir aspectos da construção do compilador.

Capítulo 1

Introdução

'Where shall I begin, please your Majesty?' he asked.
'Begin at the beginning,' the King said gravely, 'and go on till you come to the end: then stop.'
L. Carroll, *Alice's Adventures in Wonderland*

1.1 Programação em Grande Escala

A produção de programas de grande porte apresenta desafios especiais. Sistemas de computação comumente oferecem um nível elevado de complexidade, exigindo o uso de ferramentas apropriadas, sem as quais seu desenvolvimento se tornaria dispendioso ou virtualmente inexeqüível. Isso ocorre porque o esforço despendido na resolução de um problema aumenta mais que linearmente com seu tamanho. A complexidade dos atuais sistemas de grande porte exige que sejam divididos em módulos a serem projetados separadamente, se possível por grupos de trabalho individualizados, e depois integrados numa única ferramenta. Embora essa metodologia permita progredir paralelamente em vários tópicos do problema, a própria divisão das tarefas introduz questões adicionais, desde um particionamento ótimo (de modo que nenhum dos grupos termine sua parte muito antes ou depois dos outros), até a integração final, passando pela sincronização e cooperação necessárias no decorrer do projeto. Apesar das diversas propostas já apresentadas para amenizar a situação, certas circunstâncias ainda podem tornar o problema quase insolúvel [Br 82].

Às dificuldades mencionadas quando se trata de grandes projetos, o programador de sistemas pode acrescentar outras conseqüentes de sua especialidade. Neste tipo de projeto, destinado a fornecer suporte ao *software* de aplicação, são estudadas entidades de nível relativamente mais baixo, como sistemas operaci-

onais e controladores de dispositivos,[1] por vezes exigindo-se o conhecimento da arquitetura do sistema e/ou do próprio *hardware*. Tradicionalmente este requisito é conflitante com outras características importantes em *softwares* modernos: portabilidade e facilidade de manutenção. Ambas pressupõem o maior distanciamento (abstração) possível dos dispositivos físicos, deixando o projetista se concentrar apenas nos aspectos relevantes do problema (i.e., independentes do *hardware*).

1.2 Linguagens e Ambientes

Ao lado do ambiente de desenvolvimento oferecido ao projetista, a linguagem de programação exerce a maior influência sobre a produtividade e a eficiência do processo de desenvolvimento. As qualidades desejáveis de ambos devem ser pesadas em função das exigências da programação em grande escala:

- Grandes programas devem seguir uma organização modular e hierárquica, sendo desenvolvidos por equipes relativamente autônomas.
- O produto final deve ser flexível, i.e., é necessário que se preste facilmente a extensões e atualizações, inclusive possibilitando seu transporte para ambientes ou equipamentos que não aquele originalmente previsto em seu desenvolvimento. Essas atualizações quase sempre incluem a correção de erros porventura cometidos.
- O produto deve ser confiável, apesar dos erros às vezes inevitáveis com que é entregue (grandes projetos dificilmente têm todos os seus erros corrigidos).
- O tempo de desenvolvimento deve ser minimizado — este critério, que representa grande parte da competitividade de um produtor de *software*, é altamente conflitante com os anteriores. Por outro lado, verifica-se que boa parte do esforço despendido na indústria de programação é replicado, tanto por diferentes equipes de trabalho como por um mesmo programador em diversos momentos: há um evidente desperdício de recursos.
- Finalmente, deve ser facilitada a geração de protótipos do produto final, por diversos motivos. Primeiramente, pode ser interessante, no caso de um problema novo, ganhar experiência sobre suas características produzindo uma versão reduzida da solução. Mesmo que não apresente todos os detalhes de um verdadeiro sistema, o protótipo permite experimentar sua viabilidade e funcionalidade. Além disso, capacita a gerência do projeto a estudar as dificuldades encontradas na sua implementação e melhor

[1] *device drivers*, em inglês

distribuir os esforços no produto real. O protótipo facilita o planejamento evitando detalhes irrelevantes. Por outro lado, nem sempre a especificação do problema é completa: versões preliminares testam a opinião do cliente e admitem um refinamento das especificações, antes do início efetivo do desenvolvimento.

Em linhas gerais, são requeridos ambientes de programação e linguagens em cujos projetos foram levados em consideração os itens anteriormente citados.

O incentivo à divisão de programas em blocos é imprescindível; apenas isso, entretanto, não é suficiente. Tal divisão deve ser executada de forma lógica, ordenada. Um sistema pode ser representado por um grafo (rede), cujos nós representam módulos, e as arestas, as relações de transferência de dados ou controle entre os nós. Uma rede em que cada nó se liga a muitos outros é quase tão pouco gerenciável quanto um sistema monolítico. Blocos de processamento devem ser organizados com inter-relacionamento reduzido — embora este ponto dependa mais do programador que da linguagem, esta deve estimular a construção de módulos com interfaces (especificações) bem definidas. Além disso, a maior quantidade possível de informação acerca do funcionamento interno de cada unidade deve ser oculta das demais, encapsulando detalhes que, se expostos, poderiam ocasionar conflitos com outros módulos, ou ainda, reduzir a independência do conjunto ao demandar maior coordenação. Por outro lado, das informações que permanecem visíveis, deve-se extrair o necessário para verificar sua consistência com a implementação real e com o uso que se faz do módulo. Podemos observar um progresso de linguagens de programação estruturadas simples, como Pascal "puro" [JW 75] (um programa completo num módulo único) para Modula-2 [Ch 86] e Ada [Yo 83] [An 83] (diversos blocos, verificação de consistência, compilação separada), passando pela linguagem C [KR 78] (diversos blocos, compilação independente, pouca verificação).

Flexibilidade pode ser decorrência natural do projeto modular de um programa. Obviamente uma construção por partes independentes encoraja a produção de diversos blocos facilmente intercambiáveis, de sorte que alterações, em certos casos, se resumem a simples recombinações. Uma linguagem de programação adequada deve separar a definição da interface de um módulo de sua implementação (explicitamente ou não). Ou seja, exige-se suporte a *tipos abstratos de dados*. Com base nisso, o ambiente de desenvolvimento estaria apto a, após cada alteração, realizar automaticamente o mínimo necessário de processamento (compilação) para deixar o sistema num estado coerente. Mais ainda, dada uma coleção bem organizada de blocos, preparar um protótipo consiste principalmente em selecionar e reunir alguns deles, juntamente com algum código feito "sob medida" para a aplicação.

Espera-se que uma linguagem destinada à programação em grande escala apresente construções flexíveis mas com legibilidade, de forma tanto a evitar

erros por parte do programador como para facilitar a manutenção de programas. Os tipos de dados devem permitir uma verificação rigorosa, preferencialmente durante a compilação, o que reduz substancialmente a probabilidade de ocorrência de erros.

Após o desenvolvimento de um certo número de subsistemas, observa-se a repetição de diversos problemas, de forma mais ou menos semelhante; naturalmente se busca nestes casos aplicar as mesmas soluções já implementadas. Embora novamente aqui a modularização facilite adaptações, torna-se aconselhável, já durante a concepção dos módulos, prever uma utilização futura mais geral possível.

Tal como no caso de sub-rotinas, isso pode ser obtido deixando a maior parte dos elementos que definem um componente em aberto, para serem definidos apenas pela entidade que invoca seus serviços; i.e., criando módulos parametrizáveis. Pode-se considerar essas entidades como operadores polimórficos [Mi 78] sobre algum tipo (abstrato) de dados. Quanto ao ambiente de desenvolvimento, este precisa oferecer mecanismos para armazenamento e recuperação de componentes mediante sua especificação (possivelmente descrição funcional).

É importante a capacidade de re-utilizar módulos de *software* de forma análoga como se recombinam circuitos integrados. Para tanto é essencial a *independência de informação* no projeto de módulos facilmente re-aproveitáveis. Tipos abstratos de dados contribuem para essa capacidade mas não são suficientes [Me 87]. Considera-se essencial também *herança*, i.e., a possibilidade de uma entidade (uma *especialização*) herdar atributos de outra(s), alterando alguns deles; é uma característica de projetos orientados a objetos [Re 82].

Em resumo, prevenção de erros, projeto modular e incentivo à re-utilização são os principais fatores a serem considerados na avaliação da produtividade de linguagens e ambientes de programação.

1.3 Soluções e Alternativas

Não serão mais tratados aspectos de ambientes de programação, mas sim os aspectos mais significativos de algumas linguagens de amplo uso na programação em grande escala.

1.3.1 C: um Compromisso

Na atualidade, C é possivelmente a mais popular linguagem de programação de sistemas. C é geralmente associada a portabilidade e eficiência; dispõe de compiladores para uma enorme variedade de máquinas e sistemas operacionais; mas também é uma linguagem lembrada por ser fonte de programas ininteligíveis e

erros sutilmente ocultos [FG 82]. C foi imaginada desde o início como uma linguagem "para programadores" criada "por programadores", ou seja, *de e para* profissionais experientes. Procurava-se com ela uma linguagem extremamente concisa e de implementação eficiente. Ao mesmo tempo, um dos primeiros grandes projetos a usá-la foi uma das versões originais do sistema operacional *UNIX* [KP 84], na época em um computador de relativamente pequeno porte. Esse fato e mais suas ancestrais mais próximas explicam sua simplicidade e a inclusão de alguns (poucos) aspectos mais típicos de linguagens de montagem. Não deve, entretanto, ser qualificada como "de baixo nível".

O tratamento de muitos aspectos que, em outras linguagens, estariam inclusos em sua própria definição, em C é deixado a cargo do sistema operacional ou sub-rotinas de suporte. O conjunto de operações de entrada/saída, e.g., não é oficialmente parte da linguagem, que somente se preocupa em definir dados e mecanismos de controle. Na prática, invocações ao sistema e funções provenientes de bibliotecas gerenciam todos os demais serviços.

A presença de um pré-processador, a princípio independente[2] e ortogonal ao compilador e atualmente rigorosamente definido, adiciona flexibilidade. Ele gerencia inclusão de arquivos-fonte, implementa compilação condicional e permite substituições via macros. Muito do poder da linguagem é emprestado pela definição de pseudo-funções e construções como *assert()* e *va_arg* [At 89].

Deixar muito da linguagem a cargo de funções fornece motivação para estender suas construções apenas implementando novas bibliotecas. Por exemplo, desvios de controle para fora de sub-rotinas, tratamento (particular) de interrupções e concorrência limitada já foram introduzidos sem alterações na linguagem.

Infelizmente, as próprias características favoráveis de C podem ocasionar dificuldades ao programador. A verificação de tipos é especialmente permissiva, particularmente quanto a parâmetros e valores de retorno de funções (novos compiladores seguindo o padrão ANSI[3] resolvem apenas em parte esse problema). Esse legado de seus ancestrais torna possíveis programas muito versáteis mas bastante suscetíveis a erros do programador.

A excessiva liberdade propiciada pelo pré-processador cria margem a construções extremamente irregulares – por exemplo, a tarefa de gerar índices de referências cruzadas de funções, trivial para programas em Pascal, exige técnicas de compilação e macro-processamento para sua perfeita consecução em C. Outras aplicações dirigidas por sintaxe, como editores especializados, também podem encontrar dificuldades.

Um outro aspecto, a portabilidade de programas escritos em C, é correto apenas em parte. Embora o conjunto de operadores e comandos seja realmente

[2] uma espécie de *filtro* desvinculado da linguagem

[3] comitê ANSI X3J11; padrão ANSI X3.159–1989, ISO/IEC 9899

padrão de um compilador a outro, as bibliotecas de funções pré-definidas — que permitem que um programa se comunique com as entidades externas — variam de implementação para implementação. Caminha-se para uma padronização da linguagem, com a definição de um conjunto mínimo de funções (baseado primariamente nas rotinas de biblioteca encontradas em *UNIX*).

1.3.2 Programação Orientada a Objetos

Programação orientada a objetos parece destinar-se a ser o paradigma de programação desta década. Após um início restrito a nichos especializados, como a linguagem Simula [Ho 84], e um contínuo mas quase recluso desenvolvimento, é agora reconhecida como a metodologia mais apropriada ao desenvolvimento de programas em grande escala, determinando a criação de novas técnicas e linguagens de programação.

Não é fácil ou imediato definir programação a objetos, e as definições encontradas na literatura diferem entre si. Talvez seja mais apropriado ou efetivamente descritivo caracterizar os aspectos fundamentais de sistemas considerados "por objetos", e daí tirar conclusões. De qualquer forma, é a "construção de *software* como coleções estruturadas de implementações de tipos abstratos de dados" [Me 87], ou a aplicação dos conceitos de abstração de dados, encapsulamento, herança e polimorfismo.

Tipos Abstratos de Dados e Encapsulamento

Projetar tipos de dados de forma abstrata significa descrevê-los funcionalmente, ou seja, indicar seu propósito e como podem ser usados, ao invés de relacionar sua estrutura interna. Caso apenas esses aspectos sejam apresentados a entidades externas (isto é, apenas as funções e dados necessários para requerer serviços daquele tipo), observa-se encapsulamento. Essas duas características conferem alto grau de flexibilidade ao projeto de programas, tornando fácil reaproveitar o trabalho anterior sem alterações profundas. Além disso, reduz a dependência entre módulos e a possibilidade de erros de projeto e implementação.

Em programação orientada a objetos, um tipo abstrato de dados definindo dados e operações é chamado *classe*. Classes são consideradas construtores de tipos. Elementos cujo tipo é uma classe são *objetos*, e são *instâncias* (exemplos) daquela classe; as operações que podem ser executadas sobre eles são conhecidas também como *métodos* e em algumas linguagens são executadas em resposta a *mensagens* [Go 83].

Herança

Herança é possivelmente a característica mais proeminente e fundamental em programação orientada a objetos. Ela consiste em definir novos conceitos (tipos de dados) em termos de outros já conhecidos, adicionando ou modificando apenas os detalhes (dados, operadores e funções) que realmente distingüem um conceito de outro. Essa capacidade pode reduzir substancialmente o esforço de desenvolvimento e depuração de programas: novas extensões são adicionadas sem alterar ou conhecer código já existente.

O tema da herança admite variações, algumas ainda no terreno da experiência. A maior parte das linguagens com orientação a objetos admite herança apenas por um ancestral, distingüindo uma hierarquia em forma de árvore. Caso os atributos de mais de um ancestral possam ser herdados, ocorre *herança múltipla*, e a hierarquia passa a um grafo direcionado. A questão imediata que restringe a disseminação desse potencial é a resolução de conflitos quando dois atributos distintos coincidem em nome. Em certos casos é possível simplesmente rebatizar uma das alternativas, resolvendo a ambigüidade na própria entidade herdeira. Isso ocorre em Eiffel [MNM87]. A mudança de nomes, longe se ser apenas um artifício para resolver conflitos, pode realmente adicionar potencial a uma linguagem, descrevendo dados e operadores em mais de uma maneira.

Herança repetida consiste na presença da mesma entidade em duas linhas de herança em diferentes níveis.

Herança seletiva implica em herdar apenas componentes selecionados, e não toda uma entidade.

Herança negativa é relacionada à seletiva, explicitando desta vez os componentes que não serão herdados.

Polimorfismo e Ligação

A introdução de herança favorece o reaproveitamento de programas numa variedade de versões. As técnicas de abstração e encapsulamento de dados, por outro lado, não poderiam ser aplicadas sem o uso de algum tipo de *polimorfismo* [Mi 78][CW 85]. Esse conceito implica no uso de um mesmo nome numa variedade de formas com efeitos diversos. Por exemplo, o operador '+' pode ser aplicado a dois números inteiros, significando adição aritmética; a um único número, como sinal positivo; a dois complexos, como adição complexa; ou a duas cadeias de caráteres, representando concatenação; ou ainda a uma árvore e um item, indicando inserção. Em outras palavras, num ambiente polimórfico, duas entidades reagem de diferentes formas ao mesmo operador ou função, dependendo de seu tipo.

Polimorfismo está associado ao conceito de *ligação* (*binding*) de objetos, que é a determinação do significado de nomes e operadores. Em C, a ligação é pu-

ramente *estática*, sendo completada durante a compilação. O extremo oposto é apresentado por Smalltalk [Go 83]: na ligação puramente *dinâmica*, apenas durante a execução de um programa é determinada que função corresponde a cada nome. O tipo de ligação para uma linguagem é intimamente ligado ao momento da verificação de tipos e à eficiência de sua implementação — *binding* dinâmico é bem mais flexível que estático, mas pode implicar em severa perda de desempenho, tanto em velocidade, para relacionar nomes à ações, como em espaço, pela dificuldade em pré-alocar memória. Além disso, a menos que restrições sejam adotadas, é impossível verificar com segurança todos os erros de tipo.

Apesar dos problemas potenciais, advoga-se ligação dinâmica como essencial a programação orientada a objetos: ela permite agrupar elementos de forma dinâmica, conhecendo-se apenas suas características de interface, compondo sistemas completamente reconfiguráveis durante a execução.

Programação Sistemática

Herança e tipos abstratos de dados implementam duas formas de abstração — respectivamente, especialização e agregação — e devem ser vistas como ortogonais. A primeira define conceitos usando suas *diferenças* em relação a outros, enquanto a segunda *compõe* diferentes conceitos para formar um novo. Um *avião* pode ser considerado como um *veículo que voa*; se um *veículo* tem *capacidade de carga*, *custo operacional* e *velocidade*, supõe-se que aviões herdem essas características de veículos, enquanto adicionam algumas, como *taxas aeroportuárias*, e modifiquem outras, como *consumo* (talvez de quilômetros por litro para galões por hora). Entretanto, *aviões* são compostos por um número variável de *asas*, *fuselagem* e *motores*, entre outros, o que propõe outra maneira de caracterizá-los — desta vez, em termos desses componentes. A abordagem da herança poderia interessar à contabilidade da aeronave, enquanto a abordagem da agregação seria mais importante para sua manutenção.

Abstração de dados é considerada essencial para a produção sistemática de programas. As técnicas de orientação a objetos incrementam a flexibilidade de tipos abstratos de dados, criando módulos de *software* facilmente extensíveis e reaproveitáveis.

Programação orientada a objetos enfatiza o projeto cuidadoso de tipos de dados em favor do projeto de algoritmos. O programador usando objetos deve encontrar à disposição um grande número de tipos abstratos definindo dados e o conjunto de operações legais aplicáveis sobre esses dados, representando da forma mais direta possível um conceito real do problema a resolver.

Apesar da idade e da disseminação do conceito de objetos, não há formalizações amplamente aceitas sobre como se deve usá-los ou se há regras formais para associar dados a objetos, ocasionando inúmeras discussões sobre exemplos

inadequados ou simplistas [Er 90].

Note-se que, embora haja linguagens ditas "orientadas a objetos", programação orientada a objetos é uma técnica de programação independente, que pode apenas ser suportada, com maior ou menor adequação, por esta ou aquela linguagem. Por exemplo, Ada provê grande flexibilidade na definição de tipos abstratos de dados, com diversos níveis de encapsulamento, mas não oferece mecanismos de herança. A linguagem C permite a definição de tipos abstratos com alguma disciplina de programação, mas o encapsulamento se restringe à declaração estática de variáveis e funções, não sendo, portanto, absolutamente seguro. Herança pode ser *simulada* em C, mas num escopo limitado, justificando a criação de variantes da linguagem acrescentando facilidades para o paradigma de objetos.

1.3.3 C++: a Evolução

C++ ("C + 1") é um superconjunto de C tornado público pela primeira vez em 1983. Desenvolvida por Bjarne Stroustrup nos laboratórios da AT&T, tem recebido grande atenção recentemente, com várias implementações difundidas em diferentes plataformas. É uma linguagem projetada para manter grande compatibilidade (fácil migração) com C, tendo inspirado, mais que recebido inspiração, o atual padrão ANSI C. Sua principal motivação é preservar os aspectos favoráveis de C — portabilidade, eficiência e flexibilidade — evitando suas deficiências mais flagrantes.

Classes e Encapsulamento

Em C++, *classes* são a princípio uma generalização de estruturas (*structs*) de C. Embora tanto classes como estruturas em C++ possam definir dados e funções como seus componentes, classes oferecem as maiores facilidades para abstração e encapsulamento.

Uma classe é dividida em duas partes definindo seus componentes *públicos* e *privados*. Comumente os dados são declarados na parte privada, preservando a estrutura interna da classe, enquanto que as funções que os manipulam (ou parte delas) são públicas. Tal como em C, uma função pode ser declarada em separado de sua definição (que contém o código propriamente dito), tornando as declarações de classe e estrutura sintaticamente semelhantes.

Enquanto os componentes privados somente são referenciáveis por outros da mesma classe, os públicos têm seu acesso liberado para entidades de outras classes, sob certas restrições. A cláusula **public** torna o componente visível de forma irrestrita, enquanto **protected** impede a visiblidade de outras classes que não a própria e outras dela derivadas (a derivação de classes será vista mais adiante).

Outras classes já existentes podem ser explicitamente declaradas como *friends*. Essas classes (e apenas elas) têm acesso aos dados e funções também declarados *friend*.

Para cada objeto cujo tipo é uma classe C++, as funções e dados não-privados podem ser referenciados como membros de uma estrutura. Os componentes do objeto são acessíveis à própria função através de um parâmetro apontador fictício (*this*).

Todas as funções de uma classe em C++ devem ter os tipos do valor de retorno e dos parâmetros declarados explicitamente, e a linguagem inclui uma verificação rigorosa de tipos na chamada de funções. Essas declarações são na verdade essenciais porque é possível definir diversas funções na mesma classe com o mesmo nome, diferindo apenas no tipo e/ou número de parâmetros. Dada essa possibilidade (chamada *function overloading*), o compilador C++ implementa uma forma de polimorfismo, usando o tipo dos parâmetros na chamada da função para determinar qual a versão adequada.

Uma facilidade importante para encapsulamento de dados é a extensão do conceito de *overloading* para operadores. A cláusula **operator** permite que uma função seja definida com o nome de um operador binário ou unário já existente. A partir daí, o operador pode ser aplicado a instâncias da classe como um operador pré-definido. Por restrições de análise léxica, não é possível criar novos operadores, nem alterar sua ordem de precedência.

Dois tipos de função, mesmo que vazios, sempre existem para qualquer classe C++. Um *construtor* é uma função executada implicitamente para criar qualquer nova instância de uma classe e comumente é usada para dar valores iniciais aos dados da instância, bem como obter e organizar qualquer memória dinâmica que seja necessária. Sendo comum que haja diferentes meios de definir esses valores, a função construtora pode ter várias versões, redefinidas com diferentes parâmetros. Um *destruidor*, em contrapartida, é uma função sem parâmetros, executada automaticamente quando um objeto é destruído (isto é, quando a execução do programa deixa o escopo em que ele é definido).

Herança

O prólogo da definição de uma classem C++ lista as classes herdadas (apenas a versão 2.0 da definição da linguagem introduziu herança múltipla [Co 87]).

Classes herdadas são chamadas *base*, e as herdeiras são conhecidas como *derivadas*. A estas aplicam-se as mesmas restrições de acesso a componentes públicos e privados já mencionadas, além da cláusula **protect** (numa classe básica).

C++, tal como em Simula, aplica a cláusula **virtual** para declarar funções que serão somente redefinidas em classe(s) derivada(s), permitindo que outras funções da mesma classe usem funções das classes herdeiras. Essa capacidade é

normalmente implementada com ligação dinâmica, mas a verificação de tipos é feita estaticamente.

1.3.4 Objective C: Ligação Dinâmica

Enquanto C++ procurou se manter um superconjunto, com quase total compatibilidade com sua linguagem de origem, *Objective C* [Co 87] é uma linguagem bem diferente. Ações num programa Objective C são normalmente *expressas* numa sintaxe muito semelhante à de mensagens, embora também possa ser usada uma notação mais convencional. A linguagem teve um início muito rápido, enfatizando ligação dinâmica e a criação de grandes bibliotecas de *módulos* pré-construídos (*software-ICs*, ou "circuitos integrados de *software*", na concepção do autor), mas não se disseminou como C++.

1.3.5 Cm: uma Extensão

A linguagem de programação Cm (C modular e polimórfico) [SLD 88] é superficialmente semelhante a C: as estruturas de controle e comandos básicos foram preservados de modo a evitar grandes problemas na aclimatação do grande número de usuários já proficientes em C. As linguagens começam a diferir de fato quando se estudam os métodos para definição e abstração de dados. Além de construir tipos numa sintaxe menos propensa a erros, C modular implementa dois métodos de abstração: herança (especialização) e tipos abstratos de dados (modularidade).

Todos os programas em Cm são compostos por classes, que são construtores de tipos. Cada classe descreve dados e funções que os manipulam, definindo também uma interface para outras classes. Objetos cujo tipo é definido por uma classe (suas *instâncias*) podem ser **importados** por objetos de outra classe; os dados e funções (componentes) declarados como *exportáveis* na classe importada são acessíveis aos objetos importadores.

Classes em Cm se assemelham aos *packages* de Ada e aos *modules* de Modula-2 (tal como nestas linguagens, os componentes de uma variável de tipo classe são referenciados prefixando-se o nome da variável). Além disso, assim como nos *generic packages*, cada classe é parametrizável, isto é, o tipo por ela especificado pode depender de parâmetros definidos pela sua instanciação. Por exemplo, uma classe descrevendo uma coleção de objetos com um limite máximo especificado pode ser definida deixando em aberto tanto aquele limite como o tipo dos objetos.

As instâncias de classes são objetos como outros quaisquer e passíveis de operações de atribuição e comparação (os operadores "=" e "==" sofrem *overloading*. Das estruturas de dados declaradas por uma classe, algumas podem ser compartilhadas entre todas as instanciações da mesma, enquanto outras serão definidas (no instante da execução) uma vez para cada objeto daquela classe.

O segundo mecanismo de abstração de dados, herança, possibilita que uma classe compartilhe e incorpore entidades de uma ou mais classes ancestrais. Como ocorre herança múltipla, aparece uma hierarquia descrita por um grafo orientado acíclico (não necessariamente uma árvore). Essa hierarquia é ortogonal à de classes importadas.

Outras características essenciais a Cm são a verificação forte de tipos em expressões, na passagem de parâmetros a funções e no valor de retorno; a uniformização dos tipos (objetos de qualquer tipo, mesmo classes, podem ser comparados, copiados, passados como parâmetro ou devolvidos por funções); e a detecção automática de dependências entre módulos, simplificando a manutenção através da compilação do conjunto mínimo de classes para atualizar um programa).

Em resumo, Cm procura reaproveitar programas não através do *overloading* de funções e operadores dentro de uma mesma classe, mas sim através da definição de um número ilimitado de tipos (contendo dados e funções) a partir de um conjunto muito menor de definições. Cm foi desenvolvida para a construção do ambiente de desenvolvimento de programas A_HAND [DL 87], um conjunto integrado de ferramentas para a produção sistemática de programas complexos. A_HAND está baseado em *UNIX*, sistema operacional em que o uso de C para programação sistemática é quase obrigatório.

1.4 Requisitos da Linguagem

Certas características foram consideradas fundamentais na escolha de uma linguagem para o A_HAND:

Portabilidade programas devem ser executáveis em diferentes arquiteturas e sistemas operacionais com mudanças mínimas. Para facilitar esse requisito, não devem exigir suporte de execução além de pequenas bibliotecas específicas. O compilador também deve ser executável numa ampla variedade de ambientes

Versatilidade tanto como ou mais que C, deve permitir a construção de programas para diversas áreas de aplicação com rapidez e segurança, mesmo por grupos de programadores

Eficiência o ganho na produtividade do programador não deve ser anulado por uma queda significativa no desempenho dos programas em relação a C.

Verificação de tipos exige-se que verifique tipos de forma estrita, protegendo o programador contra erros de outra forma difíceis de detectar. A verificação de tipos deve ser estrutural e estática

Uniformidade de tipos todos os tipos devem possuir um conjunto mínimo de propriedades comuns, podendo objetos de todos os tipos ser copiados, comparados, passados como parâmetros e devolvidos por funções

Polimorfismo considera-se fundamental a possibilidade de criar tipos de dados polimórficos para reaproveitamento de programas, principalmente entre grupos

Herança conjugada ao encapsulamento de dados, fornece alternativas mais seguras para o reaproveitamento

Nenhuma das alternativas consideradas, como C, C++, Modula-2 e Ada suportava convenientemente todos esses predicados, motivando o desenvolvimento de Cm e de um compilador específico.

1.5 Um Compilador para Cm

Este texto descreve os conceitos principais da programação em Cm e a implementação do primeiro compilador da linguagem. Cm sofreu muitas influências de diversas linguagens e continua sendo aperfeiçoada. Além de refinar a linguagem, o autor colaborou na sua definição, tendo implementado o sistema de compilação Cm como projeto de dissertação de Mestrado na Unicamp.

Uma versão do programa foi tornada pública e usada para implementar alguns projetos num curso a nível de pós-graduação. Essa experiência contribuiu para depurar o compilador e direcionar o desenvolvimento futuro da linguagem. O programa foi desenvolvido em C e pode ser executado em sistemas MS-DOS e *UNIX*.

O restante deste texto ilustra as possibilidades da programação em Cm e aspectos de implementação do compilador, concluindo com uma avaliação geral do projeto e suas extensões. Sendo uma introdução à Cm, o capítulo 2 não requer experiência com outras linguagens para seu entendimento. O capítulo 3 pressupõe conhecimento detalhado da linguagem C.

Capítulo 2

Programação em Cm

E poi che la sua mano alla mia pose
con lieto volto, ondio mi conforta,
mi mise dentro alle segrete cose.
Dante, *Inferno, canto III*

Os conceitos principais da linguagem Cm são apresentados gradualmente, através de exemplos e programas simples. Não serão tratados aspectos particulares de cada implementação, nem tampouco uma referência exaustiva da linguagem.

2.1 Um Programa Simples

Programas em Cm são constituídos por **classes**. Uma classe é uma declaração de dados e funções que manipulam esses dados de forma abstrata, isto é, cada classe conhece o seu conteúdo e oferece informações e serviços a outras classes, mas não é obrigada a informar a essas classes *como* esses serviços e informações são realizados. É possível criar um programa em Cm usando apenas uma classe, mas normalmente eles são compostos por um grande número de diferentes *instâncias*[1] de diversas classes. Uma instância está para uma classe assim como uma variável está para um tipo. Vamos entretanto adiar uma discussão mais completa do conceito de classes e instâncias, apresentando inicialmente programas simples, com três ou menos classes envolvidas.

O programa conhecido como *"hello world"* [KR 78] pode ser considerado um clássico dos exemplos e serve perfeitamente para demonstrar um programa

[1] Instância neste texto não tem o significado usual em português, mas sim representa uma aproximação do inglês *instance*, isto é, um exemplo ou representante de uma categoria — comparável ao significado biológico de *espécime*

mínimo que pode ser compilado e executado para mostrar algum resultado. Eis aqui uma versão em Cm do programa *"hello world"*:

```
class hello ()
/* Primeiro exemplo de programa Cm */
import output;
output () out;

void main ()
{
  out.string ("Hello, world!\n");
}
```

Este exemplo, muito simples, define um programa e também uma classe, chamada *hello*. Supõe-se que o arquivo contendo este exemplo possa ser identificado também por *hello* — o apêndice A apresenta com mais detalhes o ambiente de compilação Cm, como são dados nomes aos arquivos e como programas podem ser compilados.

A declaração **class** introduz toda classe Cm. À parte comentários e espaços em branco, descritos mais adiante, deve ser a primeira cláusula da classe, definindo o seu nome e os parâmetros formais da classe, entre parênteses (neste caso, nenhum).

A cláusula **import** aparece logo a seguir — ela tem por função relacionar todas as classes importadas por *hello*. Como classes definem tipos, esta declaração fornece as informações sobre os tipos de dados usados por *hello*. A única classe importada é *output*, uma classe pré-definida responsável por algumas funções de saída de alto nível.[2] A partir desta declaração, é possível definir instâncias de *output*.

Isto é feito na linha seguinte: a variável *out* é declarada como sendo do tipo "*output*-sem-parâmetros". Note a presença dos parênteses — eles indicam uma instância da classe — e são vazios porque *output*, assim como *hello*, não tem parâmetros. Deste ponto em diante, *out* tem todas as propriedades (dados e funções) da classe importada.

Uma *variável* é uma entidade contendo dados que podem ser modificados a qualquer momento. O *tipo* de uma variável Cm não pode, no entanto, ser alterado.

A classe *hello* define uma única função, chamada *main()*. Esta função é na verdade um procedimento, isto é, uma subrotina que não devolve nenhum valor

[2] tal como em linguagens de programação, o conceito de *nível* aqui se refere a um grau de abstração; *output*, por exemplo, não se preocupa com detalhes de implementação em dispositivos ou verificação de erros

à rotina que a chamou: isso é indicado pela palavra **void** precedendo o nome da função. *Main()* também não recebe nenhum parâmetro de quem a chama, fato denotado pelos parênteses vazios.

O *corpo* de *main()* se encontra entre chaves ({ e }). Tudo o que contém é uma chamada à função *string()* da variável *out*. Essa função faz parte das informações declaradas na classe *output* e incorporadas à variável *out*, daí a referência através do nome da variável, seguida de um ponto. O parâmetro da chamada é uma cadeia de caráteres,[3] com as palavras "Hello world!". O símbolo especial \n denota um caráter especial para mudar a linha de impressão, e a função *string* é definida em *output* para escrever seu parâmetro na tela do terminal do usuário. A classe *output* fornece, entre outras, as funções *character()*, para escrever caráteres, e *dec()*, para números decimais.

A única função de *hello* se resume a esta linha; como todos os comandos e a maioria das declarações em Cm, ela é seguida por ";". Nada mais é necessário para esta classe.

Quando esta classe é compilada e executada, deve produzir como resultado a mensagem

```
Hello, world!
```

no terminal do usuário. A função *main()* de cada classe, se houver, é imediatamente executada quando cada instância começa sua existência. Neste caso, ocorre apenas uma instância de *hello*, declarada implicitamente quando a classe é compilada, que sempre existe (a criação e destruição dinâmicas de instâncias de classe serão apresentadas mais adiante).

2.2 O Formato do Programa

2.2.1 Convenções Léxicas

A linguagem Cm apresenta algumas restrições léxicas que devem ser seguidas na produção de programas corretos, principalmente quanto à pontuação e ao uso dos nomes.

Algumas palavras nos programas Cm são ditas *reservadas*; elas não podem ser usadas para identificadores, mas apenas nas declarações e comandos apropriados, ou dentro de comentários. Sempre que possível, serão apresentadas em negrito, como no exemplo anterior: **class**, **import**, **void**. A linguagem distingüe letras maiúsculas e minúsculas, de modo que *Class*, por exemplo, não é considerada reservada.

Identificadores são palavras definidas pelo programador para dar nome aos objetos por ele criados, como variáveis, tipos e funções. São compostos por letras

[3] comumente denotada, em inglês, por *string*

maiúsculas ou minúsculas, dígitos (0–9) e pelo caráter especial de sublinhado, ou "_". Não podem, entretanto, ser iniciadas por dígitos. *Out*, no exemplo, é um identificador criado pelo usuário, enquanto *output* e *string* foram definidas pelo programador da classe *output*.

Todo identificador deve ser *declarado* apropriadamente antes de ser usado, e não deve, normalmente, ser redeclarado. Por exemplo, a cláusula **import** declarou o identificador *output*, e a linha seguinte declarou *out*. Uma declaração feita pelo usuário de uma variável do mesmo nome seria considerada um erro; entretanto, o fato de *output* ser pré-declarado não implica em nenhum privilégio, e uma variável com esse nome seria válida caso *output* não tivesse sido importada. Mecanismos de *herança* permitem uma redefinição rotineira de identificadores.

Diversos caráteres são considerados como de pontuação — e, tal como espaços em branco e comentários, delimitam o início e o fim de identificadores e palavras reservadas.

Comentários são considerados pelo compilador da linguagem como equivalentes a espaços em branco, e servem meramente como documentação e auxílio ao programador.[4] Podem ser definidos de duas maneiras:

- entre as seqüências "/*" e "*/". Este tipo de comentário, que pode se estender por várias linhas, aplica-se a longas descrições e é às vezes útil para remover temporariamente trechos de código incompletos ou errôneos, visto que pode ser aninhado (por exemplo, a seqüência

$$\underbrace{/*\,ABC\,/*\,DEF\,\underbrace{*/\,GHI\,*/}\,JKL\,*/}$$

  é considerada um único comentário).

- após a seqüência "//". Neste caso, todo o conteúdo da linha após esses caráteres, se houver, é ignorado.

Constantes podem ser números, caráteres, ou cadeias de caráteres. Números são usualmente expressos na base decimal, quando podem ser *inteiros* ou de *ponto flutuante*. Constantes de ponto flutuante são expressas com um ponto decimal, e opcionalmente na notação científica de expoente e mantissa. Por exemplo,

3.1415926535 89 e 0.3141592653589e1

são equivalentes.

Constantes inteiras podem usar outras duas bases:

[4] alguns comentários especiais, definidos para modificar o comportamento do compilador, estão em estudo [Gr 89]

- octal, quando precedidas imediatamente por zero. Neste caso, apenas os dígitos de 0 a 7 podem aparecer na constante.

- hexadecimal, quando precedidas imediatamente por "0x" ou "0X". São válidos todos os dígitos decimais, mais as letras "a" até "f", maiúsculas ou minúsculas, representando os valores de 10 até 15, respectivamente.

Em qualquer caso, uma constante inteira pode ser imediatamente seguida por "u" e/ou "l", maiúsculos ou minúsculos, denotando que são do tipo unsigned e long, respectivamente.

As constantes do tipo caráter consistem de um e apenas um caráter entre apóstrofos, como '0'. Caráteres especiais de controle de dispositivo podem aparecer, precedidas do caráter "barra invertida" (ou *backslash*, ou '\'), bem como seqüências octais ou hexadecimais. Os caráteres para aspas (") e apóstrofo (') também são precedidos pela barra invertida caso apareçam numa constante caráter.

Cadeias de múltiplos caráteres (*strings*) seguem as mesmas regras dos caráteres constantes, mas são delimitados por aspas e podem ter comprimento nulo. Cadeias com comprimento unitário *não* são equivalentes a caráteres.

Normalmente, desde que corretamente delimitados por espaços em branco e/ou pontuação, as palavras e símbolos de um programa Cm podem ser livremente dispostos no programa, ficando seu formato definido pela preferência do programador.[5] Certas características da linguagem, entretanto, forçam uma disposição particular. Por exemplo, o compilador e diversos editores de texto não suportam linhas arbitrariamente longas;[6] ao mesmo tempo, uma *string* não pode começar numa linha do programa e terminar em outra — ela deveria necessariamente ser encerrada por aspas na mesma linha em que foi aberta. Para superar essa limitação, o programa pode ser editado com o caráter "\" imediamente no final de uma linha: tanto o caráter como o término da linha são ignorados e as duas linhas adjacentes são como que aglutinadas. Esse efeito pode se estender por diversas linhas consecutivas do programa.

O exemplo na figura 2.1 ilustra uma cadeia de caráteres relativamente longa. Caso o editor usado para criar o programa não comportasse tal comprimento (nem mesmo com o artifício da linha anterior "Hello world!"), o próximo exemplo (2.2) mostra como truncar a cadeia num ponto arbitrário usando a barra invertida. Entretanto, cadeias de caráteres adjacentes (isto é, separadas apenas, possivelmente, por espaços e linhas em branco), são automaticamente aglutinadas pelo compilador. Deste modo, torna-se visualmente mais organizada a

[5] argumenta-se que a disposição do texto num programa é mais que uma questão de estética, exercendo papel fundamental na sua compreensão

[6] não se pode deixar de lembrar os antigos cartões perfurados, com banda de 80 ou 96 colunas!

```
void main ()
{
      out.string (
      "Hello world!\n");
      out.string ("Esta e\' uma mensagem mais longa que a precedente");
}
```

Figura 2.1: exemplo com cadeia de caráteres longa. Note que o caráter "\" precedendo o apóstrofo não é essencial

```
void main ()
{
      out.string ("Hello world!\n");
      out.string ("Esta e\' uma mensagem mais longa q\
ue a precedente");
}
```

Figura 2.2: semelhante ao exemplo anterior, com linhas de texto mais estreitas

```
void main ()
{
      out.string ("Hello world!\n");
      out.string ("Esta e\' uma mensagem mais longa q"
                  "ue a precedente");
}
```

Figura 2.3: o mesmo exemplo que o anterior, com aglutinação de cadeias

solução do exemplo na figura 2.3. Os três fragmentos de programa, uma vez compilados, são completamente equivalentes.

2.2.2 Palavras Reservadas

Os nomes mencionados a seguir são considerados reservados pela linguagem, não podendo ser usados como identificadores:

auto	double	inherit	sizeof
break	else	int	struct
case	enum	long	switch
char	export	new	type
class	float	register	union
const	for	release	unsigned
continue	global	rename	use
cvirtual	goto	return	virtual
default	if	self	void
do	import	short	while

As palavras seguintes, reservadas para certos compiladores C, não são atualmente consideradas reservadas em Cm, mas desaconselha-se seu uso para evitar possíveis problemas em versões futuras:

asm	extern	signed	typedef
entry	fortran	static	volatile

2.3 Usando Objetos Simples

O exemplo anterior, embora conveniente para esboçar idéias, não chega a ser muito atraente: não usa dados de entrada e sua saída é constante. Além disso, é constante também o principal dado usado internamente ao programa. Introduz-se aqui a declaração de variáveis de tipos padrão, isto é, não estruturados, e alguns dos operadores mais comuns.

2.3.1 Tipos Padrão

Variáveis em Cm são identificadores que recebem e referenciam valores. A variável *out* declarada em *hello* tinha como valor uma instância de uma classe. Todavia, são mais comuns as variáveis cujo tipo é conhecido como *padrão*:

- **char** cujas variáveis podem armazenar caráteres
- **int** referente a números inteiros com sinal

- **float** isto é, números reais
- **double** significando números reais de precisão dupla
- **void** o tipo "nulo"

Desses tipos derivam todos os outros tipos da linguagem.

O nome "float" refere-se à manipulação interna e à entrada/saída das variáveis dos tipos **float** e **double**, usando aritmética de ponto flutuante.

Void não representa realmente um tipo, no sentido de que um objeto[7] de um tipo somente contém valores de um determinado domínio; de fato, não se pode declarar um objeto desse tipo. Todavia, o tipo padrão **void** torna-se útil na declaração de procedimentos (seção 2.7) e alguns tipos apontadores (2.6).

Os tipos chamados "integrais" — **char** e **int** — podem receber qualificadores que modificam algumas de suas propriedades. Por exemplo, o prefixo **unsigned** pode ser aplicado a ambos, indicando que as operações aplicadas a objetos desse tipo não usam sinal (positivo ou negativo); usualmente o domínio dos tipos sem sinal é duas vezes mais extenso que o dos equivalentes com sinal. O modificador **short** pode ser aplicado antes do tipo **int** para indicar um tipo cujos objetos têm as mesmas propriedades daqueles que são **int**, mas que podem ocupar menos memória (e abranger um domínio menor). No entanto, se o espaço e domínio serão *de fato* menores é uma decisão que cabe ao compilador. O programador pode apenas sugerir quais objetos podem ser *short*. Do mesmo modo, o prefixo **long** se aplica ao tipo **int** e denota um inteiro de (possivelmente) maior precisão. A única garantia oferecida é que nenhum caráter será maior que um inteiro, nem um inteiro "curto" maior que um **int** ou um inteiro "longo" menor que um normal. Enquanto o "comprimento" dos tipos integrais se reflete na extensão do seu domínio, aquele dos tipos "reais" implica em menor ou maior precisão. De forma análoga, garante-se que um **double** terá precisão pelo menos igual à de um **float**.

2.3.2 Tipos Enumerados

Os tipos padrão **char**, **int**, e suas variantes podem ser relacionados aos números inteiros. Certos conceitos do mundo real, entretanto, são melhor representados por um conjunto finito e ordenado de nomes simbólicos. Cores, por exemplo, enquadram-se neste caso. Cm permite definir tipos representando tais conjuntos através da cláusula **enum**:

```
enum {vermelho, verde, azul} cor1, cor2;
```

[7] quando não houver margem a dúvidas, entende-se por "objeto" uma variável, um parâmetro de função ou de classe, descritos nas seções apropriadas

Neste caso, *cor1* e *cor2* são duas variáveis que podem assumir qualquer um dos valores *vermelho*, *verde* ou *azul*, como em

```
c1 = c2 = vermelho;
```

Uma alternativa à cláusula **enum** seria associar números às cores, um procedimento potencialmente obscuro.

Dois tipos **enum** diferentes são *incompatíveis* — não é imediatamente possível dar às variáveis de um deles um valor do outro sem uma operação de *casting*. Por exemplo, o fragmento de código

```
type cor = enum {vermelho, laranja, amarelo, verde},
     luminancia = enum {claro, medio, escuro};
int i, j;
cor c1, c2;
luminancia lum;
```

cria dois tipos enumerados distintos, chamados *cor* e *luminancia*. Em Cm sugere-se, como aqui, o uso da cláusula **type** para declarar novos tipos. As variáveis *c1* e *c2* são incompatíveis com *lum* (mas não entre si) e vice-versa, sendo ilegal uma operação de atribuição ou comparação

```
c1 = lum;
```

enquanto

```
c1 = c2;
```

é perfeitamente correta.

Além disso, o mesmo valor não pode pertencer simultaneamente a dois tipos **enum** no mesmo escopo — seria ilegal declarar o tipo

```
type contraste = enum {fraco, medio, forte};
```

devido à presença de *medio* em *luminancia*.

Enumerados e Inteiros

Os tipos enumerados servem essencialmente para forçar a verificação de tipos para um conjunto de valores, e para restringir os possíveis valores assumidos por um objeto. Para todos os demais aspectos, comportam-se como inteiros. De fato, aos valores de um enumerado são associados números inteiros crescentes e consecutivos a partir de zero. Desta forma, após

```
c1 = laranja; lum = medio;
```

as três alternativas

```
i = laranja;        i = c1;        i = lum;
out.dec (i);        out.dec (i);   out.dec (i);
```

são equivalentes e produzem 1 como resultado.

De modo geral, int é um superconjunto de todos os enumerados possíveis. A uma variável inteira podem ser dados todos os valores de qualquer enumerado, mas a recíproca não é imediatamente aplicável. É permitido inclusive indicar outras expressões constantes (contendo números, caráteres e elementos anteriores) associadas aos valores enumerados, que não zero e sucessores, usando uma forma opcional de declaração, como em

```
type FrDDI = enum {
                    Paris = 1,
                    Lyon = 7,
                    Nancy,
                    Grenoble = 76,
                    StEtienne,
                    Avignon = 90,
                    Marseille,
                    Nice = Avignon + 3,
                    StTropez
                  }
```

que relaciona algumas cidades francesas com seus códigos telefônicos internacionais (sem o prefixo zero). Os valores inteiros associados a Paris, Lyon, Nancy, Avignon e Marseille, por exemplo, são respectivamente 1, 7, 8, 90 e 91. Note-se que desta forma é possível que dois ou mais valores sejam associados ao mesmo número.

Pode ser aplicada aritmética a objetos de um tipo enumerado, e o resultado é inteiro. Não se garante, entretanto, que o resultado pertença ao tipo original. Neste exemplo, decrementar uma variável cujo valor é *Lyon* é uma operação válida mas que resulta em 6, sem equivalente em *FrDDI*.

2.3.3 Declarações Simples

A declaração de variáveis de tipos padrão e enumerados é feita indicando-se o nome do tipo seguido por uma lista de identificadores e um ";". Todos os identificadores da lista terão o mesmo tipo. Por exemplo:

```
char k, k1;
int var1;
long int var2, var3;
int var4;
```

declaram algumas variáveis simples. Uma declaração pode incluir o valor inicial dado às variáveis:

```
int var5 = 2, var6 = var5 + 1;
```

A expressão usada para iniciar a variável deve ter um valor constante ou conter identificadores declarados anteriormente.

2.3.4 Declaração de Constantes

É possível em Cm associar identificadores a valores constantes através da cláusula **const**; esses identificadores podem ser usados em qualquer ponto onde constantes são válidas, mas não têm endereço definido (2.6). A cláusula requer menção ao tipo do identificador, como em:

```
const int N = 100;
const char CR = '\n';
```

2.3.5 Operadores

Operadores são funções especiais de um, dois ou três argumentos que se aplicam a objetos em Cm. Classificam-se em unários, binários e ternários, conforme o número de argumentos. São denotados por caráteres e grupos de caráteres especiais,[8] ao invés de funções convencionais, que usam nomes. São comumente empregados em expressões; todos os operadores binários aparecem *entre* seus argumentos; a maioria dos operadores unários aparece *antes* de seu argumento, sendo chamados *prefixos*, enquanto dois deles, "posfixos",[9] podem aparecer depois do argumento. O único operador ternário usa dois caráteres, distribuídos entre os três argumentos.

Cm adota uma verificação "forte" de tipos. É uma linguagem do tipo *strongly typed*. Expressões podem sofrer uma operação apenas se seus tipos são compatíveis. Numa cópia ou comparação de igualdade, dois tipos são considerados compatíveis se:

- têm o mesmo nome e não são classes (2.8)
- são básicos (**char**, **int** e variações, **float** e **double**, **unsigned** ou não); nesse caso a expressão do tipo "menor" sofre uma *promoção* para o tipo "maior"
- são vetores (2.5.1) do tipos compatíveis e tem a mesma dimensão

[8] a exceção são **sizeof** e os manipuladores de memória dinâmica

[9] aportuguesamento grosseiro do inglês *postfix* em oposição a *prefix*

- são apontadores (2.6) de tipos compatíveis
- um é um vetor e outro apontador, e os tipos base são compatíveis
- são estruturas ou uniões (2.5.2) e todos os seus membros têm tipos compatíveis um a um (à exceção da regra anterior)
- são ambos classes de mesmo nome e têm os mesmos parâmetros reais (incluindo *defaults* não mencionados)

Outros operadores têm restrições diferentes. Num exemplo, os operadores que manipulam *bits* exigem operandos de valor integral.

Atribuição e comparação de igualdade

De longe os mais comuns, estes dois tipos de operadores estão definidos para todos os tipos em Cm, não apenas os tipos padrão. O operador de atribuição, denotado por "=", como na expressão

```
expressão_esquerda = expressão_direita
```

copia para a expressão da esquerda o valor da expressão da direita, que é também o resultado da expressão completa.

Os dois operadores de comparação absoluta exprimem igualdade ("==") e desigualdade ("!="). Como não há um tipo primitivo *boolean* (isto é, cujos valores são "verdadeiro" e "falso") em Cm, o valor inteiro 0 é empregado para denotar "falsidade", e qualquer valor diferente de zero, para "verdade".

Aritmética

Os operadores aritméticos são binários e aplicáveis a todos os tipos numéricos, incluindo char. São:

```
+ adição
- subtração
* multiplicação
/ divisão
% resto da divisão inteira
```

Os dois primeiros operadores podem ser também unários, indicando uma multiplicação por 1 ou −1, respectivamente. Os dois últimos têm resultado indefinido se o segundo operando for zero (ou muito próximo de zero, caso seja de ponto flutuante). O operador % requer operandos integrais e o sinal do resultado é dependente de implementação se um dos operandos for negativo (i.e., % pode representar tanto o *resto* como *módulo* da divisão).

Os operadores + e − também se aplicam a apontadores, como visto em 2.6.3.

Operadores aritméticos podem ocasionar *overflow* ou *underflow* dependendo da implementação, do tipo e da magnitude dos operandos.

Os operadores

++ incremento
−− decremento

alteram seu único argumento somando ou subtraindo 1, respectivamente. Seu resultado é o operando *antes* ou *depois* do incremento/decremento, caso o operador seja aplicado de forma prefixa ou posfixa, respectivamente. O exemplo mostra uma declaração e o resultado de expressões com ++ prefixo e posfixo:

int i = 10;

Expressão	Resultado	Valor de i
5 + i++	15	11
5 + ++i	16	11

Ambos são aplicáveis a expressões numéricas e apontadores (2.6.3). O argumento deve indicar um objeto variável, estando proibidas expressões como

2++ ++(i + j) ++ ++i −−i++ ++−i

Operadores de *bits*

Aplicam-se apenas a dados integrais e os manipulam como pequenas seqüências de *bits*:

<< deslocamento para a esquerda
>> deslocamento para a direita

são binários e usam o segundo argumento para indicar de quantos *bits* o primeiro será deslocado. *Bits* vagos mais à direita e esquerda, respectivamente, são preenchidos com zero. Essencialmente seu resultado é o valor numérico do primeiro argumento multiplicado ou dividido por uma potência de dois.

Os operadores

& *e bit-a-bit*
| *ou bit-a-bit*
^ *ou-exclusivo bit-a-bit*
~ *inversão bit-a-bit* (complemento de 1)

tratam dos *bits* dos argumentos individualmente. Todos são binários, à exceção de '~'.

Álgebra Booleana

Considerando qualquer valor numérico diferente de zero como *verdadeiro*, e zero como *falso*,

&&	*e* lógico
||	*ou* lógico
!	negação lógica

retornam o resultado lógico (1 ou zero). Apenas '!' é unário.

Uma expressão com && tem resultado 0 se o primeiro operando é zero, e o segundo não é calculado. Uma expressão com || vale 1 se o primeiro operando não for nulo, e o segundo não é calculado. É importante notar se subexpressões são ou não calculadas quando contém efeitos colaterais, como operadores de atribuição (2.3.5) ou chamadas de função (2.7).

Operadores Relacionais

Todos binários, aplicam-se a números e seu resultado é 1 se a comparação for verdadeira, 0 em caso contrário:

<	menor que
<=	menor ou igual que
>	maior que
>=	maior ou igual que

Endereços e Indireção

Unários e prefixos, manipulam posições de memória e seu conteúdo:

&	obtém o endereço (apontador) do operando
*	obtém o conteúdo de um endereco

São complementares e serão vistos com maior detalhe na seção 2.6.

Referências a Membros

Binários, usados para selecionar um membro de uma estrutura:

.	seleciona o membro nominalmente indicado
->	idem, mas o operando da esquerda é apontador

São abordados nas seções 2.5.2 e 2.6.

Indexação

O único operador [ ] indica um membro de uma coleção (2.5.1 e 2.6).

Sizeof e *casting*

O pseudo-operador **sizeof** tem como único argumento uma expressão ou tipo, e como resultado o espaço em memória ocupado pelo operando (no caso de tipos, por uma variável daquele tipo), numa unidade convencionalmente chamada *byte*. Usualmente ela coincide com o *byte* usual de oito *bits*; garante-se apenas que equivale ao espaço para armazenar um **char**. Mesmo que o operando seja uma expressão, ela nunca será calculada, servindo meramente para determinar um tipo.

A operação de *casting* tenta converter o operando para um novo tipo. Aplica-se por exemplo ao passar um objeto **float** a um operador que espera um *int*, e denota-se escrevendo, imediatamente antes do operando, o tipo final, entre parênteses, como em:

```
float f; int i;
...
i = (int) f % i;
```

Operações de *casting* devem ser empregadas com extremo cuidado.

Seleção Ternária

O resultado é o segundo ou o terceiro operando, dependendo do valor do primeiro. Apenas a primeira expressão e uma das outras é calculada. Assim, em

```
a ? b : c
```

a expressão tem valor *b* se *a* difere de zero; *c* não é calculada. Ocorre o inverso se *a* vale zero.

Seqüências de Expressões

O operador ',' pode separar duas expressões; ambas são calculadas da esquerda para a direita e a expressão total tem o valor da segunda.

Parênteses devem ser usados ao passar uma expressão com ',' como parâmetro de função (2.7).

Atribuição Generalizada

Qualquer um dos operadores aritméticos, deslocamento de *bits* e &, ^ e | pode ser seguido imediatamente de '='. Neste caso, o operador é aplicado e o resultado é imediatamente copiado para o primeiro operando.

Operadores	Associatividade
() [] -> .	$(\bullet * \bullet) * \bullet$
! ~ ++ -- - (*tipo*) * & sizeof	$(*(*\bullet))$
* / %	$(\bullet * \bullet) * \bullet$
+ -	$(\bullet * \bullet) * \bullet$
<< >>	$(\bullet * \bullet) * \bullet$
< <= > >=	$(\bullet * \bullet) * \bullet$
== !=	$(\bullet * \bullet) * \bullet$
&	$(\bullet * \bullet) * \bullet$
^	$(\bullet * \bullet) * \bullet$
|	$(\bullet * \bullet) * \bullet$
&&	$(\bullet * \bullet) * \bullet$
||	$(\bullet * \bullet) * \bullet$
? :	$\bullet * (\bullet * \bullet)$
= += -= *= /= ...	$\bullet * (\bullet * \bullet)$
,	$(\bullet * \bullet) * \bullet$

Figura 2.4: Precedência (decrescente de cima para baixo) e associatividade de operadores em Cm

Precedência e Associatividade

A tabela 2.4 indica as relações de precedência e associatividade entre os operadores Cm. Operadores na mesma linha horizontal têm a mesma precedência, que decresce de cima para baixo.

Parênteses podem ser empregados para alterar a ordem de avaliação; a ordem entre operadores associativos e comutativos, entretanto, pode ser diferente da pretendida mesmo havendo parênteses. Por outro lado, a ordem em que os operandos de um operador são calculados não pode ser determinada: requer-se cuidado quando efeitos colaterais são envolvidos.

A coluna "associatividade" indica como são calculadas expressões com ocorrências adjacentes do mesmo operador, na ausência de parênteses. Assim, 'a / b / c' e '$a = b = c$' são calculadas como '$(a$ / $b)$ / c' e '$a = (b = c)$', respectivamente.

2.4 Estruturas de Controle

Um *comando* em Cm é um trecho de código explicitamente dedicado à execução de alguma tarefa. São sempre seguidos de ';' e somente ocorrem dentro de funções (2.7). O comando mais simples é o comando nulo, ou vazio; representa-

```
...
if (expressao)
  comando;        // executado se expressao != 0
...


if (expressao)
  comando;        // executado se expressao != 0
else
  comando1;       // executado se expressao == 0
...
```

Figura 2.5: Comando if

se simplesmente por ";". Sua utilidade se restringe a alguns casos especiais e programas incompletos. Depois dele, vem o comando de expressão, que pode ser totalmente sem efeito, como

```
a + b;
```

ou exercer algum efeito colateral, como

```
a = b;
```

ou

```
a++;
```

Um comando composto consiste numa série de comandos entre "{" e "}". Cada comando interno é executado em sucessão, mas o conjunto pode ser tratado como um único comando e inclusive declarar variáveis locais (seção 2.7).

O restante dos comandos em Cm destina-se a alterar o fluxo de execução do programa, sendo normalmente usados em função dos anteriores.

2.4.1 Comandos de Decisão

O comando **if** desvia a execução entre duas alternativas. Tem duas formas possíveis, como na figura 2.5. Múltiplas alternativas podem ser expressas encadeando-se diversos comandos **if**, como em 2.6. O emparelhamento de cada **else** é feito com o **if** anterior mais próximo. As duas alternativas na figura 2.7 indicam como implementar um **if** encadeado com a primeira alternativa sem **else**.

Tanto **if** como os comandos de iteração (2.4.2) calculam uma expressão de tipo não-estruturado (**char**, **int** e suas variações, **float** ou **double**, apontadores

```
...
if (expressao)
  comando;          // executado se expressao != 0/
else
  if (expressao1)
    comando1;       // se expressao == 0 e expressao1 != 0
  else
    comando2;       // se ambas sao zero
...
```

Figura 2.6: Comandos if encadeados

```
...                          ...
if (expressao)               if (expressao)
  if (expressao1)            {
    comando1;                  if (expressao1)
  else                           comando1;
    ;                        }
else                         else
...                          ...
```

Figura 2.7: Comandos if encadeados sem else anterior

```
switch (expressao)
{
  declarações;
  case valor1: comandos1;
  case valor2: comandos2;
  case valor3: comandos3;
  ...
  case valorn: comandosn;
  default:     comandosd;
}
```

Figura 2.8: Forma geral do comando **switch**

(2.6)) para decidir a ação a ser tomada. Considera-se qualquer valor de tipo apropriado diferente de zero como "verdadeiro", de modo que "**if** (x != 0)" e "**if** (x)" são equivalentes.

Quando há diversos testes de igualdade com valores constantes, o comando **switch** torna-se mais prático que uma sucessão de **if**s encadeados. Sua forma mais comum é ilustrada na figura 2.8: isto é, uma expressão seguida por um comando composto. Quando um **switch** é executado, a expressão é calculada e o resultado é comparado com cada valor após rótulos **case**. A execução se inicia no comando após o rótulo cujo valor se iguala ao da expressão. Caso nenhum rótulo satisfaça essa condição, a execução começa após a cláusula **default**. Se esta não existir, nenhuma ação é tomada e o comando simplesmente termina.

Quando um rótulo corresponde ao valor da expressão, a execução do **switch** não termina ao se encontrar o próximo rótulo, mas prossegue até ser executado um comando **break** (isto é, à exceção da seleção inicial, os rótulos **case** não interferem na execução); usualmente **break** termina cada série de comandos após um **case**, mas, se não estiver presente, é possível implementar seqüências de comandos para dois ou mais **case**s em que uma é um sufixo próprio de outra.

Também é possível dispor mais de um **case** para a mesma série de comandos, como no exemplo da figura 2.10.

As consideraçõess a seguir se aplicam ao comando **switch**:

- não é exigido que o comando seguindo a cláusula **switch** seja composto, mas como o escopo de **switch** (e os seus rótulos **case**) somente se aplica a esse comando, esse caso não ocorre na prática

- os valores dos rótulos **case** devem ser todos constantes integrais calculáveis

```
switch (k)
{
    case 1:     a++;
    case 2:     b++;
                break;
    case 3:     c++;
}
```

```
if (k == 1)
{
  a++; b++;
}
else if (k == 2)
        b++;
     else if (k == 3)
             c++;
```

Figura 2.9: Equivalência entre if e switch

```
type mes = enum {jan, fev, mar, abr, mai, jun, jul, ago,
                 set, out, nov, dec};
mes m;
int dmes, ano;
...
switch (m)
{
  case abr: case jun:
  case set: case nov:
        dmes = 30;
        break;
  case fev:
        dmes = ano % 4 || ano % 400 ? 28 : 29;
        break;
  default:
        dmes = 31;
}
```

Figura 2.10: Exemplo geral do comando switch

durante a compilação. O mesmo valor não pode ... ter mais de uma vez no mesmo comando **switch**

- certas implementações podem gerar código de decisão muito eficiente se a diferença entre os valores máximo e mínimo da expressão de seleção e/ou constantes case for pequena
- a expressão de seleção deve ter tipo integral
- o comando composto pode conter declarações de variáveis cujo *escopo* (seção 2.7.1) se resume a esse comando. É possível também determinar efetivamente o valor inicial dessas variáveis.[10]

2.4.2 Comandos de Iteração e Desvio

Esta categoria de comandos destina-se principalmente a desviar o controle de execução de modo a formar ciclos (iterações) e assim repetir a execução de uma série de instruções um número arbitrário de vezes. Essencialmente se compõe de quatro partes (nem todas necessitam estar presentes):

1. um prólogo de iniciação
2. o teste de iteração, que decide se o ciclo deve ou não continuar
3. a ação a iterar (comandos Cm)
4. uma mudança de estado que influencie a decisão 2

Todas essas partes, exceto a primeira, podem ser executadas zero ou mais vezes.

Dois comandos Cm não são propriamente de iteração, mas estão sem dúvida associados a essa categoria:

break, já visto em conexão com **switch**; sua execução interrompe imediatamente o ciclo correntemente em execução

continue, que durante uma iteração despreza todas as instruções até o último comando da parte 3. Isto é, transfere a execução de volta ao teste 2.

O Comando while: Ciclo com Teste Antecipado

O comando **while** não inclui uma ação preliminar. Caracteriza-se por decidir se cada iteração será executada imediatamente antes do comando a iterar. A ação 4 deve ser feita pela própria iteração, e o comando tem a forma geral:

[10] diferentemente do que ocorre em C

while (*expressão*)
    comando;

O *comando* será executado enquanto o valor da *expressão* (que deve ser redutível a um número) for diferente de zero.

O Comando do: Ciclo com Teste ao Final

Esta variante de **while** apenas inverte a posição do teste; também executa iterativamente um *comando* enquanto a *expressão* for diferente de zero:

do
    comando;
while (*expressão*);

Como a execução se inicia no *comando*, este é executado pelo menos uma vez.

O Comando for: Duas Expressões e um Teste

Este é o comando de iteração estruturado mais genérico. Inclui numa única construção todas as etapas 1 a 4:

for (*expressão1*; *expressão2*; *expressão3*)
    comando;

A *expressão1* é executada imediatamente e apenas uma vez; a segunda expressão é calculada a cada iteração: se não for zero, o *comando* é executado, a *expressão3* é calculada e o ciclo prossegue; em caso contrário, a execução do comando termina. Note que a execução de um **continue** transfere a execução para a avaliação de *expressão3*, não *expressão2*.

Como qualquer das três expressões pode estar ausente, o comando **for** pode substituir tanto **while** como **do**. O uso de um ou outro comando fica a critério do programador. A figura 2.11 mostra de forma esquemática como testar a condição de iteração em vários pontos de uma seqüência de comandos (no caso, apenas dois) executados repetidamente por um comando **for**.

Outros Comandos

O comando **return** termina a execução de uma função (2.7) e será discutido nessa seção.

O fluxo de execução de um programa Cm pode ser alterado de forma drástica através do comando **goto**. **Goto** tem como parâmetro um único argumento, que é o *rótulo* para onde será desviada a execução. Um rótulo é um identificador que marca um trecho de programa. Podem aparecer antes de qualquer comando Cm, desde que seguidos por ":", como em:

```
for (e1; e2; e3)
{
    comando1;
    comando2;
}
```

```
for (e1; ; e3)
{
    comando1;
    if (! e2)
        break;
    comando2;
}
```

```
for (e1; ; e3)
{
    comando1;
    comando2;
    if (! e2)
        break;
}
```

Figura 2.11: três exemplos do comando **for**, com o teste de iteração no princípio, meio e fim de uma seqüência de dois comandos

```
    ...
rotulo1:
    comando;
    ...
    goto rotulo1;
    ...
```

Rótulos não precisam, tal como ocorre com variáveis, ser declarados antes de seu uso em **goto**, mas somente podem ser usados internamente a uma função; e como são particulares à função onde estão definidos, não se pode usar **goto** para saltar de uma função a outra.

2.5 Tipos Estruturados

A partir dos tipos básicos em Cm é possível compor tipos mais complexos. Os dois construtores de tipos mais comuns simplesmente combinam várias instâncias de outros tipos. Se todas as instâncias são do mesmo tipo, obtém-se um "vetor", ou *array*. A outra forma de construção pode agregar tipos diferentes, e se subdivide em estruturas (**structs**) e uniões (**unions**).

2.5.1 O Tipo *Array*

Um *array* ou vetor é uma coleção de objetos de mesmo tipo armazenados consecutivamente em memória; é caracterizado pelo tipo dos objetos e pelo seu número, que é determinado durante a compilação e permanece constante por toda a duração do vetor. Cada objeto pode ser referenciado através do operador de indexação ([...]), que tem como operandos o vetor e um número inteiro. O primeiro objeto, ou *elemento*, tem sempre índice zero—portanto o último elemento tem índice igual ao número de elementos menos um.

Arrays são declarados agregando-se uma expressão inteira entre '[' e ']' ao tipo dos elementos. Por exemplo:

```
const int s = 15;
int [10] v1;
char [s * 2] v2, v3;
```

declara v1, como capaz de armazenar dez inteiros; e v2 e v3,[11] agrupando 30 caráteres. Note-se que a expressão deve ter valor constante. Feita a declaração, vetores podem constar em expressões, como:

```
v1 [0] = v1 [i] + v1 [j];
```

em que ao primeiro elemento de v1 é atribuída a soma do i-ésimo e j-ésimo elementos.

Caso seja executado o acesso a um *array* com índice menor que zero ou maior ou igual ao número de elementos, o resultado é dependente de implementação. O compilador Cm não introduz nenhuma verificação para que esse evento seja detectado e corrigido. Índices ilegais podem ser usados com cautela para o acesso a outras variáveis, mas essa prática é fortemente desencorajada.

Um caso particular do tipo vetor ocorre quando seus elementos são caráteres. Um vetor de caráteres, também conhecido por *string*, é tratado de forma especial. Embora o tamanho máximo não possa aumentar, muitos programas usam *strings* de forma dinâmica: considera-se que um elemento de valor zero indica o fim do vetor. Cadeias de caráteres constantes são criadas implicitamente com um zero após a última posição definida pelo programador. Assim,

```
"Hello"
```

é um vetor com seis caráteres. No início da execução de um programa Cm contendo essa *string*, o elemento imediamente após o 'o' terá o valor 0 (ou '\0' , uma seqüência especial equivalente que pode ser usada em constantes tipo caráter). Se for atribuído esse valor ao elemento de índice 1, por exemplo, o restante da cadeia permanece inalterado mas, por convenção, normalmente considera-se a *string* agora equivalente a "H".

Vetores *multidimensionais*, conhecidos também por *matrizes*, são declarados usando-se múltiplas seqüências de '[...]'. A declaração

```
int [10][30] v4;
```

produz uma matriz v4 de dez vetores, cada um dos quais é um vetor de trinta inteiros, num total de $10 \times 30 = 300$ inteiros. Cada "linha" de trinta elementos é armazenada consecutivamente em memória, uma após a outra. O acesso é feito de modo análogo à declaração:

```
v4 [i][j]
```

[11] Nota ao programador C: Cm usa expressões de tipo, não declaradores [Se 81] — portanto, v2 e v3 têm o mesmo tipo. Notar também a posição do identificador

representa o j-ésimo inteiro da i-ésima "linha".

Cm admite que vetores sejam comparados e atribuídos. Além disso, cada elemento de uma matriz continua sendo um vetor. Deste modo, a expressão

```
v4 [i] = v2;
```

é válida e copia todo o *array* v2 em um dos subvetores de v4.

O número de dimensões para um vetor não é limitado pelo compilador Cm.[12] Vetores cuja extensão é definida dinamicamente (durante a execução do programa) podem ser criados usando-se apontadores (2.6).

2.5.2 Estruturas e Uniões

Uma *estrutura* também é o resultado da composição de diversos objetos. Distingüe-se de um vetor pelos tipos, não necessariamente iguais, e pelo acesso, que é enumerado e não indexado. A declaração **struct** define um tipo estrutura, como a cláusula *record* de linguagens como Pascal; os diversos objetos são conhecidos como *membros*[13] e aparecem entre "{" e "}":

```
struct {
          char [30] nome;
          int idade;
       } s1, s2;
```

declara duas variáveis, s1 e s2, ambas compostas por um *array* de trinta caráteres e um inteiro. Membros de uma estrutura são expressos pelo operador de seleção (.), como em:

```
s1.nome = "X. Y. de Z.";
s1.idade = s2.idade + 1;
```

Como ocorre com vetores, estruturas podem ser comparadas e copiadas entre si, desde que sejam do mesmo tipo.

Uma *união* é uma construção sintaticamente similar a uma estrutura. A diferença, além da substituição de **struct** por **union**, é que os membros de uma união compartilham o espaço de memória ocupado, de modo que não podem ser usados simultaneamente. A linguagem não provê mecanismos para distingüir qual dos membros está sendo usado a cada momento. Normalmente, uniões aparecem como membros de uma estrutura, ao lado de um membro adicional de controle. Como exemplo, o trecho de código na figura 2.12 define um vetor para armazenar dados sobre 500 publicações. O tipo enumerado *obra* é empregado para definir qual dos membros da união está sendo usado. Considera-se que

[12] mas pode ser restrito pelo compilador C

[13] o nome é uma herança de C. Considera-se mais apropriado falar em *campos* ou *componentes*

```
type obra = enum {livro, periodico, manual};
const int Cpr_Titulo = 100;

struct {
        obra tipo;
        char [Cpr_Titulo] titulo;
        union {
                struct {
                        char [30] autor;
                        int ano;
                    } l;
                struct {
                        int volume, numero;
                    } p;
               }dados;
    } [500] catalogo;
```

Figura 2.12: Vetor com estruturas e uniões

todas as publicações tenham um *título*; *livros* têm também um *autor* e um *ano* de publicação; *periódicos* não têm autor ou ano, mas usam uma identificação de *volume* e *número*; *manuais* não têm nenhuma informação adicional. O acesso a membros de estrutura e união é feito pelo mesmo operador '.' (que, deve ser notado, é associativo à esquerda):

```
catalogo [i].titulo = "Programacao Cm";
catalogo [i].tipo = livro;
catalogo [i].dado.l.autor = "X. Y. de Z.";
catalogo [i].dado.l.ano = 1980;
```

É permitido especificar o tamanho ocupado em *bits* por um membro de tipo inteiro. Para isso, o nome do membro é seguido de ":" e de uma expressão constante inteira. Como exemplo, o fragmento de código

```
struct   {
             ...
             int ocupantes: 4, farois: 3, importado: 1;
             ...
         } automovel;
```

declara, entre outros, os membros *ocupantes*, usando 4 *bits*, *farois*, com 3, e *importado*, com 1 *bit*. Supondo-se que nenhum *automóvel* suporte mais que $2^4 = 16$ ocupantes, nem mais que $2^3 = 8$ faróis, usa-se um *bit* para discernir entre veículos nacionais e importados. Assim, conhecendo-se antecipadamente o domínio de variáveis, é possível determinar membros de tamanho menor que um *byte*, ou **char**, economizando espaço de memória.[14]

Membros com tamanho especificado são úteis para implementar listas de *bits* (*bitmaps*) curtos; também podem ser declarados membros anônimos, usados para preencher espaço entre outros membros (alinhamento). Entretanto, não podem ser tratados como vetores (isto é, indexados), nem têm sinal numérico ou endereço individual.

2.6 Apontadores

Objetos definidos pelo tipo "apontador" não são usados para conter valores: seu conteúdo são *referências* a outros objetos. Apontadores têm extrema utilidade na construção de estruturas de dados dinâmicas, que alteram sua configuração durante a execução.

Um *tipo apontador* (também chamado *ponteiro* ou *pointer*) se define em termos de outro tipo com a declaração *. Um objeto de tipo apontador pode endereçar diferentes objetos do tipo original através do operador homônimo *. No exemplo

```
char *p;
char k1, k2;
...
p = &k1;
*p = '1';          /* equivalente a "k1 = '1'" */
p = &k2;
*p = '2';          /* equivalente a "k2 = '2'" */
```

o apontador para o tipo **char** p é usado para alterar o valor de duas variáveis tipo caráter, usando o operador "&" para obter seus endereços. É possível imaginar

[14] não se garante uma economia efetiva de espaço. Membros com tamanho especificado (*bitfields*) têm os *bits* aglutinados pelo compilador C de modo a ocupar trechos de um tipo padrão, normalmente um **int**. No exemplo, caso a implementação do compilador C use inteiros com dois *bytes*, os três membros, embora contendo apenas 4 + 3 + 1 *bits*, ou 1 *byte*, exigiriam dois *bytes* completos. Ainda assim, a declaração economiza espaço em relação a três membros individuais, tipo **char**. Haveria, entretanto, *desperdício* de memória, em implementações onde o tipo **int** usa 4 *bytes*—além da perda de um *byte*, o acesso a membros *bitfield* é normalmente mais lento que o a membros convencionais

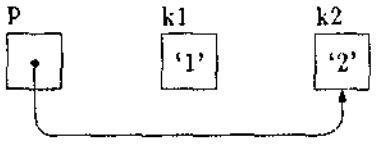

Figura 2.13: Visualização de variáveis usando um apontador

graficamente essas três variáveis segundo a figura 2.13. Note que este exemplo é muito simples e não corresponde ao uso efetivo de *pointers*.

As operações permitidas com objetos apontadores são a indireção ou referência (*), atribuição, comparação e aritmética restrita. As operações permitidas a expressões com apontadores e * são as mesmas previstas para os tipos apontados. De modo geral, uma vez obtido o endereço de uma variável v usando um apontador p, como em $p = \&v$, $*p$ e v são como que sinônimos. A apontadores também pode ser atribuído o valor NIL, uma constante pré-definida. Apontadores "contendo" NIL convencionalmente não apontam para objeto algum, e o resultado de *NIL é imprevisível.

2.6.1 Apontadores a Estruturas e o Operador "->"

Definem-se apontadores para estruturas e uniões do modo usual. O fragmento de código

```
type s1 = struct {
                   ...
                   int field;
                   ...
                 };
s1 s;
s1 *ps;
```

declara uma estrutura s e o apontador ps. Após a execução de

```
ps = &s;
```

$*ps$ e s referem-se ao mesmo valor. Entretanto, Cm define o operador * de indireção como de menor precedência que o operador de seleção ".". Deste modo, a expressão

```
*ps.field
```

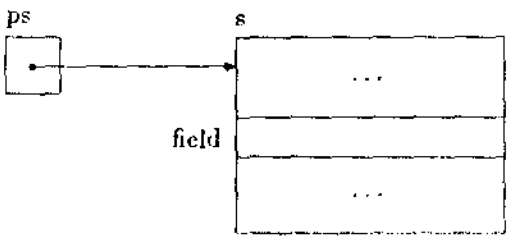

Figura 2.14: Apontador para uma estrutura

é incorreta, pois denotaria a seleção de membro de um apontador (que não tem membros) e a posterior indireção daquele membro (que não é um apontador). É necessário o uso de parênteses, como em

```
(*ps).field
```

mas apontadores a estruturas são tão comuns que é definido o operador "->", mais simples e legível. O exemplo anterior é equivalente a

```
ps -> field
```

e podemos visualizar a situação como na figura 2.14.

Estruturas que se apontam mutuamente podem ser definidas usando declaração antecipada de tipos, caracterizadas pela palavra **struct** ou **union** sem menção aos membros:

```
type x = struct;
type y = struct {
                    ...
                    x *p;
                    ...
                }
type x = struct {
                    ...
                    y *q;
                    ...
                }
```

Figura 2.15: Apontador e alocação dinâmica

2.6.2 Alocação Dinâmica de Memória

Uma das facilidades mais importantes em Cm é a alocação dinâmica de objetos; isto é, dados são criados *durante* a execução do programa, usando-se o operador **new**. **New** tem como argumentos um nome de tipo e opcionalmente uma expressão inteira. Sua execução causa a criação dinâmica de um ou mais objetos daquele tipo. Como ilustração, usando o primeiro exemplo desta seção, o trecho de programa Cm

```
char *p;
...
p = new (char);
```

obtém uma nova área de dados com espaço suficiente para conter um caráter. Efetivamente, cria-se uma nova variável "anônima", como na figura 2.15.

É claro que essa técnica pode não ser muito eficiente, exigindo um apontador para cada objeto alocado. Além da criação de *arrays*, apresentada na seção 2.6.3, apontadores são normalmente empregados em estruturas de dados dinâmicas, e não isoladamente. Essas construções fazem uso de declarações **struct/union** com membros apontador. Uma lista ligada para armazenar inteiros, talvez o exemplo mais simples [Ku 73], é definida por

```
type lista = struct
             {
                 int dado;
                 lista *proximo;
             };
lista *inicio;
```

Cada objeto do tipo *lista* tem a capacidade de referenciar outro do mesmo tipo. Uma série de operações **new** cria diversas cópias de pares (dado + apontador), que podem ser endereçadas usando o membro *proximo* para formar uma estrutura de qualquer comprimento. O começo da lista pode ser referenciado usando-se *inicio* e, a partir daí, operações de indireção tornam acessíveis todos os outros componentes da lista, cujo término é indicado pelo valor NIL do último

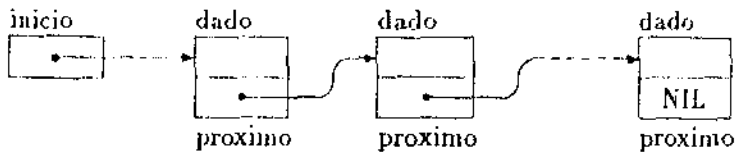

Figura 2.16: Lista ligada simples

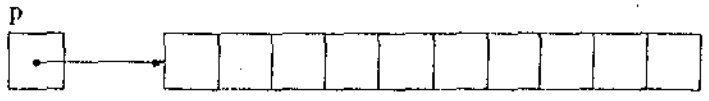

Figura 2.17: Vetor de caráteres criado dinamicamente

apontador, como na figura 2.16.

Outras construções dinâmicas podem ser definidas, usando apontadores de forma arbitrariamente complexa.

2.6.3 Vetores e Apontadores

É possível criar dinamicamente várias ocorrências de um tipo usando o segundo argumento de **new**:

```
p = new (char, 10);
```

produz dinamicamente dez caráteres, armazenados consecutivamente na memória (figura 2.17). Essa disposição (equivalente à de um *array*) pode explorar outra característica de apontadores: aritmética de endereços. Quando se soma 1 a um apontador contendo uma referência válida, ele passa a endereçar o objeto *seguinte* ao referenciado inicialmente, supondo que haja algum. Em geral, quando há m objetos do mesmo tipo dispostos consecutivamente em memória, e um *pointer* para o i-ésimo objeto, a soma de um valor inteiro k ao apontador produz uma referência ao i + k-ésimo objeto (supõe-se que $m > i + k \geq 0$, o resultado em caso contrário depende de implementação e é em geral incorreto).[15]

[15] em termos de linguagem de máquina, na qual um apontador é um valor inteiro sem sinal cujo valor é um endereço de memória, a soma de um inteiro n a um *pointer* para o tipo T é traduzida como a soma de n × **sizeof** (T) àquele endereço. As duas somas são correspondentes apenas se T ocupa apenas uma "posição" em memória; usualmente posições—e endereços—são definidos em termos de *bytes*, e **sizeof** (**char**) = 1

De modo análogo, caso se *subtraiam* dois *pointers* cujos "valores" (no diagrama, as pontas das setas) estejam separados por n objetos do mesmo tipo, o resultado é do tipo inteiro e vale n. As operações de soma e subtração com integrais e subtração entre apontadores do mesmo tipo são a única forma de aritmética permitida entre objetos apontadores.

No exemplo anterior, o código

```
int i;
for (i = 0; i < 10; i++)
    *p++ = '␣';
```

ilustra como preencher a memória alocada com espaços em branco, usando aritmética de apontadores. A expressão "*p++" refere-se ao dado apontado por p (antes do incremento).

A utilidade da aritmética de apontadores não se limita a objetos criados por **new**. Uma vez apontando para qualquer coleção de objetos consecutivos do mesmo tipo—por exemplo, um vetor—*pointers* podem ser usados como o índice de um *array*, para o acesso direto a qualquer um dos objetos. Na realidade, a aproximação entre vetores e apontadores pode ser levada mais adiante. A um objeto cujo tipo é "vetor de T com n elementos" aplicam-se as mesmas operações de um "apontador para tipo T" e vice-versa. O operador de indexação [...] aplica-se a um apontador assim como o operador de indireção "." e a soma com inteiros a vetores, seguindo a regra

```
T [n] a;
T *p;
int k, l;
```

$$p = \&a[l] \Rightarrow \begin{cases} *(p+l+k) == a[l+k] \\ p[l+k] == *(a+l+k) \end{cases}$$

Em particular,
p = &a [0] ⇒ *(p + k) == a [k]

E, como o *nome* de um vetor representa por definição o *endereço* de seu primeiro elemento,

p == & a [0] ⇔ p == a

O uso de vetores como ponteiros tem limites: são proibidas operações aritméticas com vetores, e uma expressão como &a é incorreta (na verdade, redundante).

2.6.4 Precauções

Objetos de tipo apontador emprestam flexibilidade a qualquer linguagem mas seu uso requer precauções. Referências a *pointers* com endereços incorretos podem ser desastrosas. Cm não oferece muitas facilidades na detecção dinâmica deste tipo de erro, apenas verificações semânticas durante a compilação. Por exemplo, somente podem ser atribuídos ponteiros de tipos diferentes após uma operação de conversão de tipo (*casting*). Ponteiros podem ser declarados usando o tipo **void**: um objeto do tipo **void** * pode endereçar qualquer tipo tipo sem a necessidade de *casting*, mas exige conversão de tipo para qualquer aritmética.

A capacidade de alocação de **new** não é infinita e depende da configuração local. **New** retorna o valor NIL se a alocação não pôde ser satisfeita. O operador **release** libera objetos criados por **new** e pode ser invocado quando estes não são mais necessários. **Release** também muda o valor de seu parâmetro para NIL, se possível.

Um mesmo objeto não deve sofrer **release** mais de uma vez (através de dois apontadores). O resultado é imprevisível, assim como se o argumento de **release** não foi obtido através de **new**. Cm não implementa recuperação automática de memória não-acessível (*garbage collection*). Objetos criados por **new** e não endereçados por nenhum ponteiro não podem ser usados, nem estarão disponíveis novamente, até o final de execução do programa.

2.7 Funções

Em Cm, funções implementam o conceito de subrotina, isto é, trechos de programas que são definidos uma única vez e podem ser usados diversas vezes.

Toda função se define e distingüe pela *interface*, composta de três elementos:

o nome, com o qual a função é usada

os parâmetros, ou argumentos, que modificam o comportamento da função

o tipo do valor de retorno, que identifica como serão interpretados os resultados da função

Declara-se uma função mencionando-se os três elementos, nessa ordem, seguidos por ";":

```
int lowercase (int k);
```

declara uma função *lowercase*, que recebe um único parâmetro do tipo **int** e calcula e devolve um valor do mesmo tipo.

Uma função pode ser declarada mais de uma vez, mas todas as declarações devem ser iguais.[16] Elas servem para tornar a interface de uma função conhecida, mas não introduzem nenhum código. Isto é feito com a *definição*, no caso de uma função que converte caráteres para minúsculas:[17]

```
int lowercase (int k)
{
  return ('A' <= k && k <= 'Z' ?
          k + 'a' - 'A' :
          k);
}
```

Uma vez conhecida a função, seja declarada ou definida, ela pode ser usada em expressões, através de seu nome e parâmetros apropriados:

```
int char1, char2;
...
char2 = lowercase (char1);
...
```

Cm adota a chamada passagem de parâmetros por valor [Aa 86]: o *parâmetro formal* da função (*k*) é como uma variável particular a lowercase; quando a função é chamada tendo como *parâmetro real char1*, o valor desta expressão é copiado para *k* e o código de lowercase executado até o final ou até ser encontrado o comando **return**. Se o tipo da função (isto é, o tipo do valor de retorno) não for **void**, a chamada da função é uma expressão desse tipo e pode ser usada em outras expressões. A função deve terminar sua execução com o comando **return**, seguido de um argumento daquele tipo — esse argumento é o valor retornado pela chamada da função. Se o tipo de retorno da função for **void**,[18] o comando **return** não pode ter argumentos e é opcional, caso seja o último comando executado pela função.

A função definida a seguir

```
double convtemp (double temp; int fahr2cels = 1)
{
  return (fahr2cels ? (temp - 32) * 5 / 9 :
                      9 * temp / 5 + 32);
}
```

[16] isto é, com os mesmos tipos, tanto para valor de retorno como para todos os parâmetros. Os nomes dos parâmetros também devem coincidir devido à passagem nominal (apresentada a seguir). As únicas diferenças permitidas entre redeclarações da mesma função não-herdada (2.8.3) são os valores implícitos *(default)* de parâmetros. Cada declaração pode apenas adicionar novos valores implícitos

[17] supondo codificação de caráteres consecutivos e ordenados, como no código ASCII

[18] a função é um efetivamente um *procedimento*, como definido em Pascal [JW 75]

converte temperaturas entre as escalas Celsius e Fahrenheit e é um exemplo de função com mais de um parâmetro—*temp*, a temperatura a converter; e *fahr2cels*, a direção da conversão (se zero, Celsius → Fahrenheit; em caso contrário, Fahrenheit → Celsius). Este último parâmetro tem *valor default*, ou implícito, de 1. Um parâmetro formal para o qual haja um *default* especificado não exige um parâmetro real correspondente na chamada da função. No caso, as duas chamadas em

```
convtemp (x, 1) + convtemp (x);
```

são equivalentes.

Normalmente, parâmetros formais com *default* são os últimos parâmetros da interface, pois é a ordem dos parâmetros reais que os associa aos formais. É possível no entanto passar parâmetros reais nomeando explicitamente os parâmetros formais a que se referem. Quando isso ocorre, como em

```
convtemp (fahr2cels = 1, temp = 212.0)
```

a ordem é irrelevante. Importa somente que não fiquem parâmetros formais sem valores associados (passados ou por *default*). Cm não suporta a combinação dos dois tipos de passagem de parâmetro (com e sem nome) numa mesma chamada.

2.7.1 Níveis Léxicos

O *corpo* de uma função (o trecho entre os { e } mais externos) em Cm é um *bloco* ou *nível léxico*. Blocos léxicos são trechos de programa onde podem ser criados novos identificadores, que são ditos *locais* ao bloco, ou de *escopo* reduzido ao bloco. Isso significa que:

- caso haja nomes iguais declarados dentro e fora de um bloco, qualquer uso desses identificadores dentro do bloco refere-se à entidade declarada dentro do bloco (isto é, escopos internos *mascaram* externos) enquanto ocorrências fora do bloco referem-se à entidade externa
- um identificador interno a um bloco não é acessível fora dele

Parâmetros formais de funções estão no mesmo bloco léxico de seu corpo. Blocos podem por sua vez conter outros blocos, num aninhamento arbitrário. Além da cláusula **inherit** (2.8.3) e dos nomes de membros em estruturas e uniões, essas regras de escopo são o único caso previsto em que um nome definido pelo programador Cm pode ser usado de novo com outro significado.

Regras de escopo também se aplicam a rótulos e ao comando **goto**: caso o rótulo "x:" ocorra mais de uma vez na mesma função, em blocos léxicos diferentes, o comando "**goto** x;" desvia a execução para a ocorrência no menor bloco léxico que abrange ou coincida com aquele do comando.

Variáveis em blocos léxicos são criadas dinamicamente quando a execução do bloco é iniciada, e deixam de existir quando esta termina. Cada execução da função provoca a criação de um novo conjunto de variáveis locais. Se a função é recursiva (chama a si mesma, direta ou indiretamente), haverá diferentes conjuntos de variáveis, simultaneamente.

Isso ocorre porque a *classe de armazenamento* associada a variáveis locais a níveis léxicos é normalmente **auto** (*automatic*). A classe de armazenamento[19] é um atributo (assim como o tipo) de cada objeto, opcionalmente definido imediatamente antes do tipo. Objetos **auto**, por exemplo, são automaticamente criados e destruídos quando o bloco a que pertencem respectivamente inicia e termina sua execução. Outras classes de armazenamento são

register usada para variáveis não-estruturadas que serão muito usadas no bloco. É uma sugestão ao compilador para otimizar o tempo de acesso a esses objetos[20]

+ esta classe define variáveis "estáticas", únicas durante toda a execução do programa e que sobrevivem (isto é, mantém seu valor) entre diferentes execuções do seu bloco. Uma função como

```
void f (...)
{
  + int c = 0;
  c++;

  ...
```

por exemplo, registra quantas vezes foi chamada. De certa forma, variáveis estáticas comportam-se como variáveis declaradas fora de funções, mas de escopo delimitado. Não são criadas múltiplas instâncias em ativações recursivas da função.

const indica objetos que não podem ser alterados durante a execução. Podem ser de qualquer tipo, mas não têm endereço acessível.

Outros aspectos importantes de classes de armazenamento são apresentados em 2.8.

[19] um conceito totalmente desvinculado das classes de Cm, traduzido do inglês *storage class*

[20] o compilador não é obrigado a cumprir essa sugestão. O nome **register** vem do uso de *registradores* de máquina, um recurso usualmente escasso. Portanto, mesmo que vários objetos sejam especificados como **register**, é possível que apenas os primeiros (e mesmo nenhum) sejam efetivamente alocados em registradores, ficando o restante como **auto**. Como não ocupam memória convencional, a objetos declarados **register** não se pode aplicar o operador &

2.7.2 Passagem de Parâmetros

A passagem de parâmetros por valor em Cm implica que alterações no parâmetro formal de uma função não se refletem no parâmetro real uma vez terminada a execução da mesma. Caso seja necessário preservar alterações, é possível passar o *endereço* de objetos como parâmetro real. Neste caso, o parâmetro formal torna-se um ponteiro para o tipo do objeto. No exemplo

```
int x = 1, y = 1;
...
void f (int a; int *b)
{
  a = *b = 0;
}
```

após a chamada f(x, &y), os valores de x e y são respectivamente 1 e 0.

Parâmetros de funções podem ser de qualquer tipo, mesmo estruturado. Neste caso, o que ocorre durante a chamada da função depende do tipo

- se o parâmetro formal é uma estrutura, união ou classe, o valor é copiado integralmente para a função, e modificações no parâmetro formal não são preservadas no real. Pode ser interessante usar um ponteiro como parâmetro, evitando a operação de cópia
- *arrays* não são copiados: ao invés disso, o endereço do primeiro elemento é passado e o parâmetro formal pode efetivamente ser usado como se fosse um ponteiro, como apresentado em 2.6.3. Devido a essa característica, Cm não verifica o tamanho de vetores no momento da passagem, o que permite usar *partes* de vetores como parâmetro (isto é, o endereço de um elemento que não o primeiro).

  Caso o valor anterior do vetor deva realmente ser preservado durante a execução da função, adiciona-se "[ ]" na chamada.[21]

2.7.3 Apontadores para Funções

Não é possível usar funções como parâmetros de outras funções. Pode-se, no entanto, declarar parâmetros que são *apontadores* para funções. Objetos do tipo apontador para função operam e são declarados como apontadores convencionais: podem ser referenciados, e o resultado dessa indireção executado

```
float (float t; int f, g) * fp;
...
```

[21] ainda não implementado

```
fp = convtemp;
...
... = (*fp) (300.0, 1, 2);
```

Novamente parênteses são necessários devido às regras de precedência de operadores.

Note que o sistema de declaração de tipos de Cm permite interpretar uma declaração quase diretamente numa leitura da direita para a esquerda:

int	inteiro
int *	apontador para inteiro
int * [n]	vetor de ponteiros para **int**
int [n] *	apontador para vetor de **int**
int *[m][n]	vetor de *m* linhas, cada uma contendo *n* colunas de apontadores para **int**
int * (**char** k)	função (parâmetro **char**) retornando apontador para **int**
int * () * [n]	vetor de *n* apontadores para função sem parâmetros retornando apontador para inteiro

Essa regularidade (a exceção são limites de vetores multidimensionais) facilita em muito a declaração e a manutenção de programas em Cm.

2.7.4 Funções em C

Quando é necessário usar em Cm funções já escritas na linguagem C, é indispensável informar ao compilador a sua interface, o que se faz com uma declaração convencional, precedida da qualificação **cvirtual**:

```
cvirtual char * gets (char * buffer);
```

é uma declaração que permite chamadas à função da biblioteca padrão C *gets*, que obtém uma cadeia de caráteres da entrada padrão. Essas chamadas são indistingüíveis de chamadas a funções Cm. Várias declarações **cvirtual** podem inclusive ser agrupadas numa classe e, acrescidas da qualificação **export**, invocadas como usual.

No caso de funções como *printf* [At 86], com número variável de parâmetros, a verificação estrita de Cm tornaria impossível sua declaração. Contorna-se esse problema usando três pontos consecutivos em lugar dos parâmetros formais:

```
cvirtual int printf (...);
```

A notação indica que o compilador não deve fazer qualquer verificação de número e tipos dos parâmetros reais em nenhuma invocação de *printf*, deixando essa

tarefa a cargo do programador. Esse recurso pode ser empregado também em funções declaradas sem **cvirtual**.

2.8 Classes

Em Cm, *classes* são a maneira mais geral de definir novos tipos. Uma classe declara um *tipo abstrato de dados*, englobando dados e os procedimentos que os manipulam de forma encapsulada — isto é, a função e interface do tipo, mas não sua organização ou implementação, são conhecidas publicamente por outros tipos.

2.8.1 Classes Simples

Cada objeto definido usando um construtor **class** (ou instância daquela classe) contém instâncias de todos os dados declarados na classe, podendo sofrer as operações de atribuição e comparação com outros do mesmo tipo. Tanto dados como funções de instância podem ser usados como se fossem membros de estruturas, desde que declarados (na definição da classe instanciada) da classe de armazenamento **export**.

Uma classe muito primitiva pode implementar números complexos:

```
class complex ()
export float re = 0.0, im = 0.0;
```

Uma outra classe declara variáveis cujo tipo é *complex*:

```
import complex;

complex () c1, c2;
```

E como *re* e *im* são objetos públicos em *complex()*, c1 e c2 possuem "componentes" visíveis públicos de nome *re* e *im* e tipo **float**, que podem ser referenciados individualmente:

```
c1.re = c2.im = 1.0;
```

Instâncias de classe não sofrem restrições em relação a objetos criados por outros construtores de tipos. O trecho de programa

```
complex [N] c;
int i;
...
for (i = 0; i < N; i++)
    c [i] = c1;
```

cria um vetor de números complexos, todos com valor igual ao de c1.

Do modo como foi definida a classe, cada instância de *complex* é pouco mais que um depósito de dados. Classes tornam-se interessantes com a adição de operações sobre esses dados:

```
export float rho (c)
{
    return (self.re * self.re + self.im * self.im);
}

export void cube ()
{
    float re2, im2;
    re2 = re * c.re;
    im2 = im * c.im;
    re *= (re2 - 3 * im2);
    im *= (3 * re2 - im2);
}

export complex ntimes (float n)
{
    complex temp;
    temp.re = re * n;
    temp.im = im * n;
    return temp;
}

export void xntimes (float n)
{
    self.re *= n;
    self.im *= n;
}

export complex times (complex c)
{
    complex temp;
    temp.re = re * c.re - im * c.im;
    temp.im = re * c.im + im * c.re;
    return temp;
}
```

As variáveis re e *im* (que existem para *cada* objeto declarado do tipo *complex*)

podem ser usadas em funções da própria classe tanto sozinhas como precedidas por **self**. Note-se que *ntimes* cria um novo objeto *complex* e o devolve a quem chamou a função, onde pode ser usado numa expressão — **self** permanece inalterado; em contraste, *xntimes* apenas altera as variáveis do próprio **self**, sem devolver valor algum.

Uma função na classe importando *complex* pode usar as funções exportáveis desta classe, como em:

```
class C1 ()
...
import complex ();
...
complex () c1, c2;
...
{
        float d = c1.rho () + c2.rho ();
        complex () c3 = c1;
        c3.cube (); // eleva c3 ao cubo
        c1.im += c2.re;
        c2 = c1.ntimes (d); // multiplica o valor de c1 por d; armazena em c2
        c2.xntimes (d); // multiplica c2 por d
        c3 = c1.times (c2);
}
```

Note-se que a especificação de um tipo envolvendo classes somente está completa quando o nome da classe vem acompanhado dos parâmetros reais entre parênteses (*complex* não tem parâmetros). A exceção se faz na própria classe, quando esta faz referência ao próprio tipo (a função *times* em *complex* devolve um objeto do tipo *complex*(), especificado sem parênteses; estes são, por outro lado, necessários na declaração de variáveis da classe *C1*).

2.8.2 Classes Parametrizadas

Cada classe em Cm define um número potencialmente ilimitado de novos tipos de dados, bastando especificá-la com *parâmetros*. Tal como em funções, parâmetros de classe permitem que uma mesma declaração se aplique a uma série de contextos diversos. Entretanto, contrariamente aos parâmetros de função, parâmetros de classe podem ser expressões de *tipos*. Os parâmetros reais de uma classe devem ser constantes durante a compilação.

Árvores tipo B [Kn 73] são úteis numa variedade de circunstâncias para armazenar estruturas de dados dinâmicas com acesso eficiente. Em essência, duas

características diferenciam uma espécie de árvore B de outra: o tipo de dados armazenado (suposto uniforme numa mesma árvore) e a ordem (metade do número máximo de elementos por nó, à exceção da raiz). Uma implementação de árvores B usando Cm definiria uma classe genérica *btree* com dois parâmetros:

```
class btree (type T; int order)
```

(não se preocupa aqui em diferenciar árvores implementadas em memória principal daquelas usando memória secundária). Um nó de árvore B contém uma série de $2 \times order + 1$ apontadores para nós descendentes do mesmo tipo, e um vetor de $2 \times order$ elementos armazenados. Também é necessário um contador indicando quantas posições do vetor estão efetivamente usadas:

```
type node = struct {
                        int used;
                        T [2 * order] info;
                        node * [2 * order + 1] sibling;
                    };
```

À parte dados adicionais para teste, estatísticas, verificação, o único componente essencial para um objeto do tipo *btree()* é um apontador para o nó raiz:

```
node * root = NIL;
```

Não é necessária função de iniciação para objetos definidos pela classe *btree*. A função de busca, *search*, tem como parâmetros apenas o elemento a procurar, devolvendo 1 se obtiver sucesso e 0 em caso contrário. Neste ponto ocorre uma dificuldade, dada a generalidade da classe *btree*: é necessária uma função definindo uma ordem de comparação no tipo T — isto é, as relações $a < b, a = b$ e $a > b$ para dois objetos a e b desse tipo. Ela é particular para cada tipo T e portanto não pode ser definida em *btree*. Na implementação de uma classe em Cm para busca linear tal função não seria necessária, pois a operação de igualdade é suficiente e definida para qualquer tipo em Cm.

Cria-se então um apontador para função *cmp*

```
export int (T ta, tb) * cmp;
```

fazendo-se a rotina de busca invocar a função cujo endereço for colocado em *cmp*:

```
export int search (T what)
{
        node *p = root;
```

```
        while (p != NIL)
        {
            int i, r;
            for (i = 0; i < p -> used &&
                 (r = (*cmp) (what, p ->info [i])) < 0; i++)
                 ;
            if (r == 0)
                 return 1;
            else
                 p = p -> sibling [i];
        }
        return 0;
}
```

Feita a busca, a inserção de elementos não apresenta maiores dificuldades. A criação dinâmica de nós se faz da forma

```
node * p;
...
p = new (node);
```

Uma classe pode definir objetos que são árvores B, desde que declare a classe como importada:

```
import btree;
...
const int size = 10;
type name = char [size];
btree (name) bt1;
btree (long int, 20) bt2;
btree (name) * btp;
...
btp = new (btree (name), 2);
```

Este fragmento de código declara duas variáveis de tipos "árvore B com elementos do tipo *name* e ordem *default*" e "árvore B com inteiros longos, ordem 20", além de um apontador para "árvore B de *name*", apontando para um vetor de duas árvores B. São necessárias duas funções de comparação:

```
int longcmp (long int l1, l2)
{
        return l1 < l2 ? -1 : l1 == l2 ? 0 : 1;
}
```

```
int namecmp (name n1, n2)
{
        int i;
        for (i = 0; i < size; i++)
            if (n1 [i] != n2 [i])
                return (n1 [i] - n2 [i]);
        return (0);
}
```

Após indicar corretamente os endereços de função para cada instância de árvore B

```
bt1.cmp = namecmp;
bt2.cmp = longcmp;
btp -> cmp = namecmp;
(btp + 1) -> cmp = namecmp;
```

as funções e variáveis exportáveis de cada instância podem ser usadas:

```
...
if (! bt1.search ("Nome"))
bt2.insert (1);
```

Essa técnica de declarar funções na classe importadora apresenta a desvantagem de exigir iniciação explícita e é propensa a erros e omissões. Uma solução mais apropriada para o problema de funções genéricas/específicas será apresentada mais adiante.

2.8.3 Herança

Outra forma de explorar a modularidade de classes em Cm é através da herança, que consiste em definir um tipo como contendo as mesmas características de outros, alterando ou adicionando novas propriedades.

A cláusula **inherit** relaciona uma ou mais classes *parametrizadas* (ao contrário do que ocorre com **import**). Todas as variáveis, constantes e funções declaradas nas classes relacionadas passam a fazer parte também da classe sendo definida.

Um exemplo simples, a classe *npair*, define um tipo e operações básicas para um objeto capaz de conter dois valores numéricos:

```
class npair (type T)
T v1, v2;
export void init (T V1, V2)
```

```
{
        v1 = V1; v2 = V2;
}

export T c1 ()
{
        return v1;
}
...
export npair plus (T addto)
{
        npair temp;
        temp.v1 = v1 + addto.v1;
        temp.v2 = v2 + addto.v2;
        return temp;
}

export npair times (T multiplier)
{
        npair temp;
        temp.v1 = v1 * addto.v1;
        temp.v2 = v2 * addto.v2;
        return temp;
}
```

A função c1 devolve o primeiro componente do par e é usada apenas para fins de abstração (em lugar de tornar *v1* exportável). É possível agora uma nova definição para a classe *complex*:

```
class complex ()
inherit npair (float)

export complex times (complex multiplier)
{
        complex temp;
        temp.v1 = v1 * multiplier.v1 - v2 * multiplier.v2;
        temp.v2 = v1 * multiplier.v2 + v2 * multiplier.v1;
        return temp;
}
```

Embora a operação de soma implementada pela função *plus* em *npair* seja adequada à definição de soma de números complexos, considerando que *v1* e *v2*

representam as componentes real e imaginária, respectivamente, o mesmo não se aplica à operação de multiplicação, sendo exigida sua redefinição. A partir da nova declaração da classe *complex*, outras classes podem utilizá-la como anteriormente (mudando apenas as referências a *re* e *im*; como fazer isso na própria *complex* será visto adiante). Caso fosse criada uma classe *rational* implementando números racionais, *npair* também poderia ser herdada, atribuindo a *v1* o papel de numerador e a *v2* o de denominador:

```
class rational ()
inherit npair (int);

int mmc (int i, j)
{
        ...
}

export rational plus (rational toadd)
{
        rational temp;
        temp.v2 = mmc (v2, toadd.v2);
        temp.v1 = v1 * temp.v2 / v2 + toadd.v1 * temp.v2 / toadd.v2;
        return temp;
}
```

Nesta classe a função *times* herdada se aplica diretamente, sendo requerida a redefinição de *plus*. Deve ser notada em ambas as classes uma redefinição implícita também do tipo dos parâmetros e do valor devolvido pelas funções *times* e *plus*.

Herança Múltipla e Rename

Mais de uma classe pode ser herdada simultaneamente (bem como uma classe A pode herdar e importar B), o que dá margem a conflitos quando o mesmo identificador é definido em mais de uma classe herdada. Cm sobrepõe declarações, isto é, cada definição numa classe parametrizada prevalece sobre todas as anteriores. Caso seja necessário usar definições homônimas provindas de classes e/ou parametrizações diferentes, aplica-se a declaração **rename**. Ela é usada após uma classe parametrizada e introduz sinônimos para os nomes nela declarados:

```
inherit c1 (...) rename x1 = x, c2 (...);
```

se x é implementado tanto em *c1* como em *c2*, a classe herdeira pode usar *x* para referir-se à versão em *c2*, e *x1* quando se refere àquela em *c1*. Se houvesse uma

terceira versão de *x*, redefinida na própria classe herdeira, um segundo **rename** logo após os parâmetros de *c2* lhe permitiria continuar usando a versão desta última classe.

Além de, neste último caso, viabilizar a existência simultânea de mais de uma variável ou função homônimas implementadas em classes ou parametrizações diferentes, **rename** possibilita que ambas sejam exportadas para uma eventual classe importadora, adicionando a qualificação **export** antes do novo nome:

```
class c ()
inherit c1 (...) rename export fbase = f;
...
export void f (...
```

A classe *c* provê duas versões exportáveis da função: *fbase*, implementada em *c1*, e *f*, local e provavelmente mais especializada.

A terceira aplicação de **rename** é aumentar a clareza do programa, usando nomes mais apropriados ao contexto de cada classe. Embora *npair* seja útil para definir complexos, racionais, coordenadas polares/cartesianas e vários outros conceitos, o par de nomes (v1, v2) torna-se realmente pouco descritivo nas classe herdeiras. A cláusula **rename** dá a possibilidade de se usar nas novas classes nomes como (re, im), (numerador, denominador), (x, y), (rho, phi), entre outros, mais claros e apropriados.

Funções Virtuais

Há ocasiões em que a generalidade de uma classe poderia impedir sua completa implementação: por natureza, certas tarefas somente são totalmente especificadas conhecendo-se detalhes dos tipos sobre os quais elas são aplicadas. Isso significa que, ao contrário do que vem sendo apresentado até agora, pode ser necessário que uma classe básica requeira definições presentes em classes derivadas.

Em Cm, essa capacidade é obtida permitindo-se que funções numa classe herdada invoquem funções definidas nas herdeiras. O qualificador **virtual** pode preceder uma declaração de função sem o corpo, indicando que ela será definida apenas por alguma das classes herdeiras, de forma inversa ao conceito original de herança/importação. Um exemplo simples é provido pela classe *npair*. Supondo que seja necessário depurar algumas de suas funções, deseja-se uma subrotina *dump()* que imprima o valor dos dois componentes de uma instância de *npair*, como em:

```
...
export npair times (npair multiplier)
{
```

```
        npair temp;
        cvirtual int printf (...);
        printf ("Value of self is ");
        dump ();
        printf ("Value of parameter is ");
        multiplier.dump ();
        ...
        printf ("Return from times ");
        temp.dump ();
        return temp;
}
```

A função *dump()* não pode ser definida em *npair* pois nela não se conhece o tipo dos componentes *v1* e *v2*. Assim, uma declaração como

```
virtual void dump ();
```

é incluída, precedendo qualquer outra menção a *dump()*. A classe *rational* poderia definir a função, como em

```
class rational ()
...
inherit npair (int) rename num = v1, den = v2;
...
void dump ()
{
        cvirtual int printf (...);
        printf ("(%d/%d)", num, dem);
}
...
```

do mesmo modo como *complex()* teria também sua definição particular de *dump()*.

Declarações virtuais tornam possível uma implementação mais clara de classes como *btree()*. Ao invés de importar diretamente parametrizações de *btree()* e iniciar explicitamente o apontador da função comparadora, é preferível tornar esta última virtual e definida numa classe herdeira intermediária, como em *strbtree* (árvores-B de *strings*):

```
class btree (type T; int order)
...
virtual int cmp (T v1, v2);
...
export int search (T what)
{
```

```
        ...
        i = cmp (...
}
...

class strbtree (int size, order)
inherit btree (char [size], order);
...
int cmp (char * s1, s2)
{
        int i;
        for (i = 0; i < size; i++)
            if (s1 [i] != s2 [i])
                return s1 [i] - s2 [i];
        return 0;
}
...
class c1 ()
...
import strbtree;
...
strbtree (50, 10) bt1, bt2;
```

2.8.4 Variáveis de Classe e de Instância

Todas as declarações de variáveis vistas até agora declaram objetos particulares de cada instância de classe criada. Há contudo circunstâncias em que é necessário ou desejável compartilhar variáveis entre diferentes instâncias da mesma classe. A classe de armazenamento "+", já vista como forma de declarar variáveis "estáticas" a uma função, tem outro significado quando aplicada a itens externos a funções: ela denota variáveis *de classe*,[22] criadas uma única vez para todos os objetos cujo tipo é dado por aquela classe e parâmetros. Uma aplicação muito simples dessa idéia é determinar quantas instâncias já foram criadas:

```
class stack (type T = int; int n = 100)
...
+ int count = 0;
int mynumber;

export void tellsize ()
```

[22] em contraposição às variáveis vistas até agora, ditas *de instância*

```
{
        cvirtual int printf (...);
        printf ("Stack %d (of %d): %d elements\n",
                mynumber, count, size);
}

void main ()
{
        mynumber = ++count;
}
```

Neste caso, a variável *count* é única e acessível a todas as instâncias de *stack*, para dados T e n. Isto significa que o *count* compartilhado pelas instâncias de *stack* (int) é único e *diferente* daquele das instâncias de *stack* (**char**), por exemplo.

2.8.5 A Cláusula Use

Quando é necessário compartilhar uma mesma variável entre diversas instâncias de classes diferentes ela pode ser declarada na cláusula use, que enumera diversos objetos, que serão únicos para todo o programa. Use também pode ser aplicada para que diversas instâncias referenciem o mesmo objeto sem a necessidade de passá-lo como parâmetro de função. Sua forma geral é uma lista de nomes e tipos, como em

```
use ident1 = int, ident2 = btree (long int);
```

2.8.6 Hierarquia de Classes e "Classe Principal"

Programas em Cm são uma coleção de classes, unidas por relações ortogonais de importação e herança.

A execução do programa é implicitamente iniciada na função *main()* da classe de "mais alto nível", isto é, aquela que não é importada ou herdada por nenhuma outra.

Deve ser ressaltado que a função *main()* é executada para cada instância criada daquela classe — portanto, nem ela nem nenhuma das funções por ela invocadas devem declarar variáveis locais cujo tipo é a própria classe, tampouco solicitar via new a criação de objetos desse tipo.

Capítulo 3

Cm: Implementação

'O me, what hast thou done?'
W. Shakespeare, *Hamlet*

Aspectos particulares da implementação atual da linguagem Cm são apresentados, desde o ambiente de desenvolvimento até uma descrição simplificada de cada módulo do programa -- um tradutor em um passo de Cm para C.

3.1 Considerações Gerais

A primeira implementação prática da linguagem de programação Cm foi planejada como um tradutor, ou seja, um compilador cujo resultado não é código-objeto ou executável, mas sim código-fonte em outra linguagem. Esta abordagem, chamada preprocessamento, pode ser adotada com sucesso para aumentar a clareza ou poder de expressão de uma linguagem, tendo sido empregada em diversos projetos:

- Ratfor, ou *Rational Fortran* [KP 84][Ke 75] é uma linguagem de programação que adiciona construções de programação estruturada à linguagem Fortran [Ho 84]. Os comandos básicos são mantidos, e o "compilador" quase que apenas traduz padrões de texto de uma linguagem para outra (erros do programador passam desapercebidos e sua detecção é deixada para o compilador Fortran)
- *Troff* [KP 84] é um poderoso sistema de composição de textos e gráficos para o sistema *UNIX*. Todavia, sua linguagem de especificação de textos, embora extremamente flexível e programável, é demasiado básica e complexa, demandando muito tempo de estudo mesmo para produzir trabalhos simples. Para simplificar seu uso, foram desenvolvidos diversos

pacotes de macros, como *me* e *ms*, contendo os elementos mais freqüentes em documentos comuns. Macrocomandos não são no entanto suficientes para definições complexas de figuras, tabelas e equações. Esses ambientes exigiram a criação de três linguagens de especificação chamadas respectivamente *pic* [Ke 82], *tbl* [KP 84] e *eqn* [KC 75], com processadores próprios que geram entradas para *troff*

- o interpretador de comandos do sistema *UNIX* (*shell*) permite escrever rapidamente *scripts* relativamente complexos. Em [Car89] se descreve um tradutor de *C shell* para C.

- programas em Pascal escritos para diversas implementações da linguagem ou ambientes de execução são difíceis de manter. Filtros simples como o descrito em [Bra87] traduzem um programa em Pascal "genérico" produzindo um programa para um compilador Pascal específico

- PSAIL [Le 88] é parte da linguagem de programação de sistemas SAIL. É um subconjunto funcional para o qual foi desenvolvido um tradutor gerando código C

- *Yacc* [Jo 78] é um gerador de analisadores sintáticos *(parsers)*, que facilita a produção de compiladores e outros processadores de texto. Yacc transforma uma gramática especial em declarações C acopladas a uma função que implementa um analisador sintático

- *Lex* [Aa 86] é uma espécie de companheiro de Yacc, projetado para construir automaticamente analisadores léxicos em C, a partir de especificações de linguagens regulares

- a entrada para Yacc pode ser também pré-processada por programas como *y+* [Pa 88], que abreviam ainda mais o esforço de programação para contextos específicos

- *Turing Plus* [PC 88] também foi implementada como um tradutor, criando código em C

- a linguagem *Eiffel*, orientada a objetos, permite uma integração a C e pode ser parcialmente implementada com pré-processamento para essa linguagem [MNM87]

- a proposta para programação orientada a objetos *Objective* C [Co 87] altera extensivamente o estilo de programação da linguagem original; é um pré-processador para C

- *cfront* é um tradutor que implementa a linguagem C++ convertendo-a em C. C++, originalmente chamada C *with classes*, nasceu de uma proposta mais modesta baseada inteiramente em pré-processamento [St 82]. Embora haja atualmente compiladores específicos para C++, *cfront* ainda é extensivamente usado e define o padrão da linguagem [Su 89]

Já se chamou C de "linguagem de montagem portável", em vista de sua adequação como linguagem-objetivo de tradutores. Infelizmente o uso de C não está livre de problemas, como visto mais adiante.

No caso de Cm, também foi adotada C como linguagem-alvo, dada uma série de vantagens, além do fato de ser a base para o conjunto de comandos em Cm. É uma linguagem para a qual diversos compiladores estão disponíveis em vários ambientes e sistemas operacionais (em particular, é o único compilador certamente disponível em qualquer versão do sistema *UNIX*). Por outro lado, seu sistema de tipos relativamente modesto permite que o compilador Cm gere construções pouco ortodoxas, sem verificação de tipos,[1] mas necessárias ao polimorfismo da linguagem.

Desta forma, torna-se Cm disponível para qualquer ambiente de programação onde haja um bom compilador para C, sem a necessidade de conhecer diversas arquiteturas de máquinas e sistemas operacionais. Somente se requer que o código gerado pelo compilador seja portável, compatível com os diversos dialetos de C existentes. O mesmo se exige da atual biblioteca de suporte de execução — é uma pequena coleção de rotinas escritas em C que devem ser automaticamente agregadas ao programa final para permitir sua execução.

Razões semelhantes de portabilidade determinaram C também como a linguagem de implementação do compilador propriamente dito. Outro fator foi a disponibilidade de versões do programa Yacc. O gerador Lex é comumente associado ao uso de Yacc, mas não foi utilizado no compilador Cm.

Pelo menos dois sistemas operacionais foram escolhidos como ambiente de desenvolvimento e execução do compilador — *UNIX* e MS-DOS. O sistema *UNIX* provou ser bem mais adequado a projetos do porte de Cm; a versão corrente foi completada nas implementações *CLIX* (Intergraph, System V.3) e SunOS (Sun Microsystems, BSD). O sistema MS-DOS, projetado para arquiteturas mais simples, impôs maiores restrições, tanto ao compilador como ao código gerado. O compilador usado foi o Turbo C [Bo 87].

Requer-se também que o código-fonte em C seja o mesmo para todas as versões, independentemente de arquitetura ou sistema operacional. O transporte para outros sistemas *UNIX*, ou outros compiladores em MS-DOS como Microsoft C [Mi 87], não é difícil, dada a modularidade do projeto.

O compilador foi originalmente imaginado como parte integrante do ambiente de desenvolvimento e execução de programas A_HAND [DL 87], e como tal,

[1] já examinados durante a compilação Cm

dependente do ambiente para uma série de serviços, como a interface do usuário e a determinação, através de um banco de dados especial, das instâncias de classes previamente compiladas e seus parâmetros. Entretanto, embora seja certa sua integração ao ambiente, o compilador é um sistema autônomo, exigindo apenas o sistema de compilação C para compor programas Cm executáveis. Embora essa independência tenha exigido diversas adições ao projeto inicial, ela trouxe pelo menos outro benefício — sua utilização no sistema MS-DOS, não previsto pelo ambiente A.HAND.

3.2 Módulos do Compilador

Para maior clareza e facilidade de implementação, o compilador foi dividido em várias partes, ou módulos. Procurou-se dar a essas partições funções bastante específicas, de maneira a reduzir ao máximo a interface entre diferentes módulos, e assim permitir a escrita e depuração de cada um em separado, se possível independentemente dos demais.

De modo resumido, segue-se a descrição dos componentes do compilador:

gram O analisador sintático, responsável pela verificação das construções em Cm. Invoca a maioria dos outros módulos como sub-serviços.

analex Contém o analisador léxico, que lê de um arquivo o código-fonte Cm, separando-o em palavras ou símbolos (*tokens*)

cm O módulo de mais alto nível, interface com o usuário. Determina as opções de compilação e invoca a análise sintática, geração de código e o compilador C.

code O gerador de código, traduz estruturas internas descrevendo código Cm em construções na linguagem C.

debug Módulo auxiliar para depuração das estruturas internas do compilador. Implementa o acompanhamento das funções do programa em vários níveis de detalhamento. Pode ser totalmente removido através de compilação condicional.

decl Rotinas de declaração. Introduz estruturas descrevendo os identificadores Cm.

environ Lida com o sistema operacional, ajustando detalhes de operação.

error Anuncia eventuais erros ao usuário.

expfn Expande nomes de arquivos para uma forma canônica. Muito dependente de sistema operacional.

expr Constrói e manipula estruturas de dados descrevendo expressões da linguagem Cm; colabora em escala reduzida na tradução para C.

file Inclui operações genéricas sobre arquivos e diretórios.

filer Gerenciador de arquivos de acesso indexado.

llist Funções para listas ligadas genéricas (estrutura muito usada por todo o programa).

make Produz o *makefile*, arquivo auxiliar para a compilação final do programa em C.

mem Controle de memória dinâmica.

symtab Tabela de símbolos: rotinas para inserção e busca dos identificadores nos programas Cm.

type Funções do sistema de tipos: construção e verificação.

utils Pequenas rotinas especializadas, desvinculadas de outros módulos.

wildcard Expansão de nomes de arquivos genéricos. Essencial às buscas no banco de dados de instâncias compiladas e/ou traduzidas.

A divisão não obedece a critérios rígidos. Por exemplo, **wildcard** e **expfn** poderiam ser incorporados tanto a **file** como a **environ**, mas são isoladas porque podem facilmente ser usadas novamente em outros projetos, assim como **llist**. O mesmo se aplica a **filer** em relação a **file**.

3.3 Visão Geral da Tradução

Devido ao caráter polimórfico da linguagem, a compilação de um programa Cm (isto é, a sua tradução para um ou mais programas equivalentes em C) oferece características especiais. Cada classe importada, herdada ou usada por outra pode ser instanciada diversas vezes, com diferentes parâmetros, caracterizando uma forma de polimorfismo paramétrico.

A classe *stack*, por exemplo, tem como propriedades essenciais o tipo de dados armazenados e a capacidade máxima de uma instância: são esses os dois parâmetros da classe, definida em linhas gerais como

```
class stack (type T = int; int n_elementos = 50)

T [n_elementos] lista;
int topo = 0;
```

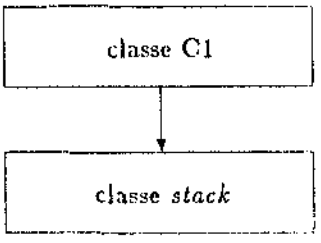

Figura 3.1: Exemplo de hierarquia de classes Cm

```
...
export int push (T objeto)
{
  if (topo >= n_elementos)
    return -1;
  lista [topo++] = objeto;
  return 0;
}

export T pop ()
{
  if (topo > 0)
    return lista [--topo];
  else
...
```

O modo de conseguir esse polimorfismo é produzir, para cada classe Cm, diferentes versões em C, uma para cada conjunto diferente de parâmetros com que a classe é instanciada. Essencialmente, um arquivo contendo um programa-fonte Cm pode ser visto como um "macro-comando" ou gabarito (*template*), da qual se tiram diversas cópias, cada uma semelhante à anterior nos aspectos gerais, diferindo apenas nos itens que foram parametrizados em Cm.

Vejamos o que ocorre caso uma certa classe *C1* use de alguma maneira instâncias da classe *stack*, por exemplo, via importação:

```
class C1 (...)
...
import stack;
```

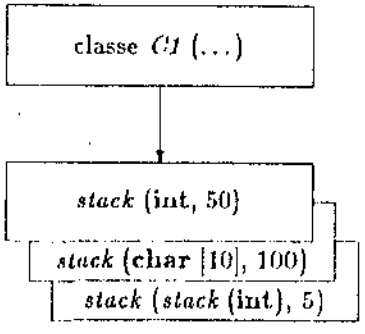

Figura 3.2: Hierarquia de instâncias de classes

Ocorre uma hierarquia de classes muito simples, como na figura 3.1, na qual estão mostradas as relações de dependência entre duas classes Cm (do ponto de vista de implementação, dois arquivos-fonte com código Cm).

O uso *efetivo* de *stack* por *C1* acontece quando esta classe declara objetos (variáveis, parâmetros ou funções) usando tipos derivados de *stack*:

```
...
stack (int, 50) stack1;
stack (char [10], 100) stack2;
stack (stack (int), 5) stack3, stack4;
...
stack1.push (strlen (stack2.pop ()));
...
```

Com a declaração destes quatro objetos, são requeridas três diferentes versões de pilha, formando uma segunda hierarquia, desta vez indicando para a classe *stack*, os parâmetros das versões, mostrados na figura 3.2. Cada uma das três referências a *stack* indica que um novo arquivo com fonte C é gerado a partir do arquivo-fonte escrito em Cm.

Note que a classe *C1* poderia igualmente por sua vez ser parametrizada, embora isso seja irrelevante para o exemplo.

Para que as variáveis *stack1*, *stack2*, *stack3* e *stack4* em *C1* sejam declaradas e, além disso, haja uma verificação completa dos tipos nas expressões onde elas ocorrem, é necessário que o compilador Cm interrompa a tradução de *C1* imediatamente após encontrar uma referência a instâncias de outras classes. Nesse

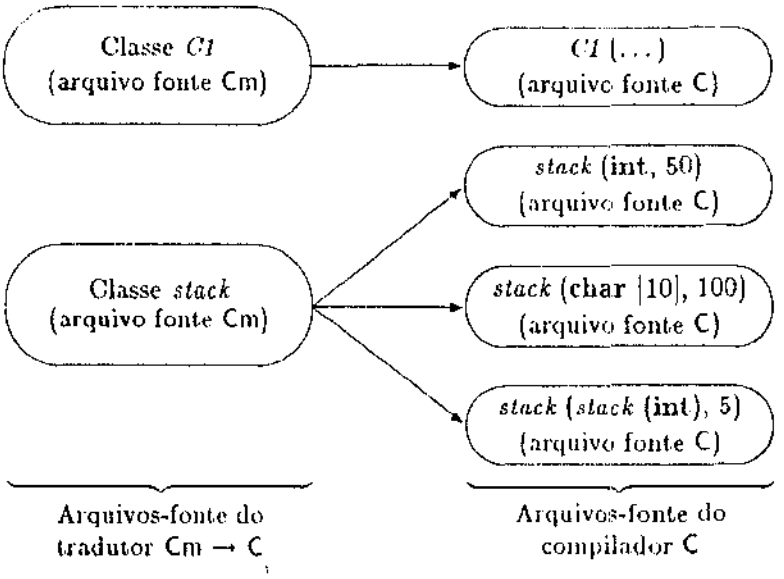

Figura 3.3: Principais arquivos na compilação de *C1* e *stack*

ponto, deve guardar o contexto de compilação (isto é, o ponto de interrupção, toda a informação até então coletada sobre a classe corrente, os identificadores conhecidos, etc.) e passar a compilar um novo arquivo-fonte Cm. Quando a compilação dessa subclasse estiver concluída, o tradutor pode retornar à classe original; nesse momento, é conhecida a interface da subclasse. É o conjunto de objetos exportáveis, necessária à verificação de tipos sobre os objetos cujo tipo é aquela instância da subclasse.

Portanto, a cada referência a uma classe instanciada, o compilador Cm muda de arquivo-fonte, a menos que aquela instância tenha sido usada anteriormente; neste caso, sua interface já é conhecida. É claro que o processo pode ser repetido indefinidamente, caso a hierarquia tenha maior número de níveis. No último exemplo, poder-se-ia implementar uma pilha usando listas ligadas: esta seria agora a classe de menor nível.

Os arquivos produzidos pelo compilador também estão sujeitos a interrupções (cada subclasse incluída pode implicar num novo arquivo-fonte C criado, como na figura 3.3), suspendendo temporariamente a confecção do código corrente, retomada juntamente com o término da compilação da subclasse.

Finalmente, o compilador Cm deve prover alguma facilidade para invocar a

compilação C dos arquivos produzidos, além da ligação (*linking*) dos arquivos em código-objeto criados pelo compilador C. O tradutor Cm tem as informações necessárias para requerer a compilação de *todos* os arquivos C exigidos para criar um programa executável completo. Alguns dos arquivos C poderiam ter existido antes da compilação da classe corrente; entretanto, caso a classe Cm fosse alterada, a recompilação seria obrigatória para manter a consistência do programa. Por outro lado, não é necessário recompilar um código-fonte C caso o objeto exista e o código Cm não tenha sido modificado. Estas funções de manutenção de programas executáveis são típicas do programa *make* [Fe 79][At 89] e foram incorporadas ao compilador. Cada subclasse usada pode implicar numa série de instâncias e arquivos C diferentes produzidos. Por sua vez, cada instância é um tipo e, como tal, tem potencialmente uma série de variáveis ou parâmetros associados (no exemplo, a última instância de *stack* declarava duas variáveis). Deste modo, as variáveis de uma classe instanciada (*lista* e *topo*, no exemplo) devem ser replicadas de alguma maneira para todos os objetos cujo tipo é aquela instância. De modo análogo, cada função original tem como conseqüência diversas funções em C— e todas, não apenas as exportáveis, podem ser produzidas. Essa questão particular à técnica de pré-processamento — um identificador na linguagem compilada deve ser mapeado para mais de um identificador na linguagem-objetivo — introduz problemas críticos cuja resolução é essencial à implementação de Cm.

O método de criação de diferentes arquivos na linguagem final para obter polimorfismo pode lembrar a definição de macros genéricas, como a clássica implementação de *quicksort()* genérico em C. Na verdade, o próprio pré-processador para C é útil para simular classes polimórficas [Gru86]. Contudo, essa técnica (extensível a C++) não suporta verificação de tipos, nem impede colisões de nomes.

O compilador foi inicialmente imaginado operando em mais de um passo, ou passagem pelo mesmo código-fonte. O passo preliminar não geraria código algum, mas coletaria todas as referências a instâncias de classes, bem como todas as declarações antecipadas de tipos. Essa passagem seria executada para todas as classes envolvidas, determinando a ordem dos arquivos compilados no segundo passo. Apenas então seriam criadas as traduções C. A técnica de múltiplos passos permitiria uma linguagem ligeiramente mais flexível quanto à declaração de tipos, evitando ao mesmo tempo algumas das dificuldades advindas da mudança de arquivo-fonte *durante* a compilação, devido ao uso de Yacc. Contudo, a complexidade de armazenar em memória permanente o máximo possível de informação entre passos (de forma a reduzir o esforço do tradutor após o primeiro passo) seria alta, por causa das estruturas de dados complexas empregadas pelo compilador.[2]

[2] na implementação atual, diversos dados *são* colocados em disco durante mudanças de classe compilada e posteriormente recuperados, mas em escala muito menor que a necessária numa

3.4 O Analisador Léxico

Implementado no módulo **analex**, o analisador léxico tem por função ler o texto de arquivos-fonte e separar os elementos da linguagem (identificadores, números, palavras reservadas, pontuação, etc.), conhecidos coletivamente como "átomos" ou *tokens*, codificá-los e passá-los à função responsável pela análise sintática. Essa separação nítida de encargos é muito comum na implementação de compiladores e outros processadores de texto, dadas as vantagens que oferece:

- o analisador sintático pode ser simplificado, caso estejam concentrados em uma só rotina todos os serviços de análise léxica. Se bem que muito mais simples que a análise sintática, uma tarefa especificada de modo tão simples como "encontrar o próximo átomo" demanda particularidades:
  - o texto é lido de um arquivo (ou, mais raramente, diretamente do teclado) usando *buffers*;[3] o analisador pode ocultar detalhes de gerência de arquivos e *buffers* do resto do programa
  - caráteres de separação devem ser descartados; estes incluem espaços em branco, tabulação e comentários
  - em certos casos torna-se necessário conhecer mais que o próximo átomo, de forma antecipada (*lookahead*).
- algumas funções podem ser rotineiramente executadas durante a análise léxica, como imprimir linhas numeradas do texto (para conferência do usuário), verificação preliminar de identificadores conhecidos, e eventual tratamento de alguns erros
- como em muitos outros problemas, as funções de entrada/saída de dados consomem tempo substancial da execução de um compilador. Sendo o único ponto de leitura de dados do programa, o analisador léxico pode ser implementado de forma mais eficiente, se bem que não tão portável, que o restante do compilador
- pode ser usado um *gerador de analisadores léxicos*, um programa que processa uma série de especificações sobre os átomos permissíveis à linguagem compilada, produzindo uma função que implementa um analisador apropriado. O mais conhecido desses processadores é provavelmente *Lex* [At 89], inicialmente implementado para o sistema *UNIX*

A possibilidade de geração automática de analisadores, em particular, é bastante atraente. Usando-se Lex, por exemplo, todos os *tokens* possíveis, e outras

tradução multi-passo

[3] memória auxiliar

características da linguagem, como espaços em branco, são representados por expressões regulares [Aa 86], juntamente com ações a executar e códigos a passar ao *parser* quando um átomo é reconhecido. Quando essa descrição é processada por Lex, o resultado é uma função (*yylex()*) na linguagem C implementando um autômato finito e uma tabela; cada chamada à função deve reconhecer o próximo *token* disponível.

Funções geradas por Lex prestam-se sobremaneira a serem integradas com analisadores produzidos pelo gerador Yacc—de fato, Yacc supõe sempre que o analisador léxico tem o nome *yylex()*. Além disso, outras variáveis e funções de código Lex podem ser igualmente usadas por Yacc.

Entretanto, apesar de Lex permitir a criação muito rápida de analisadores simples, também apresenta desvantagens:

- o autômato gerado pode ser notoriamente ineficiente em tempo e espaço de memória utilizados; além disso, tanto a eficiência como a correção do autômato geralmente dependem muito da ordem das especificações (em certos casos, da "intuição" do programador)

- o autômato usualmente obtém o texto a ser lido apenas do teclado (isto é, da "entrada padrão" do sistema operacional). Direcionar a leitura de um arquivo em disco é muito simples, mas modificar arbitrariamente o arquivo processado e depois retornar ao anterior sem perder o contexto de leitura não é tão trivial

- freqüentemente a mesma palavra ou categoria de palavras ocasiona ações diferentes quando em contextos diversos. Quando essa ação ocorre a nível léxico, a distinção é feita definindo-se "estados" de execução, que normalmente reduzem em muito a clareza do código

Em vista de dificuldades como essa, decidiu-se produzir o analisador léxico inteiramente em C, sem usar Lex.[4] Foram porém tomados cuidados para compatibilizar o analisador com o *parser* criado por Yacc.

Como o compilador Cm pode eventualmente traduzir diversas classes numa única sessão, é necessário que a versão de *yylex()* implementada seja capaz de ler vários arquivos, num esquema de pilha. Por exemplo, uma função especial requer que o arquivo corrente seja "empilhado", passando o compilador a ler dados de outra fonte. Quando ocorre um erro, ou o segundo arquivo é exaurido, volta-se à fonte original. Na implementação atual, o arquivo empilhado não é "fechado" enquanto se processa o incluído; isto é, continua disponível para

[4] deve ser ressaltado que as duas últimas propriedades *podem* ser obtidas com um analisador criado por Lex, um programa específico mas de flexibilidade incomum; o uso de Lex é possível, mas o código seria provavelmente tão ilegível que não compensaria a economia de tempo proporcionada pelo gerador

leitura. Isso pode ocasionar problemas dependentes de sistema operacional[5] se a profundidade da hierarquia de classes é muito grande. Entretanto, não é difícil modificar esse esquema e fechar cada arquivo antes de "abrir" o próximo: apenas se aumenta ligeiramente o contexto a armazenar e o tempo de processamento, necessário para reposicionar o arquivo anterior quando se termina o corrente.

O analisador léxico incorporado ao tradutor executa algumas funções "administrativas", como memorizar o texto lido recentemente para relatar mensagens de erro (com algumas precauções devido às freqüentes mudanças de arquivo). Também interage com o gerenciador da tabela de símbolos para uma análise preliminar dos identificadores encontrados e pré-processa constantes.

3.5 Análise Sintática

Enquanto a análise léxica se ocupa das diferentes "palavras" (isto é, átomos ou *tokens*) compondo um programa numa determinada linguagem, o analisador sintático observa como essas palavras estão dispostas em "frases" ao longo do programa. Na verdade, a fronteira entre os dois tipos de análise pode ser bastante rarefeita, mas concorda-se que a verificação sintática abrange estruturas maiores e mais complexas que a análoga léxica. O analisador sintático se encarrega de conferir a estrutura geral do programa; no caso do compilador Cm, é o maior módulo, responsável pela invocação de quase todos os demais.

Normalmente, a sintaxe de uma linguagem é expressa em termos de gramáticas formais. Trata-se de fórmulas em que são descritas de modo preciso quais as seqüências de átomos permitidas pela linguagem, normalmente usando regras de reescrita. Por exemplo, "um rótulo é composto por um identificador seguido por ':'". O uso de gramáticas facilita a especificação e atualizações da linguagem.

Linguagens, sejam ou não de programação, sofrem diversas classificações. Uma das mais úteis [HU 69] define a complexidade necessária a um analisador para a linguagem. A maioria das linguagens de programação se baseia na categoria das linguagens *livres de contexto*,[6] reconhecíveis pelos chamados autômatos *a pilha*. Para essa classe de linguagens diversas técnicas de análise eficiente já foram estudadas; infelizmente, a maioria das linguagens de programação não é realmente livre de contexto, não no sentido de serem totalmente reconhecíveis por um autômato a pilha.

O gerador de analisadores sintáticos Yacc, originalmente criado para o sistema *UNIX*, processa uma gramática contendo regras de reescrita (produções) seguidas por trechos de código C, produzindo como resultado uma subrotina em

[5] *normalmente*, não mais que 20 arquivos podem estar disponíves simultaneamente a um programa em MS-DOS; técnicas não-convencionais ampliam esse limite. Um limite em *UNIX* também existe, mas é mais flexível [KP 84]

[6] *independentes de contexto* seria uma tradução melhor para *context-free*

C. Essa função, chamada convencionalmente *yyparse()*, deve ser compilada juntamente com um analisador léxico (e possivelmente funções auxiliares) para compor um programa destinado a reconhecer a linguagem descrita pela gramática. Cada invocação de *yyparse()* procura reconhecer um texto completo — o valor retornado indica se houve ou não sucesso. À medida que cada regra é reconhecida, a ação (em C) associada a ela é executada. Finalmente, Yacc não faz qualquer suposição sobre o analisador léxico — desde que este obtenha os átomos e os armazene em variáveis pré-determinadas, os dados podem vir de qualquer fonte.

Essa flexibilidade tornou Yacc um programa muito popular, sendo usado na implementação de interpretadores, processadores de texto e todo tipo de compiladores.

A gramática usada como entrada para Yacc deve pertencer à classe LALR: o analisador gerado procura reconhecer a linguagem usando no máximo um *token* de *lookahead*; caso isso não seja possível, o programador é advertido da presença de ambigüidades (conflitos). Um conflito ocorre quando há mais de uma forma de um fragmento de texto ser interpretado. Por exemplo, o trecho

$$a + b * c$$

poderia ser tratado tanto como a soma de a ao produto bc como a multiplicação de c pela soma $a + b$. Esse tipo de conflito é fácil de ser diagnosticado e resolve-se considerando + e * como operadoresoperadores , associando-se precedências diferentes a cada um deles. Normalmente se atribui uma maior precedência ao operador *, resolvendo este caso particular. Quando dois operadores de igual precedência estão adjacentes, como em

```
a + b + c
```

ocorre outra forma de conflito. Para resolver qual dos operadores tem prioridade (caso '+' denote a adição usual, a questão é irrelevante, o que não aconteceria se o operador indicasse, por exemplo, indireção de apontadores), o conceito de *associatividade* é introduzido. No caso, + é associativo *à esquerda*, de modo que o operador da esquerda tem precedência sobre o da direita.

Yacc admite que a precedência e associatividade dos *tokens* para operadores sejam definidas pelo programador. Ainda assim, a gramática pode padecer de conflitos, classificados em

shift/reduce ou s/r; uma regra já foi completada, mas pelo menos um átomo seguinte também poderia fazer parte dela. O analisador não pode determinar onde termina a produção

reduce/reduce ou r/r: uma regra pode ser interpretada de duas formas distintas.

O gerador Yacc poderia considerar uma gramática conflituosa como incorreta, uma vez que o comportamento do analisador não está completamente especificado. Entretanto, mesmo assim *yyparse()* é produzida; ações especiais são definidas para as regras com conflitos:

- para conflitos s/r, a ação escolhida pelo *parser* consiste em "estender" a regra (isto é, procura-se uma seqüência de átomos maximal que satisfaça a produção). Esse procedimento coincide com a interpretação correta da maioria das construções recursivas comumente encontradas em linguagens de programação, de modo que conflitos s/r (uma vez analisados) são quase sempre toleráveis.

- conflitos r/r são resolvidos pela ordem das produções. Nesse caso, torna-se arriscado supor que a interpretação adotada por Yacc é a correta. Geralmente conflitos r/r indicam erros na formulação da gramática ou que a linguagem tal como definida não pode ser corretamente processada, para todas as entradas possíveis, por um *parser* de Yacc.

Freqüentemente conflitos são eliminados reescrevendo algumas produções. Mesmo essa solução, todavia, pode não ser a melhor, caso conduza a regras de compreensão obscura e execução menos eficiente que as originais. Yacc opcionalmente cria um arquivo descrevendo os estados do autômato implementado por *yyparse()*, o que pode sugerir como solucionar possíveis conflitos.

A partir da gramática inicial da linguagem Cm, produziu-se uma entrada para Yacc (reproduzida em forma simplificada no apêndice D). A complexidade advinda de modificações para evitar conflitos pode ser vista, entre outros pontos, nas produções referentes a comandos e expressões.

Outras dificuldades na implementação do analisador são mencionadas a seguir:

- as várias formas de declaração de tipo em Cm criaram conflitos r/r somente retirados com a adição de *lookahead* e verificação de símbolos ao analisador léxico. Os átomos PIDENTIFIER e IDENTIFIER, por exemplo, referem-se ambos a nomes (identificadores), mas o primeiro é retornado por *yylex()* apenas quando seguido de um parêntese esquerdo. Similarmente, CLASS_ID, TYPE_ID e FUN_ID são retornadas em lugar de IDENTIFIER quando um nome idêntico ao lido por *yylex()* já foi usado e declarado como classe, tipo ou função, respectivamente.

  Alguns desses artifícios acarretam em contrapartida outros problemas. Por exemplo, a produção que define membros de estruturas (originalmente equivalente a IDENTIFIER) deve incluir também como alternativas CLASS_ID, TYPE_ID e FUN_ID; de outra forma, não seria possível criar um membro com o mesmo nome de uma função.

```
program         : class_def

class_def       : CLASS PIDENTIFIER ...

class_type      : PIDENTIFIER ( ac_par_list )
                  /* A */
                  class_inst
                  ...

class_inst      : class_def EOF
                | EOF
```

Figura 3.4: Fragmento da gramática Cm

- ao reconhecer uma instanciação de classe (produção class_type), o analisador deve suspender o processamento da classe e iniciar a leitura de um novo arquivo. Na gramática isso é expresso como na figura 3.4. Isto é, o código em lugar de /* A */ instrui o analisador léxico para mudar de arquivo-fonte, de modo que o *parser* passa recursivamente a analisar outra classe, como se realmente o texto fosse inserido no arquivo original. Ao fim da classe inserida, o analisador léxico retorna também um *token* EOF (*end of file*, fim de arquivo) adicional (a não-inclusão de EOF ocasiona diversos conflitos r/r). Note-se a alternativa EOF (vazia) em class_inst; ela é acionada (novamente *yylex()* é instruída para retornar um átomo fictício) caso o código em A não encontre a classe incluída, se a classe já é conhecida, ou quando se sabe de antemão que os parâmetros reais de classe (ac_par_list) não estão corretos, sendo portanto inútil incluir realmente o arquivo.

- As variáveis internas usadas no analisador gerado por Yacc são fixas e globais, de modo que o *parser* não é reentrante. Caso houvesse reentrância, o código em /* A */ acima poderia simplesmente invocar *yyparse()* recursivamente, simplificando a gramática[7]. Há também necessidade de pilhas para armazenar as variáveis da gramática, antes de iniciar o processamento de cada classe incluída.

- embora houvesse uma versão bastante completa do programa Yacc dis-

[7] o analisador gerado por Bison, mencionado adiante, *pode* ser reentrante, mas essa característica e outras facilidades não foram aproveitadas no compilador Cm para manter compatibilidade com Yacc. É possível adicionar reentrância ao analisador de Yacc, mas é uma abordagem laboriosa e pouco compensadora

ponível para o sistema operacional MS-DOS, sua capacidade máxima de 300 regras logo foi excedida, forçando novas alterações na gramática. Seria possível gerar o analisador em ambiente *UNIX*, trazendo de volta o arquivo-fonte C para ser compilado em MS-DOS, mas esse procedimento é bastante moroso e não completamente portável. A necessidade de incorporar recuperação de erros sintáticos elevaria ainda mais o número de produções (outras questões trazidas pelo tratamento de erros estão apresentadas na seção correspondente), certamente impossibilitando o uso daquela versão de Yacc em MS-DOS. Foi assim necessário o porte de uma versão de *Bison* para este sistema operacional. Trata-se de um programa desenvolvido pelo Projeto GNU [DoS88], como uma alternativa ao uso de Yacc. Bison é compatível com Yacc a nível de sintaxe, e oferece algumas facilidades adicionais. A entrada e saída de ambos os geradores são quase totalmente compatíveis entre si, mas os algoritmos de análise implementados por *yyparse()* diferem. Bison foi escrito em C para sistemas *UNIX* em arquiteturas de 32 *bits*, o que ocasionou algumas dificuldades na conversão para MS-DOS. A maior capacidade de Bison em relação ao Yacc para MS-DOS permitiu continuar usando uma gramática único para todas as versões do compilador Cm (processada por Yacc em *UNIX* e por Bison em MS-DOS).

Apesar de todas as dificuldades para seu uso, a escolha de Yacc/Bison para a construção do analisador sintático foi uma decisão de projeto e se justifica tanto pela flexibilidade de manutenção como pela universalidade: Yacc está disponível em qualquer sistema *UNIX* que tenha um sistema de compilação C, havendo também programas compatíveis em domínio público. Por outro lado, detalhes da linguagem foram alterados em diversas ocasiões durante o desenvolvimento do projeto. Essas modificações seriam bem mais difíceis sem um gerador, principalmente quando envolviam a semântica da linguagem.

3.6 Tratamento de Erros

O processo de tratamento de erros do programador e sua recuperação é um dos tópicos mais difíceis de implementar na produção de compiladores. O tradutor Cm inclui o tratamento necessário para informar claramente o maior número possível de erros num programa.

É possível classificar os erros cometidos por programadores em quatro categorias:

léxicos referentes a palavras ou delimitadores malformados (comumente erros de datilografia)

sintáticos quando há falhas na estrutura do programa (construções incompletas, por exemplo)

semânticos se objetos definidos pelo programa não são usados apropriadamente, como no caso de erros de tipo, falta de declarações, expressões ambíguas, etc.

lógicos quando se falha no algoritmo descrito pelo programa. Este tipo de erro se distingüe por não impedir a compilação do programa; é o mais difícil (se não impossível) de tratar e não será discutido aqui

A distinção em categorias pode não ser muito nítida para o compilador. Num exemplo simples, se um comentário é iniciado mas a seqüência que o termina está incorreta (erro léxico), o comentário somente será considerado terminado na próxima seqüência de término (isto é, no próximo comentário, se houver). Todo o texto intermediário é ignorado, provavelmente ocasionando erros sintáticos. Assim, o tratamento de erros léxicos pelo tradutor Cm é bastante simples, limitando-se a verificar a legibilidade dos arquivos-fonte e passando átomos desconhecidos diretamente a *yyparse()*.

Comumente os compiladores concentram o tratamento de erros nos casos sintáticos: com uma gramática apropriada (em particular uma LALR) pode ser muito fácil determinar para cada situação quais os próximos átomos válidos, e portanto, como detectar situações inesperadas. A dificuldade está em recuperar-se do erro, isto é, continuar a análise do programa procurando por novos problemas. Essa recuperação deve supor que átomos corretos foram introduzidos, removidos ou substituídos por aquele realmente encontrado. Analogamente, o *parser* deveria remover, substituir ou introduzir um ou mais átomos fictícios, imaginar que o erro foi "corrigido" e prosseguir a análise. Infelizmente, não se pode determinar arbitrariamente a extensão da "correção" a ser feita, tornando-se bastante difícil uma recuperação robusta (isto é, que não ignore erros subseqüentes nem acuse como incorretos trechos legais de código).

Yacc provê mecanismos primitivos para recuperação de erros sintáticos. É possível, por exemplo, criar regras fictícias que reproduzem erros prováveis (a "ação" dessas regras consiste somente em advertir o usuário). Mas é claro que não é factível prever todos os modos em que um programador comete erros, o que limita a utilidade desse método. Normalmente, um analisador gerado por Yacc abandona a execução à primeira violação de sintaxe encontrada, logo após emitir uma mensagem de erro. Entretanto, o *token* pré-definido *error* pode ser usado em certas produções: quando ocorre um erro numa produção contendo *error*, o analisador modifica suas variáveis internas e emite a mensagem usual, mas não termina a análise. Ao invés disso, aceita a regra como correta, e passa a ignorar um máximo de três dos próximos átomos recebidos, esperando "sincronizar" um *token* com uma produção adequada.

Adicionalmente, Yacc fornece também recursos para controlar a quantidade de átomos a ignorar após a detecção de um erro. De qualquer modo, não há regras sistemáticas para o uso desses recursos ou do símbolo *error*. Além disso, o uso indiscriminado deste último pode aumentar consideravelmente o número de conflitos, em detrimento também da clareza da gramática. Nem é geralmente imediato garantir que nenhum erro induzirá o compilador a ciclos infinitos.

O compilador Cm apresenta uma recuperação sintática relativamente robusta. Cerca de 70 regras foram introduzidas até agora para prever/recuperar erros (aproximadamente 20% do total), e outras estão previstas. Por outro lado, o programa inclui mensagens mais precisas que as providas por Yacc ou Bison (um *yyparse()* normal produz apenas mensagens do tipo *"syntax error"*, quase nada informativas). Foram feitas alterações no arquivo matriz usado por Bison para criar o *parser*, de sorte que *yyparse()* é modificada para invocar uma rotina de erro especial. Esta função procura, dentre todos os *tokens* definidos, os que poderiam substituir aquele que causou o erro. Os códigos de *token* são transformados em nomes e adicionados à mensagem como "símbolo esperado". Assim, obtém-se automaticamente uma mensagem de erro com as possíveis alternativas corretas. Alterações semelhantes são possíveis e estão em estudo para dotar versões produzidas com Yacc dessa capacidade.

Tanto no caso dos erros sintáticos, após a "sincronização" do analisador, como para os casos semânticos, a recuperação deve se preocupar em manter consistentes as estruturas de dados do compilador. As produções da gramática comportam-se como funções, isto é, espera-se que devolvam valores (por exemplo, cada produção definindo uma subexpressão retorna uma estrutura de dados descrevendo a expressão parcialmente construída; produções para tipos devolvem uma referência aos descritores de tipos; produções para nomes devolvem apontadores para a tabela de símbolos, contendo o identificador e sua categoria, com uma indicação especial se o nome nunca foi declarado). Caso uma produção alternativa contendo *error* seja usada, a ação executada deve prover valores fictícios do mesmo tipo fornecido pelas produções convencionais. Como exemplo, as produções errôneas relativas a tipos usam um descritor "universal", de um tipo fictício compatível com todos os outros. Esse descritor também é empregado para marcar expressões contendo identificadores não-declarados.

3.7 Módulos Secundários

Algumas das secões do compilador Cm, embora essenciais ao funcionamento do programa, servem basicamente de suporte e não merecem tópicos separados.

3.7.1 A Tabela de Símbolos

Coleciona os identificadores definidos pelo usuário. A estrutura de dados que a implementa deve satisfazer como requisitos:

natureza hierárquica dado que identificadores em Cm podem ser redefinidos várias vezes em níveis léxicos diferentes

acesso rápido dada a freqüência com que identificadores são inseridos, pesquisados e removidos

A estrutura escolhida foi uma tabela de espalhamento (*hashing*) [Kn 73], com colisões resolvidas por encadeamento simples (*open chaining*). Essa característica também possibilita diferentes níveis léxicos, visto que as listas são manipuladas como pilhas. A função de *hashing* usada, *PJW* [Aa 86], apresenta bom desempenho contra colisões, sendo também facilmente calculável.

Um conjunto de estruturas de dados foi também definido para armazenar os descritores para diferentes elementos da linguagem (variáveis, classes, funções, tipos, entre outros). A natureza variável e recursiva de muitos desses elementos torna obrigatório o uso de estruturas dinâmicas.

3.7.2 Gerência de Arquivos

Arquivos são tratados pelo compilador tanto de forma seqüencial, como texto, pelo analisador léxico e pelo gerador de código, como com acesso indexado (i.e., registros acessíveis como elementos de um *array*), no tratamento de símbolos. Esta última função é aproveitada pela tabela de símbolos para armazenar e recuperar conjuntos inteiros de identificadores quando se muda de classe a compilar. Há duas razões para tanto:

- semanticamente, identificadores de uma classe herdeira/importadora não são conhecidos na compilação de classes herdadas/importadas, portanto devem ser removidos da tabela
- não sendo necessários, sua remoção economiza espaço de memória

O *swapping* de símbolos deve ser otimizado, dado que consome tempo apreciável na mudança de contexto entre classes.

3.7.3 Depuração

Para simplificar o desenvolvimento e a correção de eventuais erros, todas as partes do tradutor Cm contém, em pontos estratégicos, código de depuração e

rastreio. Normalmente ociosas, essas rotinas podem ser ativadas na linha de comando invocando-se o compilador com opções especiais. Suas funções são indicar o progresso na execução do programa e, ao mesmo tempo, descrever de forma legível o conteúdo de estruturas de dados complexas. Na realidade, para cada descritor de dados introduzido no compilador foi criada uma função associada capaz de exibir seu conteúdo. Considerou-se também fundamental para o desenvolvimento do tradutor a introdução de código de depuração concomitantemente a cada característica nova introduzida.

O nível de detalhe exibido pelas funções de depuração é controlado também por opções. Por exemplo, é possível solicitar ao compilador que descreva a interface de cada classe compilada, de cada função ou tipo criados, ou que ilustre passo-por-passo a construção de expressões; pode-se chegar a ecoar cada identificador reconhecido e seus atributos, ou cada linha de texto analisada. Alguns módulos têm código de rastreio que pode ser ativado independentemente do nível de detalhe selecionado (como exemplo, a tabela de símbolos pode fornecer estatísticas de desempenho).

O rastreio do analisador sintático criado por Bison/Yacc também pode ser ativado pela linha de comando.

Caso se julgue apropriado, é possível remover totalmente o código de depuração e o *overhead* associado[8] recompilando o tradutor Cm. Diretivas de compilação (macros) permitem retirar completamente tanto as funções como suas chamadas mudando apenas uma linha no arquivo de declarações globais. As macros usadas nas chamadas são flexíveis o bastante para suportar diversas funções com interfaces diferentes, enquanto suficientemente inconspícuas para não dificultar a compreensão do restante do código.

3.8 Geração de Código

A produção de código na linguagem-objetivo é a meta central de todo compilador; todas as outras tarefas são mero suporte para esta. No caso de Cm, esta provou também ser a parte mais complicada do projeto.

3.8.1 Requisitos de Tradução

O módulo de geração de código produz, para cada classe Cm, um ou mais arquivos com programas em C. Durante a compilação, todas as estruturas de dados descrevendo expressões e declarações em Cm são convertidas para formas equivalentes em C, seguindo algumas restrições:

[8] em tamanho de código, principalmente; há uma ligeira penalidade em tempo de execução, mesmo com o rastreio desabilitado

- programas C devem ser portáveis, aceitáveis por qualquer compilador C K&R [KR 78], em MS-DOS ou *UNIX*. Extensões ANSI [KR 88] podem ser adicionadas posteriormente.

- o domínio de nomes para compiladores C e *linkers* deve ser respeitado ao máximo. Dois ou mais tipos, funções, constantes ou variáveis exportáveis com o mesmo nome em duas classes Cm diferentes não devem causar conflitos de múltipla definição para o compilador C ou para o *linker*. O programador Cm não deve se preocupar em escolher nomes únicos entre todas as classes, de modo que o tradutor precisa eliminar conflitâncias.

  A mesma classe Cm normalmente é instanciada diversas vezes e com diferentes parâmetros; neste caso, os nomes declarados nas instâncias também não podem conflitar entre si. O mesmo ocorre com os nomes nos arquivos com as diferentes versões de cada classe.

  Se o conjunto de nomes aceitáveis como identificadores pelos compiladores C fosse arbitrariamente grande, não haveria problemas para diferenciar a mesma função de duas instâncias da mesma classe, bastando adicionar ao seu nome, na tradução para C, o nome da classe e uma tradução textual dos parâmetros da instância. É claro que isso não ocorre: compiladores C seguindo o padrão ANSI são obrigados a diferenciar nomes com até 31 caráteres, enquanto os ligadores devem reconhecer até 6 caráteres sem distinção entre maiúsculas e minúsculas. O domínio de nomes em outros compiladores geralmente é ainda menor.[9]

- operações rotineiras em Cm, como a cópia, passagem como parâmetro e devolução de objetos estruturados, não são possíveis para muitos compiladores C (sendo inclusive probidas fora do padrão ANSI), que somente permitem essas transações aos tipos padrão e apontadores. Isso significa que operações envolvendo valores em Cm podem ser traduzidas para apontadores em C.

3.8.2 Esquemas de Tradução

Classes — Estrutura Geral

A tradução de cada classe é feita de modo estanque das demais. Isto é, uma classe incluída (mencionada por outra classe numa cláusula **use**, **inherit** ou **import**) não tem nenhum conhecimento sobre a classe que a inclui e não interfere na compilação de suas subclasses, a não ser definindo seus parâmetros reais. Por

[9] o autor já usou compiladores C para *UNIX* em que apenas os seis primeiros caráteres de cada identificador eram diferenciados

sua vez, a compilação de uma subclasse produz uma estrutura de dados com a sua descrição, similar a um descritor de **struct/union**.

Cada instância de uma classe gera uma diferente versão da mesma função; cada variável cujo tipo é uma instância de classe produz diferentes versões de todas as variáveis e funções[10] declaradas naquela classe. Para evitar os conflitos de nome, cada classe é declarada em C como uma estrutura com membros cujos nomes se originam das variáveis globais e funções da classe. O domínio de nomes de membros em estruturas para compiladores C é independente daquele para variáveis e funções (um membro pode ter o mesmo nome que uma variável ou função). Por outro lado, o tratamento de membros normalmente não é passado ao ligador (este os trata como deslocamentos de endereços (*offsets*), eliminando colisões de nome).

Juntamente com o arquivo em C (extensão *.C*) para cada instância da classe, cria-se um arquivo de inclusão (extensão *.H*) contendo apenas a definição da estrutura da classe. Esse arquivo é incluído não só pelo arquivo *.C* (pela diretiva #include) com o código, mas por todos os arquivos cujas classes incluíram aquela instância.

Variáveis

Todas as variáveis globais de uma classe são diretamente transformadas em membros da estrutura em C. Variáveis não-exportáveis são também incluídas (em caso contrário, não haveria alocação de memória para elas), embora seu acesso seja proibido à classe importadora. Variáveis locais a funções não são transformadas em membros, pois sua criação geralmente é feita na pilha e decorre imediatamente da execução da função — não é necessário criar múltiplas cópias. Entretanto, as variáveis Cm locais a funções e com classe de armazenamento + devem ser tratadas como globais, pois não são criadas para cada execução da função em C. Assim, são transformadas em membros da estrutura com nomes fictícios.

Variáveis de classe exigem uma única cópia entre todas as instâncias. Portanto, são declaradas diretamente no código C como **static**, sendo invisíveis a todas as demais classes. Para que sejam exportáveis em Cm, a estrutura de classe declara membros homônimos de tipo apontador, através dos quais se faz acesso a essas variáveis.

Funções

Funções são declaradas de sorte a incluir um primeiro parâmetro extra — em C é declarado em parâmetro _self_ cujo tipo é "apontador para estrutura descrevendo a classe". Além disso, o membro da estrutura de classe correspondendo a cada

[10] embora o código em C não seja realmente repetido

função tem tipo "apontador para função" e seu valor após a criação das variáveis é o endereço da função correspondente em C. Toda chamada a função é feita passando-se um parâmetro adicional cujo valor é o próprio _self_, se a função é da própria classe, ou o endereço da instância da classe, em caso contrário.

Dessa forma, quando se invoca uma função Cm através de um objeto de tipo classe, o endereço desse objeto (isto é, a estrutura em C) é passado como parâmetro à função. No código C, esta função obtém acesso àquela estrutura específica através de _self_, possibilitando a multiplicidade de instâncias. Além disso, todas as funções são declaradas em C como **static**, impedindo conflitos de nomes com as outras classes.[11] A única exceção se faz com a função de iniciação, descrita mais adiante.

Para evitar o dispêndio de memória usada para armazenar os endereços de funções, seria possível declarar um único bloco de apontadores, **static** ao arquivo da classe e contendo todas os endereços de funções da classe. Cada instância teria apenas o endereço dessa estrutura (mais uma declaração no arquivo C) e as chamadas a função se fariam de forma duplamente indireta. A penalidade no tempo de execução seria compensada em aplicações onde a memória disponível é crítica. Uma versão futura implementará esta característica opcional; é necessária também uma forma de identificar a forma como uma classe foi compilada em C, pois as duas formas de geração de código são incompatíveis entre si.

Funções não-exportáveis não são acessíveis no código C a outras classes, portanto, não têm os endereços armazenados na estrutura de classe. Somente podem ser usadas por outras funções da própria classe e, sendo também declaradas em C como **static**, não ocasionam conflitos de nomes.

Mesmo quando funções Cm são declaradas retornando um tipo estruturado genérico, as correspondentes funções em C sempre retornam endereços, respeitando limitações de compiladores K&R. O código gerado deve criar um objeto daquele tipo e retornar seu endereço inicial.

A passagem de objetos estruturados por valor é feita declarando parâmetros formais fictícios. Para cada parâmetro (Cm) de nome x e tipo estruturado T, uma declaração substituta é criada em C, para o parâmetro _x do tipo "apontador para T". Uma variável local à função, com o nome e tipo originais é também criada, e imediatamente tem o valor copiado de _x. Como x tem efetivamente o tipo T, não é requerido tratamento especial nas expressões onde aparece. Na tradução de código contendo chamadas a essas funções, detecta-se a passagem de tipos não-escalares e, conforme a necessidade, gera-se código C para a passagem de endereços em lugar de valores.

[11] Armazenar apontadores para funções é aparentemente redundante, visto que todas as referências a funções são resolvidas estaticamente durante a compilação Cm. Entretanto, é uma forma de evitar conflitos com múltiplas funções homônimas, mesmo permitindo sua invocação de diferentes arquivos-fonte C. Além disso, facilita a introdução de ligação dinâmica

Quando funções Cm devolvem objetos de um tipo estruturado T, as funções C equivalentes devolvem apontadores para T. Para possibilitar seu uso em expressões genéricas, o código produzido não pode simplesmente declarar internamente à função uma variável estática desse tipo e retornar seu endereço. Assim, simula-se um regime de pilha (como feito com a valor devolvido por funções convencionais) criando um parâmetro extra de tipo T*. Ao devolver (via **return**) uma expressão do tipo T, a função em C deve copiar *byte* por *byte* o valor da expressão para o endereço descrito pelo parâmetro extra e devolver esse mesmo endereço. Em contrapartida, para cada expressão completa é feita uma verificação de chamada de funções; para cada retorno não-escalar é declarada uma variável desse tipo e seu endereço é aplicado na chamada. As variáveis auxiliares são criadas num novo nível léxico e, deste modo, somente ocupam memória durante a execução da expressão.

Um problema em aberto é a inclusão de classes em uniões. A área de apontadores não pode ser destruída quando outros membros da união são usados. Isso pode ser resolvido dividindo-se a estrutura de classe em duas partes, sendo a lista de apontadores de funções movida para fora da união num membro fictício.

Expressões

A produção de código em C é de certa forma complicada pela convenção de que nenhuma expressão ou chamada a função deve jamais devolver algo de tipo estruturado. O compilador necessita freqüentemente introduzir operadores de indireção (*) ou endereçamento (&) extras, como na passagem de parâmetros; ou, pelo contrário, evitar sua geração, como no caso de self.

Ao traduzir expressões, deve-se levar em consideração a transformação de variáveis e funções em membros. Expressões fora de funções somente podem ser usadas para declarar variáveis, instanciar tipos, ou declarar o valor inicial de variáveis globais, e sua tradução é mais ou menos direta, exceto quanto ao código de iniciação. Dentro de funções, sempre se considera o parâmetro _self_ apontando para uma estrutura de classe:

- variáveis globais são referidas com uma indireção via _self_
- funções exportáveis são invocadas do mesmo modo.
- funções não-exportáveis são chamadas diretamente pelo nome.
- qualquer chamada a uma função convencional em Cm passa _self_ como parâmetro adicional
- a chamada a funções de classe **cvirtual** é feita sem a passagem de _self_
- chamadas a funções virtuais são discutidas na seção dedicada à implementação de herança

- variáveis locais são referidas diretamente.
- variáveis locais de classe de armazenamento "+" (isto é, "estáticas" em C) são traduzidas por _self_ -> _CmStnn, onde nn é uma identificação única em toda a classe
- variáveis de classe são traduzidas usando _self_ e o membro de endereço mencionado anteriormente. São tomadas precauções para traduzir corretamente seu significado (*valores* em Cm, *endereços* em C). Caso sejam referenciadas na própria classe a que pertencem, podem ser traduzidas diretamente pelo nome.
- uma expressão de comparação em Cm deve verificar o tamanho do tipo dos operandos em C. Caso não seja possível a comparação imediata, a expressão é substituída pela chamada a uma função da biblioteca de execução Cm. Esta função simplesmente compara, *byte* por *byte*, dois trechos de memória
- expressões com o operador "=" também podem ser substituídas pela chamada a uma função de cópia. A função retorna o endereço de início do trecho recém-copiado, de modo que o resultado da expressão pode ser usado posteriormente.
- expressões com operadores **new** e **release** são substituídas por chamadas a funções da biblioteca Cm. As funções podem simplesmente usar *malloc* e *free* (presentes na biblioteca de execução C) como, opcionalmente, prover um controle melhorado sobre a alocação de memória dinâmica (detectar e tratar exaustão de memória, prevenir dupla de-alocação e de-alocação de blocos ilegais, etc.). Além disso, caso o tipo parâmetro de **new** inclua classes, é imprescindível executar a iniciação dos objetos criados (descrita mais adiante): a função na biblioteca recebe um parâmetro denotando a rotina de iniciação de classe, executada para o endereço inicial de cada instância criada.
- o nome **self**, referente à própria instância sendo compilada, somente pode ser usado dentro de funções. Traduz-se diretamente por _self_. Como o identificador _self_ em C é um parâmetro cujo tipo já é um endereço, a expressão Cm "**& self**" é traduzida de modo idêntico.

Código de Iniciação

O código de iniciação (em inglês, *start-up*) em C tem por finalidade preparar para execução uma variável cujo tipo é uma instância de classe. Trata-se de uma função cujo único parâmetro é um apontador para a estrutura em C daquela classe. Todos os membros referentes a variáveis daquela classe declaradas com

valor inicial sofrem uma atribuição para aquele valor. Membros da estrutura originários de funções exportáveis são iniciados com o endereço das funções (o endereço é conhecido, pois a função de iniciação é declarada no próprio arquivo C definindo a classe).

Adicionalmente, para todos os vetores cujos elementos são classes, chama-se a respectiva função de *start-up* passando-se como parâmetro o endereço de cada elemento. O mesmo ocorre para todos os membros de estruturas que são classes. É possível identificar a função, pois há um mapeamento direto entre seu nome e o da classe.[12] Esses dois procedimentos são efetuados recursivamente para todos os objetos de tipo definido em termos de alguma classe.

Caso haja uma função definida pelo usuário chamada *main()*, ela é invocada automaticamente pela função de iniciação, *após* os procedimentos normais. A função *main()* pode ser usada para um *start-up* mais elaborado que aquele provido pela simples atribuição de variáveis e endereços de funções.

Finalmente, a função de iniciação usa o próprio parâmetro como valor de retorno. Isso facilita a criação de instâncias de classes dentro de expressões.

O nome da função de iniciação e o nome da estrutura que define a instância de classe[13] são os únicos identificadores "globais" em C introduzidos para cada arquivo criado (isto é, visíveis para o *linker*), além dos identificadores de estrutura/união. Devem ser únicos entre todas as instâncias de classes.

Há outro tipo de código inicial, usado fora dessa função específica mas bastante similar. Trata-se daquele produzido para variáveis locais de funções e blocos léxicos. Neste caso, o código faz referência a variáveis simples, e não a membros da estrutura da classe.

Herança

O método proposto para mapear classes em Cm para estruturas em C aplica-se diretamente para objetos cujo tipo são instâncias de classes importadas. A implementação de herança baseia-se nos mesmos princípios mas é ligeiramente mais complexa. Durante a análise de uma cláusula inherit, o tradutor suspende a compilação da classe e passa a tratar a classe herdada, como usual. Ao retornar à compilação da classe herdeira, constrói-se uma estrutura de dados descrevendo a interface da classe herdada. A declaração inherit cria novos identificadores na tabela de símbolos, cujos descritores são na realidade apontadores para a estrutura de classe. Assim, os objetos exportáveis da classe herdada são reconhecíveis dentro da classe herdeira como se tivessem sido definidas nesta última.

[12] na presente versão do tradutor, caso a classe tenha nome X, a função de iniciação se chamará X*nn*., onde *nn* é um código que depende dos parâmetros reais daquela particular instância da classe

[13] na versão atual, corresponde ao nome da função precedido por "."

No código C gerado, a alocação de memória para os objetos herdados faz-se quase como se a classe fosse importada e uma instância sua declarada (o que implica na chamada da função de iniciação para cada classe parametrizada herdada). Todos os objetos herdados devem ser tratados de modo especial — no código C, uma operação de seleção de membro deve ser inserida. Se um objeto aparece na cláusula **rename**, uma entrada na tabela de símbolos é criada da mesma forma que para os objetos herdados. Dessa forma, torna-se equivalente ao identificador original, e este pode ser redefinido.

Ao fim da compilação da própria classe herdeira, os descritores de objetos herdados são transferidos para o descritor da classe; a informação de herança também é mantida, ou os objetos herdados não seriam referenciados corretamente pelas classes que eventualmente incluíssem a classe herdeira.

A geração da função de iniciação da classe herdeira inclui invocar, para cada classe parametrizada herdada, a respectiva função de iniciação, passando como parâmetro o endereço da estrutura declarada.

Funções herdadas em que parâmetros ou o valor de retorno têm como tipo a própria classe herdeira devem ser redefinidas de forma transparente ao programador, como visto em 2.8.3. Isso se faz com uma função simples que invoca a função herdada com os parâmetros acertados (endereço do membro em lugar de _self_), corrigindo também o tipo de retorno.

Herança: Alternativas e Restrições

Uma técnica bem mais simples consiste em, logo após a compilação da classe herdada, transformar todos os componentes da estrutura de classe em variáveis, funções e constantes da classe herdeira, simulando uma declaração local. Esses objetos passariam a ser indistingüíveis dos definidos localmente, exceto por serem passíveis de redefinição. O código de acesso e conversão é muito menos intrincado, e não há necessidade de nenhuma ação especial ao final da classe herdeira. Infelizmente, a função de iniciação da classe herdada seria invocada usando como parâmetro o endereço do primeiro objeto exportável, e não o de uma estrutura completa. Pode-se garantir que todos os objetos exportáveis estejam adjacentes e na mesma ordem que na estrutura declarada na classe herdada, mas não que o alinhamento usado na alocação de memória seja o mesmo para cada membro. Por questões de *padding*, programas compilados dessa forma poderiam ou não ser corretos, dependendo da implementação do compilador C e da arquitetura local. Portanto, o método, mesmo sendo mais prático, não é usado no tradutor Cm.

Uma limitação da compilação em um único passo é que as classes herdadas não são retraduzidas. Assim, na versão atual do compilador, alterações (na classe herdada) em objetos redefinidos não se refletem nas herdeiras. Por outro lado, funções na classe herdada não podem invocar uma função redefinida em

lugar da original. As conseqüências dessa restrição são reduzidas pela possibilidade de se usar funções virtuais. Na classe herdada, a estrutura C descrevendo a classe implica num membro com o endereço da função, e é através de _self_ e desse endereço que a função é invocada. No entanto, há um problema. Na classe herdeira, a função "real" é compilada tendo em vista um contexto diferente daquele da classe herdada (onde ela também pode ser usada). Isto é, o valor esperado do parâmetro _self_ para a função real não é o mesmo que para a classe herdada, e portanto, funções nesta classe não poderiam chamar corretamente a função real. Torna-se assim necessário um membro extra para cada função virtual na estrutura da classe herdada, contendo o endereço correto do contexto _self_ na classe herdeira, usado em lugar do _self_ da herdada em todas as chamadas àquela função. Então, para cada função definida como **virtual** na classe, acrescentam-se dois parâmetros na função de iniciação, relativos ao endereço da função e do contexto. Os valores corretos de ambos são conhecidos e passados pela classe herdeira, caso a função real esteja aí definida. Em caso contrário, ela é considerada virtual *também* nessa classe, e os endereços são obtidos de parâmetros de *sua* função de iniciação. Isso permite o "cascateamento" de funções virtuais através de uma cadeia de herança.

3.9 Gerência de Compilação

Uma sessão de compilação Cm pode demandar a tradução de diversas classes, a geração de vários arquivos *.C* e *.H*, e a posterior compilação e ligação dos arquivos-fonte em C. Para que uma classe seja compilada, todas as instâncias de classes por ela incluídas devem também ser compiladas com sucesso. Seria bastante tedioso deixar este processo (aliás propenso a erros e omissões) ao programador. Felizmente, Cm provê recursos para que isso ocorra automaticamente.

Durante o desenvolvimento de um projeto usando Cm, classes são criadas, alteradas e recompiladas várias vezes. Cada alteração numa classe exige que todas as classes que a incluem sejam novamente traduzidas. As traduções das subclasses, entretanto, já existem e não necessitam ser refeitas. Assim, requer-se que o tradutor Cm:

- mantenha a *consistência* do projeto: uma superclasse depende das subclasses e, quando qualquer uma destas é modificada, a superclasse deve ser recompilada
- evitar compilações redundantes: quando já existe uma tradução de uma classe com os parâmetros desejados, não há nada a ser feito

Requisitos semelhantes aplicam-se aos arquivos-fonte em C, e são discutidos em mais detalhe na seção 3.10.

O programa *make* popularizou-se inicialmente no sistema *UNIX* por facilitar a especificação de projetos. Através dele pode ser definida uma série de relações de dependência temporal entre arquivos e como de um arquivo podem ser gerados outros. Normalmente é associado à manutenção de programas em C (atualizar um programa executável com o mínimo de compilações) mas seus usos são inúmeros.[14] Infelizmente, por causa dessa flexibilidade, *make* exige que todas as relações de dependência e ações de atualização sejam definidas pelo usuário. O tradutor Cm executa tarefas similares, mas de âmbito bem mais delimitado. Além disso, todas as relações de dependência são dadas implicitamente pelas cláusulas **use**, **import**, e **inherit**. Portanto, a compilação de uma classe e de todas as suas subclasses pode ser automatizada sem nenhuma especificação adicional por parte do programador Cm.

Após a tradução de uma classe, o compilador atualiza um arquivo com o nome de todas as subclasses incluídas. Este arquivo existe para cada classe, num subdiretório com o nome da classe (sistemas de arquivos com subdiretórios são essenciais para impedir superposição de nomes) e é consultado antes de cada compilação subseqüente, juntamente com os arquivos correspondentes das classes incluídas, e assim por diante, recursivamente. Caso nenhuma das classes Cm tenha sido alterada mais recentemente que a classe a ser traduzida, nem esta após a tradução mais recente, a compilação pode ser abandonada.

Por outro lado, quando uma instância *deve* ser compilada, os parâmetros de instanciação são comparados com os registros de outro arquivo (também particular para cada classe), contendo os parâmetros reais de todas as instâncias já compiladas. Caso um registro coincida, indicando que já há tradução daquela classe com os mesmos parâmetros reais a compilação não é necessária; o registro também indica o arquivo contendo a tradução.

O arquivo de parâmetros[15] é atualizado a cada compilação efetiva e destruído caso seja mais antigo que a classe (isto significa que a classe foi alterada, tornando obsoletas todas as traduções).

Mesmo que não haja produção de código C para uma instância, é imprescindível conhecer seus objetos exportáveis (ou não seria possível a verificação de tipos). Como isso exige ler e analisar o arquivo da classe, a economia de tempo dispensando a geração de código não seria significativa. Para acelerar a análise, o tradutor pode preparar durante a primeira tradução uma *versão reduzida* da classe. Trata-se de um programa Cm derivado da classe original mas contendo apenas as declarações exportáveis: tanto comentários e declarações locais como corpos de funções, por exemplo, são eliminados. Expressões e declarações são se possível simplificadas. A versão reduzida será em média muito menor que a original e, portanto, lida rapidamente; é também parametrizável e única para

[14] como a manutenção deste texto

[15] na versão atual são diversos arquivos no mesmo subdiretório, cada um com os parâmetros de um única instância — é um método menos eficiente mas foi escolhido pela simplicidade

cada classe. Como o arquivo de parâmetros, é dependente da classe e refeita a cada alteração de classe.

Em certa medida, a capacidade de produzir código em *duas* formas — Cm para os arquivos de parâmetros e versões reduzidas e C para o código final — adiciona razoável complexidade ao tradutor, mas deve contribuir para reduzir sensivelmente o tempo da compilação de projetos em Cm.

Deve ser ressaltado que boa parte das funções de controle de instâncias e arquivos compilados foi originalmente atribuída ao gerenciador de bancos de dados do ambiente A_HAND [Ha 90]. Os serviços incorporados ao tradutor formam apenas parte das tarefas previstas. Em particular, espera-se incluir um controlador de versões [VMD89] bastante flexível, capaz de distingüir grupos de usuários e catalogar diferentes versões de classes, com acesso restrito e controlado via grupos. Ou seja, futuramente, o programador Cm não requisitará a herança, uso ou importação de uma classe, mas de uma *versão* daquela classe, seja ela a definitiva, experimental, particular, etc.

3.10 Procedimentos Finais

A classe cujo nome é o parâmetro principal do tradutor (isto é, aquela de mais alto nível, que inclui todas as outras) não deve ter parâmetros formais ou, se os tiver, todos devem possuir valores *default*. Em caso contrário, embora todas as subclasses sejam compiladas, não é possível gerar código para essa classe, pois o compilador ainda não tem como instanciá-la. Uma versão já prevista terá a capacidade de ler parâmetros reais de classe da linha de comando. Essa facilidade não é essencial (sempre se pode declarar uma classe fictícia sem parâmetros importando uma instância da classe original) e de implementação algo complexa (deve-se alterar o analisador léxico para ler a linha de comando, e o analisador sintático para aceitar um subconjunto da linguagem, contendo apenas expressões e expressões de tipo; possivelmente serão adicionados símbolos fictícios para transpor as limitações de Yacc), mas será útil ao ambiente integrado do A_HAND.

De qualquer modo, se foi possível gerar o código C para todas as classes usadas, o tradutor cria um último programa em C com uma única variável — uma instância da classe de mais alto nível — e apenas uma função, chamada *main()*. Esta é o "programa principal" usual em programas C, e sua única tarefa é invocar a função de iniciação da classe.

Com toda a informação sobre a hierarquia de classes e instâncias, o tradutor deve cuidar para que todos os arquivos C produzidos sejam compilados e ligados ao arquivo com *main()* e à biblioteca de execução Cm, gerando um programa executável. O compilador conhece a hierarquia de arquivos e poderia, verificando a data das traduções, invocar por si o compilador C para todos os arquivos a

atualizar. Contudo, é bem mais prático preparar um *makefile*, ou arquivo de especificações para o programa *make*, tarefa que, para o programador, seria tediosa e inclinada a erros. Além da vantagem da simplicidade, isso torna-se interessante em sistemas com pouca memória disponível (tanto o compilador C como o programa que o requisitou permanecem na memória simultaneamente, e *make* é normalmente bem menor que o compilador Cm).

Embora haja várias implementações de *make* para MS-DOS, nem todas são tão poderosas ou seguem a mesma sintaxe da versão encontrada em todos os sistemas *UNIX*. Além disso, há diferenças inerentes ao sistema operacional, como o formato dos nomes de arquivos e diretórios, o interpretador de comandos, e opções de compilação. O tradutor Cm produz *makefiles* usando um entre dois "estilos" diferentes, normalmente predeterminado pela configuração local, mas alterável de acordo com parâmetros definidos na linha de comando:

- para o *make* de *UNIX*
- para a versão distribuída com o compilador Turbo C

Mesmo assim, grande parte das opções e comandos é definida usando macros, conveniência comum a ambas as versões de *make* suportadas. Isso facilita alterações e ajustes por parte do programador. Por exemplo, ao compilar programas na arquitetura IBM PC, deve ser decidido o "modelo" de memória do código objeto. Esse modelo estabelece alguns compromissos entre tamanho do código, velocidade de execução e capacidade de endereçamento do programa. O sistema de arquivos do compilador Cm possibilita manter diferentes conjuntos de código objeto, um para cada modelo e instância de classe. Para selecionar um modelo que não seja o usual, basta que o programador Cm edite o *makefile* e altere um único caráter.

Outra complicação do ambiente MS-DOS é o fato da linha de comando ter comprimento reduzido, o que impede a ligação de muitos arquivos-objeto. O tradutor Cm é obrigado a gerar um arquivo adicional de especificação para o *linker*. A sintaxe desse arquivo é relativamente a mesma para os ligadores mais populares, em contraste com os arquivos de biblioteca — diferentes para cada compilador C.

Normalmente, o compilador Cm não invoca automaticamente o programa *make*; sua execução termina com a criação do *makefile* (cujo nome também é derivado do da classe de mais alto nível). Uma opção de comando na versão *UNIX* solicita a execução de *make* e destrói o *makefile* se *make* teve sucesso, deixando apenas o programa executável. Essa opção não seria muito útil se aplicada diretamente em MS-DOS, sendo a memória mais limitada, mas pode ser implementada com o auxílio de um programa em *batch*.

Capítulo 4

Cm: Perspectivas

Thus, let the surface be what it will, I must always put the question: what is beyond?
G. Bruno, *Infinite Universe and Worlds*
Trad. por D. Singer

Cm é uma linguagem em evolução. A primeira versão pública do tradutor permitiu os primeiros estudos sobre a viabilidade da linguagem e as direções no seu desenvolvimento futuro.

4.1 Restrições Inerentes à Implementação

Certos aspectos da versão atual do compilador Cm não estão resolvidos de forma plenamente satisfatória. Alguns problemas ligam-se diretamente à forma de implementação escolhida para a linguagem.

4.1.1 Avaliação de Expressões e Características de Arquitetura

Normalmente, expressões em programas Cm são analisadas quanto ao tipo e categoria. O tipo é essencial dada a verificação estrita exigida pela linguagem. A categoria abrange classificações como:

- se a expressão é constante (calculável durante a compilação); isso é um requisito para limites de *arrays*, valores de enumerados, tamanho de membros e constantes de seleção do comando **switch**, entre outros.

- se a expressão é um *lvalue*. Um *lvalue* [Aa 86] é uma referência a uma posição de armazenamento cujo valor pode ser modificado. Essencialmente, é algo que pode estar do *lado esquerdo* de um operador como "=", daí o nome (de *lefthand side value*).

- se o tipo conhecido da expressão em Cm exprime mais ou menos fielmente o tipo equivalente em C. Por exemplo, variáveis não-estruturadas internas a funções são exatamente equivalentes no código Cm original e no código C traduzido. Um parâmetro formal de função cujo tipo é estruturado mas não indexado é um valor em Cm mas um apontador em C, enquanto uma variável de classe exportada em Cm pode ser um apontador, enquanto é um valor no programa C gerado.

Em diversos momentos é desejável o cálculo do valor de uma expressão constante em Cm; ou, pelo menos, a comparação entre duas expressões constantes, como no caso em que se requer decidir se duas parametrizações de uma classe exprimem o mesmo tipo. Entretanto, o operador **sizeof** exige conhecimento sobre o compilador C usado, o que significa que expressões contendo esse operador não podem ser completamente avaliadas pelo tradutor Cm. Embora seja simples produzir versões particulares do tradutor com informação sobre o valor de **sizeof** para os tipos simples num determinado compilador C (com o agravante dos modelos de memória em MS-DOS), não é imediato lidar com estruturas e uniões sem que se saiba também como o compilador trata *padding* (alinhamento) de membros.

Apesar da avaliação completa do valor não ser imprescindível à geração de programas executáveis (sempre se pode traduzir uma expressão equivalente em C), essa restrição limita a precisão do sistema de equivalência de tipos em Cm a verificação de dígitos significativos em operações de cópia, promoção de tipos e comando **switch**..

4.1.2 Herança

Uma premissa básica do tradutor é a compilação em um único passo, para cada parametrização diferente de cada classe. Como classes herdadas não são recompiladas após a herança, o código C produzido não é alterado devido a redefinições nas classes herdeiras. Isso é adequado enquanto cada herança apenas adiciona atributos. Não há herança negativa em Cm, e a redefinição de um objeto adiciona um membro à estrutura de classe sem remover o original. Isso significa crescente desperdício de espaço de armazenamento através de uma cadeia de herança à medida que atributos vão sendo redefinidos ou deixam de ser usados.

4.1.3 Restrições do Compilador C

Certas características de programas Cm, como o número máximo de níveis léxicos, são intrinsicamente limitadas pela implementação atual, enquanto outras, como o total de variáveis exportáveis, dependem da capacidade de memória e da arquitetura local. Além disso, não é possível garantir que nenhum programa Cm se tornará tão complicado ao ser traduzido a ponto de sua compilação C se tornar impossível. Todo compilador tem limites, e algumas construções Cm são efetivamente convertidas para estruturas mais complexas em C, principalmente expressões (com a acomodação de objetos estruturados, tratamento de instâncias de classes e variáveis de classe, por exemplo). Os limites de um compilador C mais passíveis de violação por complexidades introduzidas pelo tradutor são:

- número de níveis (blocos) léxicos, adicionados para a declaração temporária de variáveis estruturadas e índices para iniciação de instâncias estruturadas de classes
- profundidade de parênteses, introduzidos em expressões principalmente nas operações com apontadores
- total de parâmetros formais por função (são introduzidos até dois parâmetros adicionais por função; a função de iniciação tem um parâmetro fixo, mais dois por função virtual Cm pendente)
- número de membros de estrutura/união

Os valores *mínimos* que devem ser aceitáveis por qualquer compilador C segundo o padrão ANSI são 15, 127 e 31, respectivamente, para os três primeiros limites. A probabilidade de colisão de identificadores na fase de ligação, apesar de muito minimizada pelo esquema de tradução adotado, ainda existe caso haja nomes de classe muito extensos.

4.1.4 Depuração

A depuração (*debugging*) de programas pré-processados não é simples. Depuradores simbólicos para C e outras linguagens obtém do código executável informações sobre como apresentar o estado de execução do programa, relacionando-o com o código na linguagem-fonte. Entretanto, o tradutor Cm introduz ou mesmo altera completamente características do programa ao convertê-lo para C. Isto requer do programador intuição e algum conhecimento dos esquemas de tradução do compilador para relacionar corretamente objetos apresentados pelo depurador (C) com aqueles no programa original (Cm).

O tradutor pode até certo ponto auxiliar a depuração inserindo diretivas e comentários no código C ligando partes desse código com trechos do fonte Cm.

É possível que essa questão seja resolvida apenas com a implementação de um compilador completo, independente de C.

4.2 Questões de Eficiência

Reconhecidamente, a linguagem C é muito usada em aplicações comerciais requerendo código razoavelmente eficiente. De fato, a maior parte de suas construções reflete conceitos próximos do *hardware*, o que possibilita ao programador proficiente um melhor aproveitamento dos recursos usados pela aplicação. Deve no entanto ser notado que várias capacidades da linguagem, como a possibilidade de *aliases* e a forma de passagem de parâmetros, podem dificultar consideravelmente a geração de código otimizado.

A linguagem Cm não foi desenvolvida com eficiência de execução como o objetivo principal, enfatizando muito mais a velocidade de programação. Mesmo assim, algumas comparações se fazem necessárias para avaliar a implementação do compilador.

Três são as potenciais fontes de degradação de desempenho na tradução de Cm para C:

- a passagem de variáveis globais para membros de estrutura, transformando acessos diretos à memória em acessos indiretos via apontador; isso é agravado em variáveis de classe
- a conversão de chamadas de função a chamadas indiretas via endereço, somada à passagem obrigatória do parâmetro adicional _self.
- a redefinição de itens herdados acrescentando membros de estrutura não-utilizados[1]

Alguns testes muito simples foram realizados com programas equivalentes em Cm e C, observando a variação de desempenho. Todos os programas foram compilados e executados num microcomputador IBM PC, no modelo de memória *small*. Esse ambiente é muito mais simples que um sistema *UNIX*, apresentando resultados mais controláveis.[2]

O programa na classe Cm *tim1* foi usado para testar a velocidade de acesso a variáveis:

```
class tim1 ()

cvirtual long int clock ();
```

[1] esta inconveniência também ocorre na implementação de *cfront* estudada [Su 89]

[2] testes mais completos estão em andamento envolvendo o compilador *cfront* em *UNIX*

Versão	Cm		C	
	Otimizado	Não-otimizado	Otimizado	Não-otimizado
1	137/7076	275/7108	138/6676	275/6692
1'			69/6676	138/6692
2	271/7092	487/7140	298/6692	299/6692
3	447/7124	663/7172		

Tabela 4.1: tempo de execução (interrupções de relógio) e tamanho de código executável (*bytes*) para programas Cm e C: *tim1*

```
cvirtual int printf (...);

void main ()
{
  int i, j, k;
  long int t0, t1;
  t0 = clock ();
  for (i = 0; i < 1000; i++)
    for (j = 0; j < 1000; j++)
      k = i + j;
  t1 = clock ();
  printf ("Total ticks: %ld\n", t1 - t0);
}
```

O teste privilegia acessos simples à memória, sem indexação. Três versões do programa foram compiladas e executadas em Cm. A função *main* foi copiada diretamente para um programa C também compilado em duas versões e executado como controle. A tabela 4.2 indica para cada caso o tempo de execução (medido em interrupções do relógio de *hardware*, aproximadamente 18,9 por segundo) e o tamanho do código executável em *bytes*. Cada programa foi executado três vezes e o resultado prevalente foi incluído na tabela; não houve variações superiores a 1.

Note-se que um compilador C relativamente sofisticado poderia perceber que o resultado do programa independe do número de iterações, substituindo tudo por "$k = 1998$;" e falseando os resultados do teste. Mais ainda, k é local e não é usado, o que permite eliminar totalmente o comando. Essa análise não é substancialmente difícil porque nenhuma função é chamada, todas as variáveis são locais e nenhuma é **volatile**. A versão 1' foi compilada apenas em C para verificar a capacidade de otimização do compilador (Turbo C) e é praticamente igual à versão 1, mas o limite superior do comando **for** externo vale exatamente

Cm		C	
Otimizado	Não-otimizado	Otimizado	Não-otimizado
106/7096	110/7128	93/6712	97/6728

Tabela 4.2: tempo de execução (interrupções de relógio) e tamanho de código executável (*bytes*) para programas Cm e C: *fat*

a metade. Como se depreende dos resultados, pouca ou nenhuma otimização foi introduzida.

Para cada versão, foram executadas variantes com as opções de otimização ativadas e desativadas. As diferenças verificadas foram sensíveis em alguns casos e praticamente desprezíveis em outros.

A versão 1 usa apenas variáveis locais i, j e k. A tradução para C do corpo de *main* é essencialmente idêntica ao código original e, como esperado, os resultados quase coincidem em C e Cm.

Na versão 2, as variáveis i, j e k tornaram-se globais (isto é, em Cm são variáveis de instância, recriadas em cada possível instância da classe *tmi* e acessíveis via apontador de estrutura). A diferença de desempenho das versões normal e otimizada é marcante em Cm.

Finalmente, a versão 3 não tem equivalente em C. Nela, as variáveis são declaradas como sendo *de classe* (classe de armazenamento "+"). A perda de velocidade é sensível, devido à dupla indireção introduzida na tradução para C. A otimização simples proposta para variáveis de classe não-exportáveis eliminaria essa degradação, tornando o acesso a essas variáveis tão rápido quanto aquele feito a variáveis "globais" em C.

O segundo teste enfatiza o desempenho de chamadas a funções. A classe *fat* define uma função recursiva com um parâmetro:

```
class fat ()

long int fatorial (int n)
{
  return (n < 3 ? n : n * fatorial (n - 1));
}

void main ()
{
  int i;
  cvirtual long int clock ();
  cvirtual int printf (...);
```

```
    long int c0, c1;
    c0 = clock ();
    for (i = 0; i < 10000; i++)
      fatorial (12);
    c1 = clock ();
    printf ("%ld\n", c1 - c0 );
}
```

O programa de controle em C é quase idêntico. Os resultados da execução comparada estão na tabela 4.2. No modelo de memória usado, o *overhead* de passagem de parâmetros para a versão em Cm é exatamente o dobro daquele para C, além da chamada indireta a função. Essa desvantagem se diluiria em funções com mais parâmetros. Além disso, em muitos casos a abordagem de Cm "economiza" um parâmetro que seria necessário em C, como visto na seção 4.3.5, tornando os *overheads* ainda mais próximos.

Outras otimizações no código gerado são possíveis, em particular em funções devolvendo tipos complexos. Algumas operações de cópia desnecessárias podem ser geradas, o que se evitaria com uma análise dos descritores de expressões.

4.3 Próximas Etapas

4.3.1 Parâmetros de Classes

Para simplificar a análise, atualmente apenas expressões constantes são aceitas como parâmetros reais de classes em Cm. A próxima versão deixará a análise de quando uma expressão não-constante é inaceitável (por exemplo, na especificação do tamanho de vetores) a cargo da classe herdada/importada, ao contrário do que ocorre hoje. Parâmetros não-constantes são muito úteis e evitam replicação de código C.

4.3.2 Macroprocessamento

Algumas características da linguagem Cm no tocante ao suporte a modularidade foram imaginadas como forma de dispensar o pré-processamento, evitando a inclusão explícita de arquivos de *header* para manter a consistência entre módulos. A definição de macros simples pode ser substituída por tipos enumerados e pela declaração de constantes.

Cm não prevê macrocomandos complexos ou parametrizáveis, tampouco compilação condicional. A linguagem não exclui o uso do pré-processador para resolver essas questões, mas soluções integradas à linguagem seriam preferíveis e devem ser estudadas futuramente.

4.3.3 Múltiplos Passos

A proposta original de implementação previa mais de uma passagem sobre o código-fonte Cm, inicialmente resolvendo todas as relações de dependência entre classes para só então gerar código C. Por razões de velocidade de compilação e simplicidade, decidiu-se pela implementação em um único passo. Isso introduziu a questão da memória em objetos herdados redefinidos e forçou a declaração antecipada de tipos. Uma nova implementação seguindo a proposta inicial resolveria esses problemas, ao custo de alguma complexidade para passar informações entre passos e menor desempenho.

4.3.4 Cm e C++

Embora a linguagem Cm enfatize uma abordagem diferente daquela proposta por C++, são inevitáveis as comparações. As duas linguagens são de certa forma complementares, enfatizando de um lado polimorfismo paramétrico e de outro *overloading* de operadores. Ambas provêem herança múltipla e verificação forte de tipos. Também oferecem alguma compatibilidade com C e compartilham alguns problemas de violação de abstração [Li 91]. Rigorosamente, herança como implementada em Cm e C++ (e em outras linguagens) viola o encapsulamento porque todos os itens definidos numa classe são visíveis à herdeira.

Os mecanismos de polimorfismo em Cm — na experiência de seus primeiros programadores — mostraram-se satisfatórios e em certos casos mais apropriados que os de C++. Todavia, alguns aspectos desta última foram especialmente sugeridos como incorporáveis a Cm.

- funções *in-line* são expandidas como macros, mas respeitando tipos e escopo. Têm como principal vantagem a velocidade de execução, poupando o *overhead* da chamada de funções ao custo de alguma perda de memória (desprezível em funções reduzidas ou invocadas poucas vezes). Outros reflexos na eficiência do programa são possíveis: a otimização do código é muito mais simples num bloco sem chamadas a função [Aa 86][HC 89].

- a definição de funções para construção e destruição é um ponto forte de C++, principalmente com o *overloading* de construtores. Em contrapartida, Cm oferece apenas uma função de iniciação e a função de destruição, caso houver, deve ser invocada explicitamente. Contudo, Cm não deve introduzir tais recursos sem que sejam investigadas algumas inconsistências, como a invocação de construtores com parâmetros em tipos *derivados* de classes, a ordem de execução de destruidores e a destruição desnecessária de objetos [Sa 88].

- alternativas à função de destruição implícita devem ser analisadas. Em muitos casos, sua tarefa é destruir memória dinâmica criada pela instância.

Garbage collection automática é uma opção mais segura e cômoda para o programador, embora tenha conseqüências no desempenho do programa. Uma separação explícita a referências "automáticas" e "manuais", como em Modula-3 [Cal89] pode ser a solução mais adequada a linguagens híbridas como Cm.

- é desejável um suporte mais simplificado a programas e arquivos de inclusão em C. A resolução desse problema em C++ também deixa a desejar [Su 89].

4.3.5 Suporte à Execução

Foi um requisito de implementação que Cm não dependesse de *binding* dinâmico (para acelerar a execução dos programas compilados), nem de um sistema de suporte a execução dispendioso (para facilitar o transporte e a disseminação de programas). O suporte atual contém apenas funções para manejar objetos estruturados e acarreta pouco *overhead*. Um sistema mais complexo deve ser implementado quando a parametrização de classes via linha de comando estiver completa.

Poucos programas podem ser úteis se não interagirem com o ambiente externo. Tal como C, Cm praticamente não define meios de entrada/saída, cabendo a bibliotecas de suporte todo o gerenciamento dessas tarefas. Até o momento, o único modo de executar em programas Cm rotinas escritas em C, chamadas ao sistema operacional e outras funções de suporte é através da declaração **cvirtual**. Se isso é satisfatório para grande parte das necessidades, não é suficiente para declarar funções que usam tipos complexos. Para esses casos e para usar constantes definidas externamente em C, deve ser feita uma tradução de cabeçalhos e declarações em arquivos *header*.

O próximo projeto importante no desenvolvimento de Cm destina-se a superar essa dificuldade e consiste em reescrever os "pacotes" de funções mais conhecidos em C (vulgarmente conhecidos pelos nomes dos arquivos de inclusão que os declaram, como *stdio*, *string* e *fcntl*) em Cm, de forma mais racional e adequada ao paradigma de objetos. Por exemplo, a biblioteca de entrada/saída de arquivos de alto nível em C declara no arquivo de inclusão *stdio.h* uma tipo abstrato de dados *FILE* e funções que devolvem e usam como parâmetro principal apontadores para esse tipo. Em C, a função *fopen()* torna um arquivo disponível e devolve um apontador que o identifica; ele é normalmente copiado em uma variável e usado em chamadas de funções subseqüentes:

```
FILE * arquivo;
arquivo = fopen ("arquivo.txt", "w");
fprintf (arquivo, "Texto a escrever no arquivo");
```

```
fclose (arquivo);
```

Uma futura classe *file* definiria um tipo de dados para arquivo, e os objetos desse tipo chamariam diretamente as funções definidas na classe:

```
import file;
file () arquivo;
arquivo.open ("arquivo.txt", "w");
arquivo.prints ("Texto a escrever no arquivo");
arquivo.close ();
```

Alternativas mais sofisticadas devem ser buscadas para usar "pacotes" mais complexos, como *curses* [At 89] e *X Window* [QR 90] (possivelmente um gerador automático de *headers* Cm) e geradores como Yacc (reescrever o arquivo matriz de *parsers* e criar uma nova versão do gerador que produza código Cm).

4.3.6 Compilação Direta

O projeto de ambiente de desenvolvimento de programas A_HAND prevê a definição de uma *representação essencial* para programas, na qual os objetos executáveis do ambiente serão codificados, seja qual for a sua linguagem original. Essa representação seria genérica o bastante para incluir num mesmo programa trechos compilados (executados diretamente), não-compilados (a compilar *on-the-fly* ou mesmo interpretar), incompletos (a depurar ou saltar) e em diferentes linguagens. A representação essencial, incorporada ao sistema gerenciador de bancos de dados juntamente com o controle de versões, será uma linguagem-objetivo do compilador Cm, numa futura versão independente de compilação intermediária em C.

4.3.7 Outros Tópicos

Cm é, entre as linguagens de programação criadas para o ambiente A_HAND, a mais voltada para a programação de sistemas. A integração de Cm com as demais linguagens (LegoShell, gráfica, para criação de protótipos; CO^2, textual) é imprescindível à operação do ambiente; aguarda-se para breve o término do sistema de execução conjunto LegoShell e CO^2, o que permitirá o uso indistinto de programas Cm e composições de programas (computações, [Dr 89]) pela LegoShell e CO^2.

A introdução de tratamento de exceções como parte integrante da linguagem Cm, como feito em Ada, foi proposta já há algum tempo como projeto de Mestrado. Esse estudo analisaria como incorporar de modo flexível e genérico

eventos de exceção, muito comuns na programação de sistemas. Não há um consenso sobre quanta informação deve ser dedicada ao tratamento de exceções, ou quais os procedimentos de recuperação mais adequados [Mee90][WO 90]. Uma proposta interessante encontra-se em [Do 88].

O suporte provido a funções com número variável de parâmetros é o mesmo em Cm e C. Oficialmente não são parte da linguagem, e os parâmetros opcionais são obtidos a partir do endereço dos parâmetros fixos. Essa técnica é bastante genérica e permite combinar tipos arbitrariamente. Contudo, requer a desativação da verificação de tipos, tornando o programa mais suscetível a erros. Devem ser analisadas formas de declaração e chamada com um compromisso entre flexibilidade e segurança [CK 90].

Produzir uma linguagem como Cm também oferece a oportunidade de formalizar alguns pontos obscuros ou deixados dependentes de implementação na linguagem C, como o pseudo-operador **sizeof**. Em Cm, se o operando de **sizeof** é uma expressão, esta por definição nunca é avaliada, usando-se apenas seu tipo (evitando efeitos colaterais). Entre os aspectos a considerar no futuro estão a reorganização de subexpressões com a mesma precedência (que pode levar a problemas de precisão e *overflow*), ordem de avaliação de parâmetros (possíveis efeitos colaterais) e o operador "%" (cujas funções deveriam ser separadas como ocorre em Ada). Outros detalhes merecem revisão mas dependem da implementação do compilador C (não sendo então resolvidos de forma universal), como a promoção automática de parâmetros de tipo **float** em compiladores pré-ANSI.

4.4 Conclusão

A primeira implementação de Cm foi completada e demonstrou a capacidade da linguagem em suportar programação modular através da abstração de dados, encapsulamento e herança. Cm ainda deve sofrer diversos refinamentos em resposta às necessidades de uso em projetos complexos e distribuídos entre vários programadores.

A consolidação da linguagem através do *feedback* dos usuários deve estabelecer Cm como a principal ferramenta de desenvolvimento de programas complexos no Projeto A_HAND e na construção do próprio ambiente. O refinamento da linguagem será facilitado pela proximidade da equipe de desenvolvimento com a comunidade de usuários e programadores do Projeto. Entretanto, esse processo demandará algum tempo: o Projeto não tem muitos recursos para suportar a disseminação de Cm e a manutenção do código produzido. Espera-se dispor logo dessa possibilidade, com a conclusão das bibliotecas de classes mais importantes e a estabilização do compilador. Um Manual de Referência completo e atualizado apresentando de forma sistemática todos os aspectos de Cm está em

andamento, a ser publicado em breve.

Finalmente, a programação em Cm pressupõe e estimula o uso de técnicas de organização de programas, tanto nos detalhes internos como no relacionamento entre módulos. Esses métodos de programação já foram propostos há algum tempo, estando submetidos à avaliação do Projeto; após esse processo devem ser também formalizados e publicados.

Apêndice A

Cm: Página de Referência UNIX

NOME

cmc — compilador Cm

SINOPSE

cmc [**opções**] [classe [.]

DESCRIÇÃO

Cmc compila um programa Cm. A *classe* parâmetro corresponde ao mais alto nível de uma hierarquia de classes. Cada arquivo contendo uma única classe é inicialmente traduzido para um conjunto de arquivos em C, que devem por sua vez ser compilados por *cc*(1). Em lugar do arquivo pode ser passado '.' como parâmetro, sendo usada a entrada padrão em lugar de um arquivo (os arquivos compilados terão prefixo *stdin*). As opções possíveis são:

-al # permitir linhas de texto de até # caráteres

-as # aceitar *strings* constantes de até # caráteres (há um limite efetivo, dependente de *cc*(1))

-ai # aumentar o comprimento máximo de identificadores para # caráteres

-cl inserir diretivas *#line* no código gerado

–ca produzir código segundo padrão ANSI

–e reproduzir entrada no terminal (eco)

–h relacionar opções possíveis nesta versão

–mM invocar *make*(1) imediatamente para gerar um programa executável (*default* em *UNIX*); remover *makefile* se *make* teve sucesso (dispensa invocar a compilação C)

–mm não produzir o programa executável (*default* em MS-DOS)

–mp preservar *makefile* mesmo se compilação teve sucesso

–ms inibir mensagens advindas de *make*

–mt gerar *makefile* para MS-DOS

–mu gerar *makefile* para *UNIX* (*default*)

–x # usar nível de rastreio # (apenas para depuração)

–X # liga rastreio específico (dependente de versão)

–y habilitar rastreio do analisador sintático

As variáveis de ambiente CMC, CMDATA e CMSRC podem ser usadas para passar informação adicional ao compilador. CMDATA indica um diretório onde o tradutor armazena dados sobre as classes compiladas. CMC é o diretório de instâncias traduzidas. CMSRC pode conter uma lista de diretórios (separados por ":"[1]) onde o tradutor procura pelas classes Cm (os diretórios são pesquisados da direita para a esquerda). Qualquer dos diretórios nas variáveis pode conter referências absolutas ou relativas, além de referências ao HOME de usuários. Se alguma das variáveis estiver indefinida, "." é usado.

As variáveis de ambiente CMCFLAGS, CMLDFLAGS e CMCLIBS são aceitas para modificar o comportamento da compilação do código C gerado. CMCFLAGS altera a produção do código-objeto, enquanto CMLDFLAGS inclui as diretivas para o ligador. CMCLIBS indica bibliotecas ou módulos-objeto adicionais (possivelmente escritos em C ou outra linguagem).

EXEMPLOS

```
export CMSRC=.:$HOME/src/cm:/usr/local/lib/cm
cmc project
cmc -mm project1
make -f project1.umk
```

[1] ";" em MS-DOS

ARQUIVOS

classe.cm	Fonte Cm (contido em $CMSRC)
$CMDATA/*classe*/p*nn*.dat	Dados da instância
$CMC/*classe*/p*nn*.c	Tradução em C
$CMC/*classe*/p*nn*.h	
$CMC/*classe*/p*nn*.o	
/tmp/Cm*	arquivos temporários
./*classe*.umk	*makefile UNIX*
./*classe*.dmk	*makefile* MS-DOS
./*classe*.c	arquivo principal C
./*classe*.o	arquivo principal (objeto)
./*classe*	programa executável

DIAGNÓSTICOS

o código de retorno é 0 em caso de sucesso. Diversos valores não-nulos indicam falha de compilação

VEJA TAMBÉM

cc(1), *make*(1)

Apêndice B

Cm: Incompatibilidades com C

A definição de Cm mantém certa compatibilidade com C a nível de comandos da linguagem e estrutura geral. Contudo, diversas características são bastante diferentes nas duas linguagens e devem ser ressaltadas para programadores já proficientes em C.

- funções, tipos, variáveis e constantes sempre pertencem a uma classe (2.1, 2.8) e cada módulo é precedido pela declaração **class**
- uma classe define tipos e pode ser usada para criar diversas variáveis
- variáveis "globais" (como seriam consideradas em C) têm diversas instâncias, uma para cada instância da classe com dados parâmetros
- a cláusula **const** indica um objeto equivalente a uma constante, e não variáveis ou parâmetros cujo valor não deve ser alterado (2.3.4)
- os operadores de construção de tipos (para apontador e vetor) seguem uma ordem mais descritiva e não se agregam aos objetos declarados (2.5)
- tipos de parâmetros de funções são sempre declarados junto aos mesmos, segundo convenções do item anterior (2.7)
- Cm e C têm diferentes conjuntos de palavras reservadas (2.2.2)
- a função *main()* não é necessariamente única para todo o programa Cm, podendo haver uma para cada classe parametrizada. Não devem ter parâmetros e são executadas a cada criação de instâncias da classe (2.1, 2.8.6)

- tipos são sempre verificados, mesmo enumerados; a comparação de tipos é estrutural (2.3.2, 2.3.5)
- o tipo de retorno de uma função deve ser sempre explicitado, mesmo que seja **int** (2.7)
- os operadores **new** e **release** são introduzidos para requerer e liberar memória dinâmica, respectivamente (2.6.2). O uso das funções *malloc()*, *calloc()* e *realloc()* ainda é possível (exceto para objetos cujo tipo contenha alguma classe), desde que declaradas como **cvirtual** (2.8, 2.7.4)
- todos os objetos numa classe têm acesso implicitamente restrito a funções da própria classe; a cláusula **export** deve ser usada para adicionar visibilidade (2.1, 2.8)
- a cláusula **static** foi substituída por '+' (2.2.2, 2.8.4, 2.7.1)
- a declaração **cvirtual** permite ligar módulos em C e Cm

Apêndice C

Mensagens de Erro do Compilador

Os diagnósticos emitidos pelo compilador Cm, seus significados e possíveis correções são apresentados a seguir. Algumas mensagens (ou seus significados) variam com o ambiente em que o compilador é executado. Todas são passíveis de alteração em versões subseqüentes.

Mensagens de erro completas incluem o nome do arquivo/classe onde o erro foi detectado, o número da linha, e alguma informação sobre a posição do erro na linha, se possível. Nas descrições, XXX, YYY, etc. significam uma ou mais mensagens adicionais ou particulares para cada caso.

(warning) Alphabetic character next to digit -- space assumed há um caráter alfabético imediatamente após um número—pelo menos um espaço em branco deve ser inserido

(warning) Bad hexadecimal digit apenas os dez dígitos de 0 a 9, além dos caráteres "A" até "F", tanto maiúsculos como minúsculos, podem ser usados numa constante na base 16

(warning) Could not remove compiled instance XXX o compilador não pôde remover um arquivo com uma instância obsoleta da classe em compilação. Verifique a variável de ambiente descrevendo o diretório de classes compiladas (CMC) e a permissão do arquivo/diretório

(warning) Could not remove data files XXX não foi possível atualizar os arquivos de parâmetros. Verifique a variável de ambiente descrevendo o diretório de dados (CMDATA) e as permissões de acesso

(warning) Identifier XXX not used um identificador foi declarado mas nunca usado. Possível erro de programação

(warning) Illegal escape sequence um caráter '\' ocorreu numa constante caráter ou *string*, mas os caráteres a seguir não formam uma seqüência correta

(warning) Illegal octal digit — decimal constant assumed dígitos 8 e 9 não são permitidos numa constante octal (ao contrário de vários compiladores C e da primeira definição de Cm)

(warning) Non-hex digit next to number — space assumed caracteres alfabéticos não hexadecimais adjacentes a uma constante na base 16 não formam parte do número

(warning) Possible loss of precision um objeto teve seu valor copiado a outro que (possivelmente) ocupa menos espaço em memória. Provável perda de dígitos significativos (a mensagem pode ou não ser significativa, dependendo da arquitetura local e da implementação do compilador C)

(warning) Repeated storage class uma classe de armazenamento foi especificada mais de uma vez na mesma declaração

(warning) Too high hexadecimal number — truncated uma constante de tipo **char** definida como um número na base 16 ultrapassou 255 (0xff)— ocorre truncamento

(warning) Too high octal sequence — truncated uma constante caráter definida como uma seqüência octal ultrapassou 255 (\377)—subtrai-se 255 da constante

(warning) Too many alternate XXX directories — ignored a variável de ambiente definindo o diretório para XXX contém diretórios alternativos em excesso (o excedente é ignorado)

(warning) Variable XXX not used a variável XXX foi declarada mas nunca usada. Possível erro de programação

Cannot allocate input buffer memória exaurida ao iniciar leitura de arquivo

Cannot allocate new class field descriptor memória exaurida ao tentar definir uma classe incluída

Cannot generate code for class XXX não é possível produzir código para a classe XXX—provavelmente faltam parâmetros a definir

Cannot get data directory XXX diretório de dados para o compilador inacessível. Verificar variável de ambiente CMDATA

Cannot inherit class twice uma classe não pode ser herdada mais que uma vez pela mesma classe herdeira, com os *mesmos* parâmetros reais

Cannot open data file for class XXX arquivo de parâmetros da classe XXX inacessível. Verificar variável de ambiente CMDATA e permissão de acesso. No sistema MS-DOS, verificar variável de configuração FILES (total de arquivos abertos simultaneamente); se estiver com o valor máximo, simplificar estrutura de classes

Cannot open source file XXX arquivo fonte XXX não foi encontrado. Classe inexistente ou variável de ambiente CMSRC incorreta. Em MS-DOS, conferir o valor da variável FILES

Cannot rename identifier XXX, not inherited o identificador XXX não proveio de herança, portanto não pode ser rebatizado

Cannot set compiled C directory XXX diretório de instâncias compiladas XXX inacessível. Verificar variável CMC e permissões de acesso

Cannot set data file arquivos de parâmetros para classe não puderam ser determinados. Variável de ambiente CMDATA incorreta ou memória exaurida

'case' after 'default' not allowed nenhum rótulo **case** pode ocorrer após o rótulo **default**

'case' outside a 'switch' o rótulo **case** não pode ser usado externamente ao comando **switch**

Character constant cannot be empty a seqüência '' não é um caráter válido. Caráter faltando ou pretendia-se *string*

Character constant must contain a single character deve haver apenas um caráter (ou seqüência precedida por "\") entre ' e '. Use aspas para definir cadeias de caráteres

Circular class hierarchy detected ocorreu circularidade na hierarquia de classes (isto é, uma classe terminou definida em termos de si mesma)

Class cannot include itself uma classe não pode usar instanciações de si mesma (apenas se o nome da classe não for seguido por (), significando o próprio tipo sendo compilado)

Class component XXX is not a function o componente da classe selecionado não é uma função e portanto não pode ter parâmetros

Class name XXX does not match file name um arquivo deve ter o mesmo nome da classe por ele definida (além da extensão .cm)

Class parameter for XXX should be a type o parâmetro real da classe deve ser uma expressão descrevendo um tipo

Constant expression required o valor de seleção de um rótulo case deve ser uma constante determinável durante a compilação. O mesmo se aplica ao valor inicial de variáveis cuja classe de armazenamento é "+" e a parâmetros de classe

Could not add symbol table entry — probably disk full não foi possível atualizar arquivo temporário para símbolos. Verificar capacidade de disco disponível

Could not create C file for top-level class o compilador não pôde criar programa com a função *main()*. Provavelmente disco exaurido ou permissão de escrita no diretório corrente negada

Could not create makefile impossível produzir arquivo de dados para *make*. Disco exaurido ou acesso negado ao diretório corrente.

Could not create temporary file for symbol table
arquivo temporário para *swapping* da tabela de símbolos não pôde ser criado; disco exaurido ou diretório temporário inexistente. Em MS-DOS, verificar variável FILES

Could not declare class classe não declarada—memória exaurida

Could not find symbol table file arquivo temporário com tabela de símbolos perdido. Erro grave no sistema de arquivos

Could not open C header file XXX impossível criar arquivo de inclusão para instância da classe corrente. Verificar variável de ambiente CMC e permissão de gravação no diretório. Em MS-DOS, verificar variável de configuração FILES

Could not open C output file XXX impossível criar arquivo de código para instância da classe corrente. Verificar variável de ambiente CMC e permissão de gravação no diretório. Em MS-DOS, verificar variável FILES

Could not restore symbol table tabela de símbolos não pôde ser restaurada a partir do arquivo temporário. Erro no sistema de arquivos

Could not restore symbol table entry símbolo perdido enquanto tabela era restaurada. Memória exaurida ou erro no sistema de arquivos

Could not resume output to [XXX] arquivo XXX com código compilado não pôde ser reaberto após inclusão de classe. Erro no sistema de arquivos. Verificar mudança de permissão no diretório de CMC

Could not update makefile list impossível adicionar classes à lista para *makefile*. Memória exaurida

Could not write to file (probably disk is full) não é possível continuar gravando código. Provavelmente disco exaurido

Duplicated 'case' selector seletores repetidos em comando **switch**

Enum value conflict in XXX dois ou mais componentes de tipo **enum** não podem ter o mesmo valor

Error in lexical level symbol table prefix — probably disk full problemas no *swapping* da tabela de símbolos

Function cannot return a value funções declaradas como **void** não podem retornar um valor

Function required apenas funções podem ser seguidas por parâmetros reais entre parênteses

Function should return a value funções não definidas como **void** não podem usar **return** sem um argumento

Identifier XXX cannot be redefined tentativa de redefinir um nome XXX

Identifier XXX is not a class name identificador XXX não pode ser usado como classe

Identifier XXX is not a label nomes não definidos como rótulos não podem ser usados no comando **goto**

Identifier XXX redefined as a label o rótulo XXX já foi definido anteriormente

Identifier XXX not declared nome XXX foi usado antes de ser declarado

Illegal data pathname variável de ambiente CMDATA com valor incorreto

Illegal username or directory error for C filename diretório descrito em CMC incorreto—em *UNIX*, verificar diretórios usando HOME de usuário

Illegal decimal point ponto decimal incorreto, ou duplicado, em constante numérica

Illegal digit dígito ilegal para a base numérica usada

Illegal lefthand side for expression a expressão à esquerda de um operador de cópia não é uma referência válida

Incompatible cast conversion tentativa de executar operação de *cast* impossível

Incompatible storage class combination: XXX uma ou mais classes de armazenamento formam uma combinação ilegal para o objeto declarado

Incomplete character constant faltando "'" em constante caráter

Integer expression required apenas expressões de tipo integral podem ser usadas para especificar tamanhos de *arrays* e membros de estruturas

XXX is not a parameter função ou classe foi referenciada com parâmetros nominais, mas XXX não é um parâmetro declarado daquela função ou classe

XXX is not a proper class component não há nenhum componente da classe com nome XXX

XXX is not a proper member a estrutura/união não tem nenhum membro com nome XXX

Label XXX doubly defined rótulo XXX declarado em dois ou mais pontos no mesmo bloco léxico

Label 'XXX' not declared nome XXX usado em comando **goto** mas nunca declarado

Memory exhausted — module XXX, line YYY falta de memória; simplifique o programa ou a estrutura de classes; aumente o espaço disponível ao compilador, se possível.

Misplaced 'break' statement o comando **break** somente pode ser usado internamente a comandos de iteração e **switch**

Misplaced 'continue' statement comando **continue** somente pode ser usado internamente a comandos de iteração

Missing ';' comandos devem ser terminados por ponto-e-vírgula

Missing a 'return' statement função de tipo diferente de **void** pode terminar execução sem retornar um valor

Missing parameters for class XXX with no default instanciação de classe XXX não tem todos os parâmetros reais e não há valores *default* definidos

Missing parameters for component XXX componente XXX de classe é uma função e deve ter parâmetros reais

Missing value for constant XXX XXX declarado como uma constante, deve ser seguido de valor

Nested function declaration not allowed funções não podem ser aninhadas

Nonterminated string constant caráter "\n" terminador de *string* não encontrado

Not allowed in an expression item estranho ao contexto encontrado numa expressão

Not allowed type in enum value XXX uma expressão para componentes de tipo **enum** deve ser de tipo integral

Not implemented yet característica da linguagem ainda não implementada—contacte o autor

Only one 'default' is allowed per 'switch' somente pode haver um rótulo **default** em cada comando **switch**

Parameter may not be an instance of own class o tipo de um parâmetro de função não pode ser uma instanciação da própria classe (pode ser usado o nome da classe sem () para indicar a instanciação sendo compilada)

Parameter may not be void parâmetros de função não podem ser do tipo **void** (mas podem ser *apontadores* para esse tipo)

Pointer required at left side of '->' operator o operador -> exige um operando esquerdo do tipo apontador

Pointer to struct/union/class type required o operando do lado esquerdo deve ser um apontador para classe, **struct** ou **union**

Recursive class definition not allowed uma objeto definido na classe não pode ser uma instanciação da própria classe. Verificar circularidade na hierarquia de classes

Recursive type definition not allowed um tipo não pode ser definido em termos de si mesmo (à exceção de apontadores para **struct**)

Selector expresson type must be integral seletor de rótulo **case** deve ser de tipo integral

'self' only allowed inside a function o nome self não pode ser usado fora de funções

Structure/union member selector can have no parameters um membro de estrutura ou união foi usado incorretamente como função

Structure/union/class type required apenas classes, estruturas e uniões podem sofrer seleção de membros

Structure/union bit field cannot be indexed membros de estrutura/união com tamanho definido por uma constante não podem ser declarados como vetores

Syntax error — XXX/YYY/.../ZZZ erro sintático. Segue uma lista dos possíveis símbolos válidos

Test expression should be scalar a expressão usada para testes em comandos **if**, **while**, **do** e **for** não pode ter tipo estruturado, devendo ser redutível a um número

Too long string constant uma cadeia de caráteres constante excedeu o limite do compilador (ou não foi terminada com ", o que é mais provável). Verifique o término e use a opção de compilação *-as*

Too many class parameters instanciação de classe com parâmetros reais excedentes

Too many lexical levels nível léxico máximo excedido. Reduza a profundidade do aninhamento de blocos ({—}); comunique ao autor

Too many nested struct/unions o nível máximo de aninhamento de estruturas/uniões foi excedido; contacte o autor

Too many nested switch statements nível máximo de aninhamento em comandos *switch* excedido; contacte o autor

Too many parameters in function call - ignored parâmetros extras na chamada de função ignorados

Type mismatch: cannot take address of constant expression

Type mismatch: illegal ternary switch

Type mismatch: invalid types

Type mismatch: not a pointer to release

Type mismatch: not a pointer type

Type mismatch: not a valid type to index

Type mismatch: not an array or pointer to release

Type mismatch: operator requires integral type(s)

Type mismatch: operator requires scalar or pointer types

Type mismatch: operator requires scalar type(s)

Type mismatch: suspicious pointer operation

Type mismatch: wrong pointer arithmetic tipos incompatíveis em expressão; operação inválida para os objetos envolvidos

Type mismatch in class parameter XXX erro de tipo na instanciação da classe XXX

Type mismatch in function arguments tipo incorreto na chamada de função

Type mismatch in function return função devolve expressão com tipo diferente daquele com que foi declarada

Type XXX should be defined um tipo foi declarado antecipadamente mas nunca foi definido

Type stack overflow pilha de tipos exaurida. Expressão provavelmente muito complexa.

Unknown array size Expressão para limites de vetor incorreta ou ausente

Unknown identifier (XXX) to rename não há nenhum identificador XXX a mudar de nome

Unknown structure/union member size expressão definindo tamanho de membro de estrutura/união (após ':') incorreta ou não-constante

Unresolved virtual function XXX a função XXX foi declarada **virtual** numa classe herdada, mas nunca foi definida

Variable may not be a void o tipo padrão **void** não pode ser usado como tipo de uma variável (**void** * suposto)

Variable may not be instance of its own class uma variável de classe não pode ter como tipo uma instanciação diferente da própria classe

Virtual function cannot be redefined funções definidas como "virtuais" não podem ter código definido, apenas a declaração

'void' pointer arithmetic not allowed Não são permitidas adição, subtração ou indexação de apontadores para o tipo **void**, sem que seja feito um *cast*

XXX (INTERNAL ERROR) o compilador descobriu que não está operando propriamente. Esta mensagem *nunca* deve ocorrer quando compilando programas corretos. Pode ocasionalmente aparecer após certos erros cuja recuperação não foi completa (nunca como a primeira mensagem). Caso seja emitida, comunique ao autor sua ocorrência e a mensagem XXX

Apêndice D

Gramática Yacc

A versão da gramática aqui apresentada é uma simplificação da usada no tradutor Cm. Não estão presentes as ações em C executadas após cada regra, nem as produções adicionadas para prever e tratar erros sintáticos. Algumas regras não seguem a abordagem mais simples ou "natural", devido à necessidade de inserir ações C e, principalmente, evitar ambigüidades na gramática.

```
/*****************************************************************

 Cm: C modular and polymorphic


 Module: Gram.y - Yacc grammar

 Language: Standard/ANSI C

 Author: Carlos A. Furuti

 Version:

*****************************************************************/

/*-------------- Yacc Stack Type Declaration ----------------*/

%union
        {
          int           Number;
          char          *String;
```

```
        TYPE_DESC       *Type;
        PAR_DESC        *Parameter;
        CLASS_DESC      *Class;
        STR_DESC        *Struct;
        FIELD_DESC      *FList;
        EXPRESSION      *Expression;
        SYMBOL          *Symbol;
        llist           *List;
        node2           *BNode;
        output_desc     *Output;
        FUNCT_DESC      *Function;
        voidp           General;
      }

/*---------------- TOKEN DEFINITIONS ----------------------*/

%token <Number> AUTO BREAK
%token <Number> CASE CHAR_SY CLASS CONST CONTINUE
%token <Number> DEFAULT DO DOUBLE
%token <Number> ELSE ENUM EXPORT FLOAT_SY FOR
%token <Number> GLOBAL GOTO
%token <Number> IF INT IMPORT INHERIT
%token <Number> LONG NEW
%token <Number> RELEASE REGISTER RENAME RETURN
%token <Number> SELF SHORT SIZEOF STRUCT SWITCH
%token <Number> TYPE
%token <Number> UNION UNSIGNED USE
%token <Number> VOID WHILE
%token <Number> TYPE_ID CLASS_ID FUN_ID
%token <Number> COMPILED_ARG_LIST_END END_OF_FILE

%token <String> IDENTIFIER PIDENTIFIER
%token <String> CHAR STRING DEC_INT HEX_INT OCT_INT FLOAT
%token <Number> IND_SELECTION INC_OP DEC_OP
%token <Number> LSHIFT_OP RSHIFT_OP LE_OP GE_OP
%token <Number> EQ_OP NEQ_OP
%token <Number> LOG_AND LOG_OR ASGN_OP

%token <Number> ',' '=' '?' ':' '|' '^' '&' '>' '<' '+' '-'
%token <Number> '*' '/' '%' '~' '!' '.'

/* Precedence/Evaluation Order, Logic-Arithmetic Operators */
```

```
%left   ','                     /* sequential evaluation */
%right  '=', ASGN_OP            /* assignment operators */
%right  '?', ':'                /* conditional (ternary) */
%left   LOG_OR                  /* logical or */
%left   LOG_AND                 /* logical and */
%left   '|'                     /* bitwise or */
%left   '^'                     /* bitwise xor */
%left   '&'                     /* bitwise and */
%left   EQ_OP NEQ_OP            /* (in)equality */
%left   '<', '>', LE_OP, GE_OP  /* relational */
%left   LSHIFT_OP, RSHIFT_OP    /* bit shifting */
%left   '+', '-'                /* additive (binary) */
%left   '*', '/', '%'           /* multiplicative */
%right  '~', '!', INC_OP DEC_OP /* log.not, prefix inc/dec */
%left   '.', IND_SELECTION      /* (in)direct selection */

%start program

/*-------------  Cm SYNTAX RULES START HERE ---------------*/

%%

program         :  class_def
                ;

/*------------------- Module Declaration -------------------*/

class_def       :  CLASS  PIDENTIFIER
                   '('  mod_arg_list  ')'
                   use_clause
                   import_clause
                   inherit_clause
                   decls
                ;

/*------------------ Preliminary Clauses -----------------*/

use_clause      :  USE  use_list  scolon
                |  /* empty */
                ;
use_list        :  use_list  ','  use_id
```

```
                | use_id
                ;
use_id          : IDENTIFIER '=' type
                ;
import_clause   : IMPORT import_list scolon
                | /* empty */
                ;
import_list     : import_list ',' imported_class
                | imported_class
                ;
imported_class  : IDENTIFIER
                ;
inherit_clause  : INHERIT inherit_list scolon
                | /* empty */
                ;
inherit_list    : inherit_list ',' inhren_clause
                | inhren_clause
                ;
inhren_clause   : inherited_class
                | inherited_class RENAME rename_list
                ;
inherited_class : class_type
                ;
rename_list     : rename_list ',' rename_id
                | rename_id
                ;
rename_id       : EXPORT ren_id
                | ren_id
                ;
ren_id          : IDENTIFIER '=' rename_end
                ;
rename_end      : type
                | ridentifier
                ;
decls           : n_e_decls
                | /* empty */
                ;
declaring       : declaration
                | funct_def
                ;
n_e_decls       : n_e_decls declaring
                | declaring
```

```
                ;

/*----  Module argument list (class formal parameters) ----*/

mod_arg_list    :  no_emp_arg_list
                |  /* empty */
                ;
no_emp_arg_list :  no_emp_arg_list  scolon  unit_arg_list
                |  unit_arg_list
                ;
unit_arg_list   :  type_name  list_init_ident
                |  TYPE  list_type_ident
                ;
list_type_ident :  list_type_ident  ','  type_ident
                |  type_ident
                ;
type_ident      :  IDENTIFIER
                |  IDENTIFIER  '='  type_name
                ;

/*-------------------------------------------------------------
   Type declarations
-------------------------------------------------------------*/

type            :  type_name
                |  type_spec
                ;
index_list      :  index_list  index_range
                |  index_range
                ;
index_range     :  '['  list_exp  ']'
                ;
type_name       :  TYPE_ID
                |  std_type
                |  class_type
                |  type_name  '*'
                |  type_name  index_list
                |  type_name  '('  funct_arg_list  ')'  '*'
                ;

/*----------------- Standard Cm Types -------------------*/
```

```
intlen          :  SHORT
                |  LONG
                ;
std_scalar      :  CHAR_SY
                |  intlen
                |  intlen  INT
                |  INT
                ;
std_type        :  UNSIGNED std_scalar
                |  std_scalar
                |  FLOAT_SY
                |  DOUBLE
                |  VOID
                ;

/*- Class Instantiation: class name and actual parameters */

class_type      :  PIDENTIFIER '(' ac_par_class ')'
   class_inst
                ;
ac_par_class    :  ac_par_class_list
                |  /* empty */
                ;
ac_par_class_list: ac_par_class_list  ','  unit_par_class
                |  unit_par_class
                ;
unit_par_class  :  expression
                |  IDENTIFIER  ':'  expression
                |  type_name
                |  IDENTIFIER  ':'  type_name
                ;
class_inst      :  class_def END_OF_FILE
                |  END_OF_FILE
                ;

/*---------------- Struct/Union Definitions ---------------*/

type_spec       :  struct_specifier
                |  enum_specifier
                ;
str_uni         :  STRUCT
                |  UNION
```

```
                ;
struct_specifier: str_uni  '{'  list_str_dclrt  '}'
                ;
list_str_dclrt  : list_str_dclrt  str_declaration
                | str_declaration
                ;
str_declaration : type  list_str_ident  scolon
                ;
list_str_ident  : list_str_ident  ','  str_ident
                | str_ident
                ;
field_name      : IDENTIFIER
                | CLASS_ID
                | FUN_ID
                | TYPE_ID
                ;
str_ident       : fidentifier
                | ':'  expression
                | fidentifier  ':'  expression
                ;
ridentifier     : IDENTIFIER
                | FUN_ID
                ;
fidentifier     : ridentifier
                | TYPE_ID
                | CLASS_ID
                ;
/*------------------- Enumerate types --------------------*/

enum_specifier  : ENUM enum_tag
                ;
enum_tag        : '{'  inner_enum_tag  inner_enum_end  '}'
                ;
inner_enum_tag  : inner_enum_tag  ','  enum_ident
                | enum_ident
                ;
inner_enum_end  : ','
                | /* empty */
                ;
enum_ident      : IDENTIFIER
                | IDENTIFIER  '='  expression
                ;
```

```
/*-------------------------------------------------------
   Types, variables, function declarations
-------------------------------------------------------*/

declaration     :  type_decl
                |  data_decl
                |  funct_decl  scolon
                ;
type_decl       :  TYPE  type_list  scolon
                ;
type_list       :  type_list  ','  type_id
                |  type_id
                ;
type_id         :  IDENTIFIER  '='  type
                |  IDENTIFIER
                ;
data_decl       :  stclass  type  list_init_ident  scolon
                ;
list_init_ident :  list_init_ident  ','  init_ident
                |  init_ident
                ;
init_ident      :  IDENTIFIER
                |  IDENTIFIER  '='  initializer
                ;

/*----------------- Function Declarations ---------------*/

fname           :  PIDENTIFIER
                |  FUN_ID
                ;
funct_decl      :  stclass type_name
                   fname '(' funct_arg_list ')'
                ;
funct_def       :  funct_decl  '{'  decls  stmts  '}'
                ;
funct_arg_list  :  f_arg_list
                |  /* empty */
                ;
f_arg_list      :  f_arg_list  scolon  funct_arg
                |  funct_arg
                ;
```

```
funct_arg       :  type_name  list_init_ident
                ;
stclass         :  '+'
                |  '+'   EXPORT
                |  '+'   CONST
                |  REGISTER
                |  EXPORT
                |  CONST
                |  /* empty */
                ;
initializer     :  '{'  init_exp_list  '}'
                |  '{'  init_exp_list  ','  '}'
                |  expression
                ;
init_exp_list   :  init_exp_list  ','  initializer
                |  initializer
                ;

/*-------------------------------------------------------
   Statements
-------------------------------------------------------*/

compound_stmt   :  '{' decls stmts '}'
                ;
stmts           :  statement_list
                |  /* empty */
                ;
statement_list  :  statement_list  stmt
                |  stmt
                ;
stmt            :  bal_stmt
                |  unbal_stmt
                ;
bal_stmt        :  basic_stmt
                |  bal_while_stmt
                |  bal_for_stmt
                |  bal_ifelse_stmt
                |  bal_switch_stmt
                |  label  bal_stmt
                ;
unbal_stmt      :  unbal_while_stmt
                |  unbal_for_stmt
```

```
                | unbal_ifelse_stmt
                | unbal_if_stmt
                | unbal_switch_stmt
                | label  unbal_stmt
                ;
basic_stmt      : e_stmt
                | compound_stmt
                | do_stmt
                | break_stmt
                | continue_stmt
                | return_stmt
                | goto_stmt
                | null_stmt
                ;
condition       : '('  list_exp  ')'
                ;
if_prefix       : IF condition
                ;
else            : ELSE
                ;
bal_ifelse_stmt : if_prefix  bal_stmt  else  bal_stmt
                ;
unbal_ifelse_stmt: if_prefix  bal_stmt  else  unbal_stmt
                ;
unbal_if_stmt   : if_prefix  stmt
                ;
e_stmt          : list_exp  scolon
                ;
while_prefix    : WHILE  condition
                ;
bal_while_stmt  : while_prefix  bal_stmt
                ;
unbal_while_stmt: while_prefix  unbal_stmt
                ;
do_stmt         : DO  stmt  while_prefix  scolon
                ;
for_prefix      : FOR  '('  for_1_prefix
                | FOR  '(' list_exp for_1_prefix
                ;
for_1_prefix    : scolon  for_2_prefix
                | scolon  list_exp0 for_2_prefix
                ;
```

```
for_2_prefix     :  scolon  ')'
                 |  scolon  list_exp0  ')'
                 ;
unbal_for_stmt   :  for_prefix  bal_stmt
                 ;
bal_for_stmt     :  for_prefix  unbal_stmt
                 ;
switch_prefix    :  SWITCH  '('  list_exp  ')'
                 ;
bal_switch_stmt  :  switch_prefix  bal_stmt
                 ;
unbal_switch_stmt:  switch_prefix  unbal_stmt
                 ;
break_stmt       :  BREAK  scolon
                 ;
continue_stmt    :  CONTINUE  scolon
                 ;
return_stmt      :  RETURN  scolon
                 |  RETURN  list_exp0  scolon
                 ;
goto_stmt        :  GOTO  IDENTIFIER  scolon
                 ;
null_stmt        :  scolon
                 ;
label            :  IDENTIFIER  ':'
                 |  CASE  expression  ':'
                 |  DEFAULT  ':'
                 ;

/*------------------- EXPRESSIONS ----------------------*/

list_exp         :  list_exp0
                 ;
list_exp0        :  expression
                 |  list_exp0  ','  expression
expression       :  cond_exp  ASGN_OP  expression
                 |  cond_exp  '='  expression
                 |  cond_exp
                 ;
cond_exp         :  or_oper_exp  '?'  list_exp0  ':'  cond_exp
                 |  or_oper_exp
                 ;
```

```
or_oper_exp      : or_oper_exp  LOG_OR  and_oper_exp
                 | and_oper_exp
                 ;
and_oper_exp     : and_oper_exp  LOG_AND  bitor_oper_exp
                 | bitor_oper_exp
                 ;
bitor_oper_exp   : bitor_oper_exp  '|'  bitxor_oper_exp
                 | bitxor_oper_exp
                 ;
bitxor_oper_exp  : bitxor_oper_exp  '^'  bitand_oper_exp
                 | bitand_oper_exp
                 ;
bitand_oper_exp  : bitand_oper_exp  '&'  equ_oper_exp
                 | equ_oper_exp
                 ;
equ_oper_exp     : equ_oper_exp  eq_op  rel_oper_exp
                 | rel_oper_exp
                 ;
eq_op            : NEQ_OP
                 | EQ_OP
                 ;
rel_oper_exp     : rel_oper_exp  rel_op  shift_oper_exp
                 | shift_oper_exp
                 ;
rel_op           : GE_OP
                 | LE_OP
                 | '<'
                 | '>'
                 ;
shift_oper_exp   : shift_oper_exp  shift_op  add_oper_exp
                 | add_oper_exp
                 ;
shift_op         : LSHIFT_OP
                 | RSHIFT_OP
                 ;
add_oper_exp     : add_oper_exp  add_op  mult_oper_exp
                 | mult_oper_exp
                 ;
add_op           :  '+'
                 |  '-'
                 ;
mult_oper_exp    : mult_oper_exp  mult_op  unary_op_exp
```

```
                | unary_op_exp
                ;
mult_op         : '*'
                | '/'
                | '%'
                ;
unary_op_exp    : cast_exp
                ;
cast_exp        : prefix_exp
                | '(' type_name ')' cast_exp
                ;
prefix_exp      : postfix_exp
                | incdec_op cast_exp
                | negat_op  cast_exp
                | '*'       cast_exp
                | '&'       cast_exp
                ;
incdec_op       : INC_OP
                | DEC_OP
                ;
negat_op        : add_op
                | '!'
                | '~'
                ;
postfix_exp     : primary_exp
                | postfix_exp incdec_op
                ;
primary_exp     : prim_p2_exp
                ;
prim_p2_exp     : prim_p1_exp
                | prim_p2_exp '[' list_exp0 ']'
                | prim_p2_exp '(' actual_par_list ')'
                | prim_p2_exp '(' ')'
                | prim_p2_exp '.' field_name
                | prim_p2_exp IND_SELECTION field_name
                ;
expr_id         : IDENTIFIER
                | FUN_ID
                ;
prim_p1_exp     : expr_id
                | SELF
                | literal
```

```
                | paren_exp
                | SIZEOF  '(' type_name ')'
                | SIZEOF  '(' prefix_exp ')'
                | NEW     '(' type_name ',' expression ')'
                | RELEASE '(' prefix_exp ')'
                ;
field_ident     : IDENTIFIER
                | PIDENTIFIER
                | FUN_ID
                | TYPE_ID
                ;
field_name      : field_ident
                | field_ident  '('  actual_par_list  ')'
                | field_ident  '('  ')'
                ;
literal         : DEC_INT
                | FLOAT
                | CHAR
                | HEX_INT
                | OCT_INT
                | STRING
                ;
actual_par_list : u_actual_par_list
                | n_actual_par_list
                ;
u_actual_par_list: u_actual_par_list  ','  u_actual_par
                | u_actual_par
                ;
u_actual_par    : expression
                ;
n_actual_par_list: n_actual_par_list  ','  n_unit_par_list
                | n_unit_par_list
                ;
n_unit_par_list : IDENTIFIER  ':'  expression
                ;
paren_exp       : '('  list_exp0  ')'
                ;
scolon          : ';'
                ;

%%
```

Apêndice E

Arquivos do Tradutor

A versão do compilador Cm tornada pública em junho de 1991 para *UNIX* e MS-DOS é composta pelos arquivos relacionados a seguir. A lista é produzida pelo programa *wc* (*word count*) [At 87], que indica, para cada arquivo e para o total dos arquivos, o número de linhas, palavras e caráteres (nessa ordem, da esquerda para a direita em cada linha).

```
  74   268   2042  header/analex.e
 315   890   6775  header/cm.e
  47    96    771  header/cm.h
 133   633   4758  header/code.e
  75   340   2789  header/code.h
  93   388   2905  header/debug.e
  74   250   2248  header/decl.e
  65   252   1812  header/environ.e
 181   577   4424  header/error.e
 218   819   6789  header/expr.e
  44   142   1136  header/file.e
  90   367   2703  header/filer.e
 101   353   2399  header/global.h
  38    60    589  header/gmem.e
  40   128   1036  header/llist.e
  66   333   2568  header/make.e
  50   208   1687  header/symtab.e
  63    63    705  header/tknames.h
 149   652   5084  header/type.e
  53   155   1181  header/utils.e
  34    95    828  header/wildcard.e
```

1093	3809	27455	analex.c
321	805	5813	cm.c
180	671	4893	cmlib.c
802	2779	21817	codecdcl.c
754	2751	22374	codecgen.c
963	3014	25943	codeexpr.c
354	1210	10018	codemisc.c
602	1874	13780	debug.c
686	2395	19433	decl.c
659	2476	18956	environ.c
637	2272	18322	error.c
257	1027	6260	expfn.c
1694	6013	51173	expr.c
308	1004	7680	file.c
300	896	7432	filer.c
69	189	1718	gmem.c
3370	8989	79234	gram.y
201	635	5388	llist.c
384	1401	11317	make.c
771	2761	21899	symtab.c
1147	4113	33551	type.c
171	590	4644	utils.c
277	837	6943	wildcard.c
86	301	2010	y_tab.h
4337	16410	131661	ytab.c
22426	76291	614943	--Total

Arquivos com extensão ".c", ".e" e ".h" são código-fonte na linguagem C. Os arquivos y_tab.h e ytab.c são gerados por Yacc a partir da gramática gram.y. Ytab.c é o analisador sintático e contém código modificado para tornar as mensagens de erro sintático mais informativas ao usuário. Cmlib.c não faz parte do compilador propriamente dito, sendo a biblioteca de suporte a execução para programas Cm.

Bibliografia

[Aa 86] Aho, Alfred V., Sethi, Ravi & Ullman, Jeffrey D. *Compilers Principles, Techniques, and Tools.* Addison-Wesley Publ. Co., 1986.

[An 83] *The Programming Language Ada Reference Manual* American National Standards Institute Inc., ANSI/MIL-STD-1815A-1983. Lecture Notes in Computer Science, ed. G. Goos, J. Hartmanis, Springer-Verlag, 1983.

[At 89] AT&T, *UNIX System V Release 3.2 Programmer's Guide.* Prentice-Hall, Englewood Cliffs, NJ, 1989.

[At 86] AT&T, *UNIX System V Release 3 Programmer's Reference Manual.* AT&T, 1986.

[At 84] AT&T, *C++.* AT&T, Nov. 1984.

[At 87] AT&T, *UNIX System V User's Reference Manual* Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Bo 87] Borland International *Turbo C Reference Guide.* Borland International Inc. Scotts Valley, Ca., 1987.

[Br 82] Brooks, *The Mythical Man-Month Essays on Software Engineering.* Addison-Wesley Reading Mass., 1985.

[Bra87] Braunschober, W. *COMPAS: Compatible Pascal.* ACM SIGPLAN Notices 22(3): 78–82, Mar. 1987.

[Ca 91] Calliss, F. W. *A Comparison of Module Constructs in Programming Languages.* ACM SIGPLAN Notices 26(1): 38–46, Jan. 1991.

[Cal89] Cardelli, L., Donahue, J., Jordan, M., Kalswo, B., Nelson, G. *The Modula-3 Type System.* Proceedings of POPL 1989.

[Car89] Carneiro, L. C. D., *Compilação de Linguagens de Comando de Alto Nível.* Dissertação de Mestrado, Unicamp, março de 1989.

[CK 90] Carvalho, C. S. R., Kowaltowski, T. *On Open Arrays and Variable Number of Parameters.* Unicamp, IMECC, Relatório Técnico, 1990.

[CW 85] Cardelli, L. & Wegner, P. *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, **17**(4): 471–522, Dec. 1985.

[Ch 86] Christian, K. *A Guide to Modula-2.* Springer-Verlag Inc., NY, 1986.

[Co 87] Cox, B. *Object-Oriented Programming – An Evolutionary Approach.* Addison-Wesley Publ. Reading, Mass., 1987.

[DL 87] Drummond, R. & Liesenberg, H. K. E., *Requisitos para um Ambiente de Desenvolvimento de PROGRAMAS.* I Encontro IBM de Ciência e Tecnologia da Informática. Rio de Janeiro, nov. 1987.

[Do 88] Dony, C. *An Object-oriented Exception Handling System for an Object-oriented Language.* Proceedings of European Conference on Object-Oriented Programming 88 146–161, ed. S. Gjessing, K. Nygaard, Oslo. Reprinted in Lecture Notes in Computer Science Springer-Verlag, 1988.

[DoS88] Donnelly, C., Stallman, R. *Bison Reference Manual.* Free Software Foundation, 1988.

[Dr 89] Drummond, R. *LegoShell Linguagem de Computações.* III Simpósio Brasileiro de Engenharia de *Software*, pp. 1–13, Recife, out. 1989.

[DS 88] Drummond, R. & da Silva, F. Q. B. *Manual de Referência — Linguagem Cm.* Unicamp IMECC-DCC, 1988.

[Er 90] Eriksson, M. *A Correct Example of Multiple Inheritance.* ACM SIGPLAN Notices **25**(7): 7–10, Jul. 1990.

[FD 89] Furuti, C. A. & Drummond, R., *Cm, Linguagem e Implicações.* Unicamp IMECC-DCC (exame de qualificação para mestrado), 1989.

[Fe 79] Feldman, S. *Make — a Program for Maintaining Computer Programs.* Software — Practice & Experience **9**(4): 255–265 Apr. 1979.

[FG 82] Feuer, A. R. & Gehani, N. H. *A Comparison of the Programming Languages C and Pascal.* ACM Computing Surveys **14**(1): 73-92 Mar. 1982.

[Go 83] Goldberg, A. *Smalltalk-80 The Language and its Implementation.* Addison-Wesley Reading Mass., 1983.

[Gr 89] Grogono, P. *Comments, Assertions and Pragmas.* ACM SIGPLAN Notices **24**(3): 79-84, Mar. 1989.

[Gru86] Grune, D. *Generic Packages in* C. ACM SIGPLAN Notices **21**(8): 31-39 Aug. 1986.

[Ha 90] Hara, C. S. *Utilização de um Banco de Dados Orientado a Objetos em um Ambiente de Desenvolvimento de Software.* Dissertação de Mestrado, Unicamp, 1990.

[HC 89] Hwu, W. W., Chang, P. P. *In-line Function Expansion for Compiling* C *Programs.* Proceedings of SIGPLAN 89 Conference on Programming Languages Design and Implementation. Portland, Oregon, Jun. 1989.

[Ho 84] Horowitz, E. *Fundamentals of Programming Languages* 2nd Ed. Springer-Verlag, 1984.

[HU 69] Hopcroft, John E., Ullman, Jeffrey D., *Formal Languages and Their Relation to Automata.* Addison-Wesley Publ. Co., 1969.

[Jo 78] Johnson, S. C. *Yacc: Yet Another Compiler-Compiler.* Bell Laboratories, 1978.

[JW 75] Jensen, K., Wirth, N. *Pascal User Manual and Report.* Springer-Verlag, NY, 1975.

[KC 75] Kernighan, B. W., Cherry, L. L. *A System for Typesetting Mathematics.* Communications of ACM **18**(3): 151-157, Mar. 1975.

[Ke 75] Kernighan, B. W. *Ratfor — A Preprocessor for Rational Fortran* Software — Practice and Experience **5**(4): 395-406, Oct-Dec. 1975.

[Ke 82] Kernighan, B. W. *PIC — A Language for Typesetting Graphics.* Software — Practice and Experience **12**(1): 1-21, 1982.

[Kn 73] Knuth, D. E. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley Publ. Co., Reading, Mass., 1973.

[KP176] Kernighan, B., Plauger, P. J. *Software Tools.* Addison-Wesley Publ. Co., Reading, Mass., 1976.

[KP 84] Kernighan, B. W., Pike, R. *The UNIX Programming Environment.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1984.

[KR 78] Kernighan, B., Ritchie, D. *The C Programming Language.* Prentice-Hall Inc., Englewood Cliffs, NJ, 1978.

[KR 88] Kernighan, B., Ritchie, D. *The C Programming Language*, 2nd. ed., Prentice-Hall Inc., Englewood Cliffs, NJ, 1988.

[Le 88] Lemkin, P. F. *PSAIL: A Portable SAIL to C Compiler -- Description and Tutorial.* ACM SIGPLAN Notices **23**(10): 149-171, Oct. 1988.

[Li 91] Liu, C. *On the Object-Orientedness of* C++. ACM SIGPLAN Notices **26**(3): 63-67, Mar. 1991.

[Me 87] Meyer, B. *Reusability: The Case for Object-Oriented Design.* IEEE Software **4**(2): 50-64 Mar. 1987.

[Mee90] Meek, B. *Failure is Not Just One Value.* ACM SIGPLAN Notices **25**(8): 80-83, Aug. 1990.

[Mi 87] Microsoft Corporation *Microsoft C Language Reference.* Microsoft Corporation, USA/Canada, 1987.

[MNM87] Meyer, B., Nerson, J., Matsuo, M. *Eiffel: Object-Oriented Design for Software Engineering.* Proceedings of 1st European Software Engineering Conference, 221-229. Edit. H. K. Nichols e D. Simpson. Strasbourg, França, 1987.

[Mi 78] Milner, R. *A Theory of Type Polymorphism in Programming.* Journal of Computer and System Sciences **17**: 348-375.

[Pa 88] Park, J. C. H. *y++: A Yacc Preprocessor for Certain Semantic Actions.* ACM SIGPLAN Notices **23**(6): 97-106, Jun. 1988.

[PC 88] Perelgut, S., Cordy, J. R. *Turing Plus: A Comparison with* C *and Pascal.* ACM SIGPLAN Notices **23**(1): 137-143, Jan. 1988.

[QR 90] Quercia, V., O'Reilly, T. *X Window System User's Guide.* O'Reilly & Associates, Inc. May 1990.

[Re 82] Rentsch, T. *Object Oriented Programming.* ACM SIGPLAN Notices **17**(9): 76-87, Sep. 1982.

[Sa 88] Sakkinen, M. *On the Darker Side of* C++. Proceedings of European Conference on Object-Oriented Programming 88 162–176, ed. S. Gjessing, K. Nygaard. Oslo, 1988.

[Se 81] Sethi, R. *Uniform Syntax for Type Expressions and Declarators*, Software — Practice & Experience **11**(6): 623–628, Jun. 1981

[SLD 88] da Silva, F. Q. B., Liesenberg, H. K. E. & Drummond, R. *Programação em Cm.* Anais do XV SEMISH, Rio de Janeiro, RJ, jul. 1988, pp. 101-102.

[St 82] Stroustrup, B. *Classes: An Abstract Data Type Facility for the* C *Language*. ACM SIGPLAN Notices **17**(1): 42–52, Jan. 1982.

[SF 85] Scheiner, A. T. & Friedman Jr., H. G. *An Introduction to Compiler Construction with UNIX*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1985.

[Su 89] Sun Microsystems, *Sun* C++ *Programmer's Guide*. Sun Microsystems, Inc. 1989.

[VMD89] Victorelli, E. Z., Magalhães, G. C., Drummond, R., *Mecanismo de Gerenciamento de Versões e Configurações do A_HAND*, Revista Brasileira de Computação **5**(2): 3–9, dez. 1989.

[We 90] Wegner, P. *Concepts and Paradigms of Object-Oriented Programming*, OOPS Messenger, ACM Press, **1**(1), Aug. 1990.

[WO 90] Wong, L. S., Ool, B. C. *Treating Failure as a State*. ACM SIGPLAN Notices **25**(8): 24–26, Aug. 1990.

[Yo 83] Young, S. J. *An Introduction to Ada*. Ellis Horwood Limited Publishers Chichester, 1983.

Índice