Projeto de uma Linguagem de Programação

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. Claudio Sergio Da Rós de Carvalho e aprovada pela comissão julgadora.

Campinas, 22 de agosto de 1989.

Orientador: _____
Prof. Dr. Tomasz Kowaltowski

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

"Se você perceber que há quatro maneiras de uma coisa dar errado e driblar as quatro, uma quinta surgirá do nada."

A Lei de Murphy. A. Bloch

Aos meus pais,

meus irmãos, parentes

e a todos que se

cansaram de ouvir:

"No próximo mês, sai!"

Sumário

A proposta deste trabalho é a apresentação dos aspectos principais da linguagem de programação MC, que foi engendrada com o objetivo de incorporar algumas facilidades, chamadas de baixo nível, da linguagem C à estrutura da linguagem Modula-2.

Como evoluções de MC em relação a Modula-2 e C podem-se citar, entre outras, a inclusão de tratamento de exceções, vetores com limites abertos, subprogramas com número variável de parâmetros e processos. O resultado é uma linguagem simples e precisa, de propósito geral mas adequada à programação de sistemas.

Conteúdo

Capítulo 1

Introdução

A linguagem de programação Modula-2 [Wirth 85] é descendente de Pascal [JensenWirth 85] e Modula [Wirth 77], introduzindo os conceitos de *módulos* (próprios para o desenvolvimento de sistemas relativamente grandes com diferentes grupos responsáveis por suas várias partes componentes e com interfaces bem definidas entre eles; cf. [Crawford 85,Wirth 88]) e *processos* (como chave para facilidades para multiprogramação). É uma linguagem adequada à programação de sistemas de uma maneira estruturada. Apresenta regras rígidas de verificação de tipos e mecanismos para burlá-las explicitamente. O acesso às facilidades de baixo nível providas pela linguagem é feito de maneira ordenada (por exemplo, a manipulação de endereços).

A linguagem de programação C [KernRit 78] é de propósito geral, embora esteja intimamente associada com o sistema UNIX.[1] É uma linguagem de alto nível mas apresenta facilidades que poderiam ser associadas às de baixo nível. Sua generalidade e falta de restrições tornam-na mais conveniente para várias classes de problemas do que outras linguagens mais poderosas. Possui um grande conjunto de operadores embora não realize verificação estrita de tipos.[2]

O projeto da linguagem MC visa reunir as facilidades da linguagem C e sua generalidade à estrutura da linguagem Modula-2. Adições importantes são os conceitos de *exceções*, como uma solução intermediária entre as propostas por Ada [DraftAda 83] e Chill [Smedema 84], *vetores com limites abertos* (uma extensão de vetores abertos[3] do padrão ISO para a linguagem Pascal, Nível 1; cf. [ISOPascal 82,C oper 83]), *subprogramas com um número variável de parâ-*

[1] UNIX é marca registrada da AT&T.

[2] Por exemplo, um caractere pode ser interpretado como um inteiro sempre que necessário, sem nenhuma conversão explícita.

[3] Em inglês: *conformant arrays*.

metros (com uma descrição sintática que denota essa particularidade) e uma maior precisão nas regras de compatibilidade entre tipos. O resultado é uma linguagem de propósito geral, muito próxima sintaticamente de Modula-2, mas com algumas características de C que permitem uma utilização ordenada, segura e eficiente de seus recursos de baixo nível.

O presente texto trata dos seguintes itens:

- Linguagens Modula-2 e C: histórico de seus desenvolvimentos, características, elogios e críticas feitas por diversos autores.
- Linguagem MC: Descrição de suas características principais e comparação com as linguagens Modula-2 e C.
- Apresentação dos aspectos de MC: exceções, vetores com limites abertos, passagem de parâmetros, número variável de parâmetros em subprogramas, processos e compatibilidade entre tipos. Serão feitas comparações entre soluções adotadas por várias linguagens, principalmente Modula-2, C, Ada e Chill.
- Implementação: Proposta de um esquema de execução simples para MC.
- Conclusão: Avaliação da linguagem MC de acordo com alguns critérios de desenvolvimento de linguagens e propostas de alterações na linguagem.

Um documento auxiliar e indispensável para a leitura deste texto é o *Manual de Referência da Linguagem MC* [Carvalho 89a].

Cabe ressaltar que a idéia principal do trabalho é fazer um exercício de projeto de linguagem de programação e mostrar que é viável se ter uma linguagem com grande parte das facilidades de C mas sem as suas inconveniências. Além disso, a prática de sintetizar novas linguagens mais poderosas é atual. Veja, por exemplo, as linguagens *Oberon* [Wirth 88], *Modula-3* [Cardelli et al. 88] e *Turing* [HoltCordy 88], que são trabalhos bem recentes.

Capítulo 2

Linguagem Modula-2

2.1 Histórico

A linguagem Modula-2 é descendente de Pascal [JensenWirth 85] e Modula [Wirth 77]. Enquanto Pascal tinha sido desenvolvido para ser de propósito geral, adequado para o ensino de programação e com implementações confiáveis e eficientes, Modula resultou de experiências com multiprogramação. Acabou por introduzir, também, o importante conceito de módulos.

Em 1977, o Instituto de Informática da ETH de Zurique desenvolveu um projeto que tinha por objetivo a criação de um sistema de computação integrado (hardware e software). Esse sistema, chamado posteriormente de *Lilith*, deveria ser programado em uma linguagem de alto nível que satisfizesse tanto as necessidades de um sistema em alto nível quanto a programação em baixo nível daquelas partes que interagissem com o hardware dado. Modula-2 surgiu dessas deliberações como uma linguagem que incluía todos os aspectos do Pascal e algumas extensões, como o conceito de módulos e facilidades para multiprogramação.[1] Sua sintaxe, no entanto, é mais próxima de Modula.

Uma primeira implementação de Modula-2 tornou-se operacional num computador PDP-11 em 1979 e a definição da linguagem foi publicada como Relatório Técnico em março de 1980. Após vários testes em aplicações específicas, o compilador foi modificado para usuários externos ao referido Instituto em março de 1981. O interesse nesse compilador cresceu muito pois passou a incorporar uma ferramenta poderosa para o desenvolvimento de sistemas, implementado em um grande número de microcomputadores.

[1] Cf. [Wirth 85], prefácio.

A evolução de Modula-2 pode ser sintetizada pela figura 2.1.

2.2 Características

As principais características de Modula-2, consideradas em relação ao Pascal, são apresentadas a seguir:

- Introduz o conceito de *módulo* com a facilidade de divisão de um módulo em uma *parte de definição* e uma *parte de implementação*, e a possibilidade de compilação em separado.

- Apresenta uma sintaxe mais sistemática do que o Pascal, o que facilita a aprendizagem. Em particular, toda estrutura que começa com uma palavra-chave também termina com uma palavra-chave. Não há ordem na declaração de constantes, tipos, variáveis e procedimentos; permite recursão indireta sem a necessidade de um identificador especial como o **forward** do Pascal; e, na parte de comandos, introduz os comandos RETURN para terminar a execução de um procedimento, EXIT para forçar o término de uma malha, LOOP como um novo comando repetitivo e elimina o comando GOTO.

- Possui tipos de dados, operações e funções que possibilitam a programação concorrente (através do conceito de *processos*) e a utilização de recursos de baixo nível (por exemplo, manipulação de unidades de armazenamento de baixo nível e endereços, como WORD e ADDRESS, respectivamente).

- Apresenta regras rígidas de consistência entre tipos. Podem ser burladas através das chamadas *funções de transferência de tipos*, da seguinte maneira: os identificadores de tipos podem ser usados como identificadores de funções para transferir o seu argumento num valor correspondente de tipo especificado pelo identificador. Exemplo: se e é do tipo INTEGER, BITSET(e) resulta no valor correspondente de tipo BITSET. As correspondências não são definidas pela linguagem mas o objetivo é não envolver instruções de conversão explícitas mas mudar a interpretação de um determinado objeto.

- Introduz o tipo *procedure*, que permite que procedimentos sejam atribuídos dinamicamente a variáveis e *vetores abertos* como parâmetros de procedimentos.

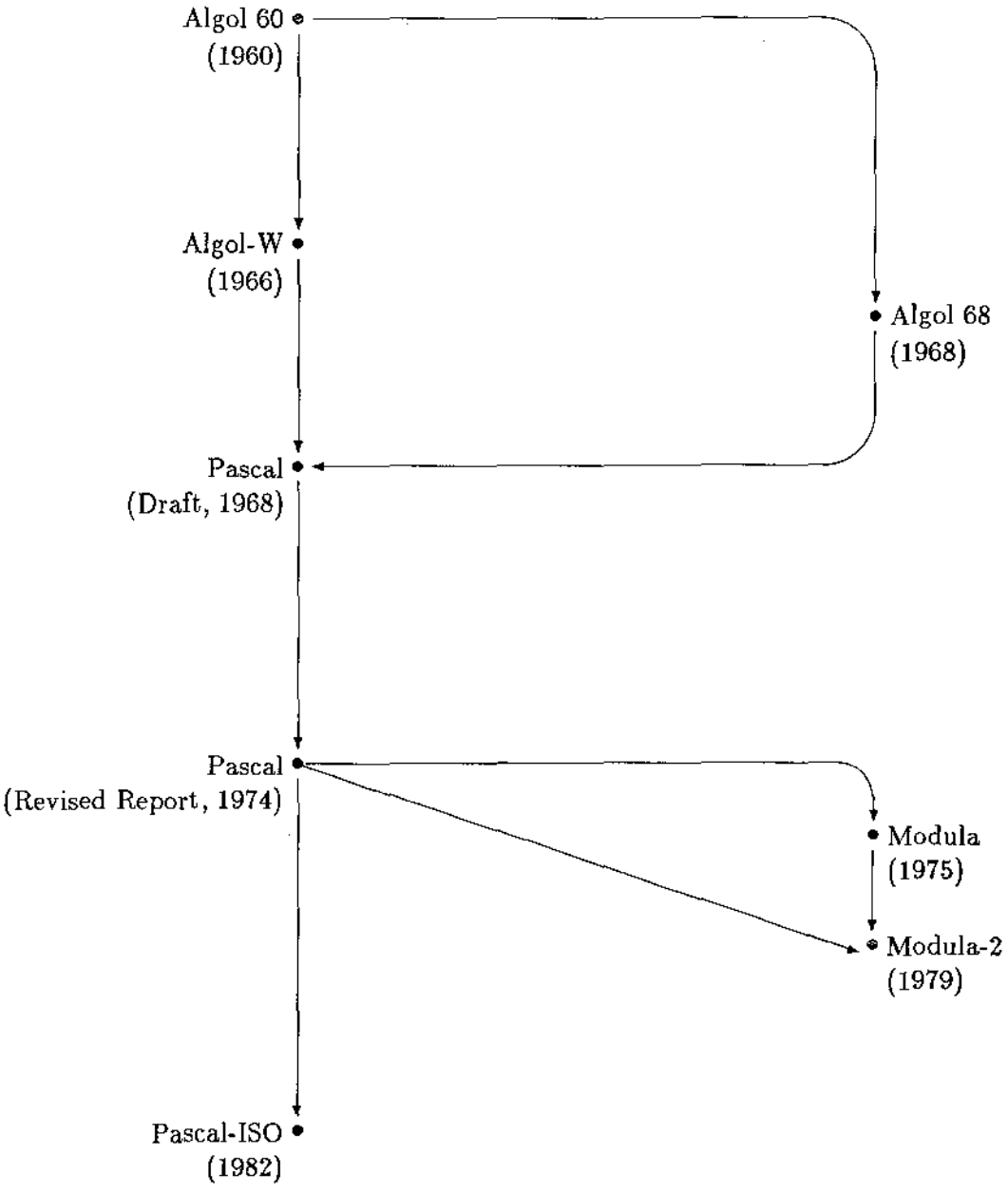

Figura 2.1: Evolução da linguagem Modula-2

- Não inclui a definição de E/S[2] como parte da linguagem.
- Apresenta um módulo especial (pseudo-módulo padrão) SYSTEM com as facilidades para a programação de operações de baixo nível e utilização de processos.[3]
- Proporciona uma facilidade não-padrão na declaração de variáveis que permite a especificação de um endereço absoluto de memória a uma variável. Esta facilidade não faz parte da linguagem, estando disponível em algumas implementações de Modula-2.

2.3 Elogios

A seguir, alguns elogios à linguagem Modula-2 feitos por diversos autores:

- Apresenta bom equilíbrio entre simplicidade e variedade de recursos. Além disso, a simplicidade facilita a padronização e maior portabilidade [Spector 82].
- A modularidade é desejável para a solução de problemas muito grandes com interfaces bem definidas [Spector 82,Coar 84,Pase 85,Greenwood 86].
- A compilação em separado é boa não tanto para esconder detalhes de implementação mas para obter resultados mais imediatos quando do desenvolvimento de sistemas grandes [Christ. et al. 86]. É boa também para manter sua integridade [Jameson 85,Greenwood 86].
- Apresenta um considerável suporte a estruturas de dados, através de registros, tipos aritméticos usuais, caracteres, conjuntos, tipos enumerados, lógicos e apontadores [Pase 85].
- Permite a implementação de tipos abstratos de dados, usando a construção de módulos e tipos opacos [Coar 84,Bielak 85,SegreStanton 85].
- Os procedimentos IOTRANSFER, TRANSFER, NEWPROCESS são excelentes para a criação de um núcleo de sistema numa máquina simples, sem mecanismos de proteção [Christ. et al. 86]. Além disso, a possibilidades de especificar um endereço absoluto a uma variável é uma característica muito

[2] Abreviação de *Entrada/Saída*, que doravante será usada nesse texto.

[3] Pelo fato dos objetos importados do módulo SYSTEM obedecerem a regras especiais, este deve ser conhecido pelo compilador. É, por isso, chamado de *pseudo-módulo* e não precisa estar disponível como um módulo de definição em separado.

desejável [Coar 84]. Modula-2 permite o acesso às facilidades da máquina [Spector 82].

- Apresenta todos os benefícios de C com a vantagem de ter rígidos mecanismos de verificação entre tipos [Jameson 85,Christ. et al. 86,Greenwood 86]. No entanto, esse esquema pode ser burlado em alguns casos [Spector 82,Coar 84].
- É melhor separar a E/S como uma parte fora do resto da linguagem através de facilidades na forma de rotinas de biblioteca [SegreStanton 85].
- Resulta numa linguagem concisa, que é fácil de aprender e usar [Bielak 85,Christ. et al. 86].

2.4 Críticas

As críticas mais relevantes a Modula-2, realizadas por vários autores, são:

- O tamanho efetivo da linguagem é muito grande (comparado com o Pascal) [Moffat 84].
- Os identificadores que são importados, se não forem qualificados (isto é, precedidos pelo nome do módulo onde foram declarados), podem entrar em conflito com identificadores locais ou outros identificadores importados. O texto de Modula-2 [Wirth 85] é omisso nesse caso [Anderson 86]. Problema semelhante ocorre com os conflitos com nomes de campos em múltiplos comandos WITH [Spector 82].
- A importação é restrita a módulos. Seria desejável que procedimentos pudessem importar objetos [Anderson 86]. A prioridade de um módulo é limitada a uma constante e aplica-se a todo o módulo e não a uma pequena região de código ou a um procedimento [Spector 82].
- Módulos locais raramente são usados, trazem problemas para os compiladores e complicações adicionais às regras de visibilidade [Wirth 88].
- As conversões entre tipos são dependentes de implementação e não transportáveis [Coar 84,Christ. et al. 86,Cornelius 88]. A conversão implícita, em alguns casos, não deveria ser permitida [Torbett 87a]. As funções de transferência são uma armadilha sedutora [Wirth 88]!

- Os tamanhos das unidades de armazenamento são restritivos. Poderiam ser criados outros tipos como WORD8, WORD16, WORD32 [Spector 82,Christ. et al. 86]. Além disso, a definição do tipo WORD é ambígua [Torbett 87a]. A aritmética com o tipo ADDRESS não está bem definida [Christ. et al. 86]; operações com apontadores de qualquer tipo resultam em valores completamente dependentes de máquina [Pase 85].

- A manipulação de baixo nível é restritiva em alguns casos (por exemplo, o alinhamento dos componentes de um registro em palavras de máquina) [Spector 82]; a numeração de BITSET não está especificada, afetando a portabilidade [Torbett 87a].

- Não existe um esquema para atribuir valores iniciais a variáveis na ativação de um bloco [Coar 84,Pase 85,Zimmer 85,Christ. et al. 86,Torbett 87a].

- Não é permitida a criação de vetores de tamanho dinâmico [Coar 84]. Os vetores abertos permitidos pela linguagem são limitados a uma dimensão [Moffat 84,Wirth 88].

- Modula-2 não estende a sobrecarga a todos os operadores [Spector 82,Coar 84].

- O comando EXIT deveria permitir um rótulo (para facilitar a saída de malhas encaixadas, por exemplo) [Spector 82,Coar 84]. Além disso, deveria poder ocorrer em qualquer comando repetitivo [Spector 82].

- Não há uma definição precisa dos tipos numéricos que inclui e as operações sobre eles [Spector 82,Moffat 84,SegreStanton 85,Torbett 87a,Cornelius 88].

- As funções retornam apenas valores escalares [Moffat 84].

- Não há cadeias de tamanho variável e procedimentos orientados para cadeias [Spector 82,Moffat 84,Cornelius 88].

- Quando uma variável é exportada, não há maneira de protegê-la quanto à escrita [Spector 82,Zimmer 85,Torbett 87a]. Não há a especificação de que atributos de um tipo exportado deveriam ser visíveis, bem como de variáveis [Cornelius 88].

- Os argumentos de procedimentos são passados de maneira posicional [Spector 82]. A implementação do mecanismo de passagem de parâmetros não está especificada (chamada por referência ou cópia-na-entrada-cópia-

na-saída?), bem como a ordem de avaliação dos parâmetros formais [Torbett 87a].

- As bibliotecas são diferentes em todas as implementações, em particular as que tratam de E/S [Spector 82,Moffat 84,Pase 85,Christ. et al. 86].
- Não há tratamento de exceções [Spector 82,Coar 84,Torbett 87a].
- Há poucas informações sobre processos [Coar 84,Spector 82].

2.5 Comentários

Esta seção traz comentários do autor sobre alguns itens apresentados nas seções precedentes quanto aos elogios e críticas a Modula-2:

- *Modularidade*: Conceito importante introduzido por Modula, tem-se mostrado cada vez mais importante no desenvolvimento de grande sistemas, sob vários aspectos: busca de interfaces bem definidas, facilidade para alterações, desenvolvimento em paralelo, entre outros. Tipos abstratos de dados podem ser bem implementados através do uso de módulos e tipos opacos. Linguagens recentes têm observado essa característica como Ada, Oberon e Modula-3. Módulos locais, no entanto, têm-se mostrado de pouca utilidade, complicando as regras de visibilidade.[4]
- *Tamanho da linguagem*: [Moffat 84] critica o tamanho efetivo de Modula-2 contando o número de palavras reservadas e identificadores predefinidos, comparando-o com Pascal. Ora, *tamanho da linguagem* precisaria ser melhor definido: seria o vocabulário dos identificadores, o tamanho da carta sintática ou, ainda, alguma outra definição? Comparando as cartas de [JensenWirth 85] e [Wirth 85], pode-se afirmar que Modula-2 é "maior" em função das características adicionais que apresenta, como módulos, mas de forma coerente.
- *Tipos*: A variedade de tipos da linguagem é uma característica saudável, servindo de bom suporte para implementação de estruturas de dados. Aparentemente, com o objetivo de simplificar os compiladores, o conceito de vetor aberto não foi estendido a mais de uma dimensão; vetores dinâmicos, como os permitidos em Algol 60, seriam interessantes se incorporados à

[4] Cf. [Wirth 88,Cardelli et al. 88].

linguagem. Existem regras de compatibilidade rígidas mas não completamente coerentes (em alguns contextos, como numa atribuição, podem-se ter expressões do tipo INTEGER ou CARDINAL). Os tipos numéricos não são completamente definidos (o que acontece quando um cardinal é subtraído de outro menor que ele?). As funções de transferência de tipos são dependentes de implementação e é razoável que sejam consideradas dessa maneira.

- *Portabilidade*: Programas que utilizam recursos de baixo nível dificilmente são portáveis entre máquinas. Seria interessante a indicação de que um programa utilizasse esses recursos (note-se que as facilidades de baixo nível não se restringem ao módulo SYSTEM. Por exemplo, considere as funções de transferência de tipos).

- *E/S*: Não é um aspecto tão relevante o fato de fazer ou não parte da linguagem. Modula-2 oferece recursos para essas operações de uma forma muito coerente (um procedimento para cada tipo de parâmetro) mas pouco prática. Uma padronização, nesse sentido, é muito difícil. Poder-se-ia discutir um conjunto mínimo de operações.

- *Importação de identificadores*: O conflito apontado por [Spector 82] está mais na descrição imprecisa da linguagem do que na intenção do projeto. Que procedimentos pudessem importar objetos é possível de ser feito com os recursos já existentes (basta criar um módulo com tais procedimentos).

- *Tamanho das unidades de armazenamento*: A linguagem foi definida em 1977; hoje faz sentido falar em WORD8, WORD16, WORD32 e até WORD64. Mas esses tipos deveriam fazer parte da linguagem? Afinal, são recursos de baixo nível e estreitamente ligados a uma determinada configuração.

- *Valores iniciais de variáveis*: Uma característica interessante e útil mas que aumentaria um pouco mais o tamanho da linguagem.

- *Comando* EXIT: [Spector 82,Coar 84] fazem considerações consistentes a seu respeito. Seria mais coerente que este comando pudesse interromper qualquer comando repetitivo.

- *Cadeias de caracteres*: Não são um aspecto relevante da linguagem; são passíveis de serem manipuladas com os recursos já existentes. Vetores de tamanho variável, como já foi citado, seriam interessantes.

- *Valores de retorno de funções*: Não há motivo aparente para as funções de Modula-2 apresentarem apenas o retorno de valores escalares.

- *Argumentos de procedimentos*: A crítica feita por [Torbett 87] procede, embora a experiência leve à conclusão que a passagem dos argumentos por variável seja feita por referência.

- *Exceções*: Outro recurso desejável, presentes em linguagens mais modernas como Ada e Modula-3 que também aumentaria o tamanho da linguagem.

- *Processos*: A omissão de informações parece proposital, pelo simples fato de processos não fazerem parte da linguagem. Os procedimentos IOTRANSFER, TRANSFER e NEWPROCESS são apenas um exemplo de como utilizar corrotinas para implementar multiprogramação.

Capítulo 3

Linguagem C

3.1 Histórico

C é uma linguagem de propósito geral. Seu rico conjunto de operadores, a ausência de restrições quanto a tipos de dados em expressões e sua generalidade tornam-na mais conveniente e eficiente para muitas tarefas do que outras linguagens, supostamente mais poderosas. Essencialmente, não é dependente de uma determinada configuração de hardware ou de sistema operacional; é fácil escrever programas que sejam *portáveis*, isto é, programas que possam ser executados em qualquer máquina que suporta a linguagem C.

A linguagem foi originalmente projetada para o sistema operacional UNIX e implementada em um PDP-11 da Digital Equipment Corporation (DEC), por Dennis Ritchie [RitThomp 74,RitThomp 78]. O núcleo[1] do UNIX consiste de aproximadamente 10.000 linhas de código em C e 1.000 linhas de código em Linguagem de Montagem (200 linhas por causa da eficiência obtida e 800 que realizam funções de hardware que não são disponíveis em C) [Thompson 78].

A evolução de C pode ser sintetizada pela figura 3.1.

Cabe notar que, nas linguagens BCPL e B, o único tipo de dados embutido é a palavra de máquina, sendo o acesso a qualquer outro tipo de objeto realizado através de funções especiais. C apresenta uma variedade maior de tipos de dados bem como possibilita o acesso às palavras de máquina de maneira eficiente.

[1]Em inglês: *kernel*.

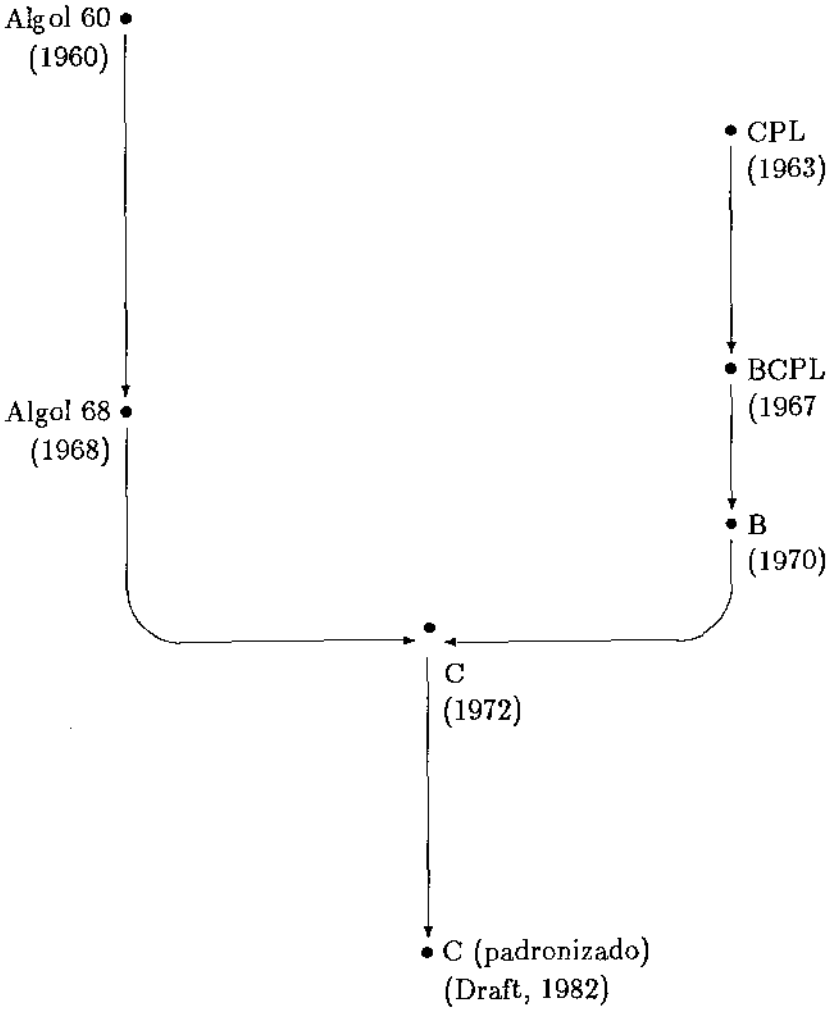

Figura 3.1: Evolução da linguagem C

3.2 Características

As principais características de C, descritas por [KernRit 78], são apresentadas a seguir:

- É uma linguagem de propósito geral, relativamente de baixo nível, pela sua capacidade de manipular elementos de máquina como registradores e endereços. Originalmente, foi projetada para programação de sistemas.
- É relativamente pequena, com compiladores simples e compactos. Com um pouco de cuidado, é fácil escrever programas em C que sejam portáveis, isto é, que podem ser executados em qualquer máquina que suporta C. Admite compilação em separado.
- Apresenta bibliotecas-padrão de rotinas e um pré-processador capaz de expandir macros e realizar outras funções em tempo de compilação.
- Apresenta construções para controle de fluxo de execução requeridas pelos programas bem estruturados e um grande conjunto de operadores.[2] As classes de objetos que manipula são caracteres, números, estruturas e apontadores, com a possibilidade de realizar operações aritméticas com endereços. Não provê operações que lidam diretamente com objetos compostos tais como cadeias de caracteres, conjuntos, estruturas e listas.
- Não é uma linguagem com tipos rígidos como Pascal, Algol 68 ou Modula-2. É relativamente permissiva quanto à conversão de tipos automaticamente. Compiladores existentes não provêem verificações em tempo de execução[3] de índices de vetores, tipos e número de argumentos de uma função, etc. O verificador de tipos, *lint*, é uma ferramenta à parte.
- Permite a utilização de recursos de baixo nível como, por exemplo, a alocação de variáveis em registradores de máquina e aritmética de endereços.
- A definição de E/S não faz parte da linguagem.
- A linguagem não define qualquer facilidade para gerenciamento de memória dinâmica além da definição estática e a disciplina de pilha proporcionada pelas variáveis locais de funções.

[2] C tem algumas características de uma linguagem de expressões ([GhezJaz 87] usa o termo *orientada-para-expressões*, significando que as expressões têm papel mais relevante que os comandos).

[3] Em inglês: *run-time*.

3.3 Elogios

A seguir, alguns elogios feitos à linguagem C por diversos autores:

- É uma linguagem relativamente de baixo nível [KernRit 88] e uma excelente ferramenta para programação de sistemas devido às seguintes características: capacidades de pré-processamento, possibilidade de construção de tipos estruturados a partir de tipos básicos e tipos definidos pelo usuário e grande e poderoso conjunto de operadores [FitzJohn 81,KernRit 88].
- É compacta e eficiente [Stroustrup 86,KernRit 88,Stroustrup 88]. Sua ausência de restrições é uma de suas forças [KernRit 88].
- As possibilidades de aplicações são dependentes das bibliotecas-padrão, não dos compiladores [Hayward 86]. C é flexível, sendo aplicável a várias áreas, não apresentando limitações inerentes que impeçam que trechos de programas possam ser escritos [Stroustrup 86,Stroustrup 88].
- Programas mais portáveis são desejáveis [Hayward 86,KernRit 88]. C é disponível em um grande número de máquinas, desde microcomputadores até super-computadores [Stroustrup 86,Stroustrup 88].
- Vem sendo padronizado desde 1983 pela ANSI com modificações, entre outras, na sintaxe de declarações e definições de funções [KernRit 88]. Apesar de não ser uma linguagem com uma rígida verificação de tipos, tem evoluído para tal.

3.4 Críticas

As críticas mais relevantes a C, levantadas por diversos autores, são:

- A sintaxe da linguagem é irregular e confusa; é difícil escrever formatadores e analisadores sintáticos para C. Existe uma certa criptografia na definição de tipos devido à sua notação sucinta [KernRit 78,Anderson 80]; cf. também [FitzJohn 81]. É notável a grande quantidade de semântica associada para tornar viável a implementação. A sintaxe é incorreta em alguns casos [Anderson 80,Meissner 82]; ocorrem construções que não são aceitas semanticamente [FitzJohn 81]. A descrição da linguagem é vaga [FitzJohn 81].

- Apresenta alguns problemas de portabilidade (por exemplo, a ordem na avaliação dos parâmetros de uma função) [KernRit 88]. Não há uma maneira de especificar a precisão de variáveis inteiras independente de implementação; a numeração de bits em um campo é definida pela implementação [Metz 86].

- C deixa certos detalhes de avaliação de expressões não especificados. Por isso, algumas expressões podem produzir resultados diferentes sob compiladores diferentes [McKee et al. 85].

- A precedência de seus operadores não foi bem escolhida [KernRit 78,Hayward 86,KernRit 88].

- Os tipos de C são pouco estritos; a verificação de tipos, nesse sentido, não é rígida [Anderson 80].

- Os tipos da linguagem são muito restritos [Anderson 80]. Funções não podem devolver tipos estruturados, apenas apontadores para tais tipos [FitzJohn 81].

3.5 Comentários

Esta seção traz comentários do autor sobre alguns itens apresentados nas seções precedentes quanto aos elogios e críticas a C:

- *Capacidade de recursos*: C apresenta um grande e rico conjunto de operadores que permitem desde as operações mais comuns, como a soma de dois inteiros, até a manipulação dos bits da máquina. Se usados com cuidado, programas poderão ser, além de mais rápidos, transportados entre máquinas, sem problemas. Algumas implementações permitem a inserção de código em linguagem de montagem em meio ao código C. Tem-se mostrado apropriada a uma variada gama de áreas de aplicação.

- *Sintaxe*: A notação sucinta em algumas construções pode trazer dificuldades de interpretação, como na declaração

```
char (*(*f())[])();
```

que, numa notação próxima ao Modula-2, deve ser interpretado como

f: **function** ( ): **pointer to array** [ ] **of pointer to function** ( ): **char**;[4]

- *Descrição da linguagem*: Em muitos casos, não está claro que resultados devem ser esperados de uma construção, como em x=x++. [McKee et al. 85] mostra várias construções que resultam em valores diferentes, dependendo da implementação.

- *Portabilidade*: Programas que utilizam recursos de baixo nível dificilmente são portáveis entre máquinas. C diminui consideravelmente essa dependência de implementação para resolução de problemas específicos se a programação for feita de maneira cuidadosa. No entanto, a descrição omissa em alguns pontos e o problema da verificação de tipos compromete a portabilidade.

- *Tipos*: C não é uma linguagem com verificação estrita de tipos. Um programa que realiza essa função, *lint*, é uma ferramenta do sistema UNIX. A padronização de C [DraftC 85] modificou a linguagem de forma que muitos casos onde a mistura de tipos era permitida não mais o será (em particular na passagem de parâmetros de uma função).

- *Funções/Compilação em separado*: Devido à sintaxe original, não era permitida a especificação do número de parâmetros e seus tipos quando da declaração de uma função.[5] Isso impossibilitava a verificação estática da compatibilidade dos parâmetros na chamada e, mesmo, o seu número. Nesse sentido, a compilação em separado pode ser uma fonte de problemas.

- *E/S*: O fato de sua definição não fazer parte da linguagem, está relacionado com o sistema UNIX essencialmente. Tem-se procurado padronizar as rotinas de E/S para diminuir o problema de portabilidade. Bibliotecas-padrão são úteis conquanto que não se proliferem demasiado.

- *Valores de retorno de funções*: O fato de funções não devolverem tipos estruturados está relacionado com a simplicidade de implementação: a devolução de resultados pode ocorrer nos registradores de máquina.

---

[4] f é uma função que devolve um apontador para um vetor de apontadores para funções que retornam valores do tipo **char**. Cf. [Anderson 80].

[5] [KernRit 78] considera *declaração de função* como a descrição do tipo de resultado devolvido por ela e *definição de função* como a descrição completa da mesma: tipo do resultado, número, nomes e tipos dos parâmetros e seu bloco.

Capítulo 4

Linguagem MC

4.1 Objetivos

A linguagem MC tem como objetivo principal unir as facilidades de baixo nível apresentadas pela linguagem C à estrutura e disciplina de Modula-2, resultando numa linguagem versátil, de propósito geral e adequada à programação de sistemas. O acesso às facilidades de máquina tornam os programas menos portáveis mas diminuem a dependência do uso das linguagens de montagem das mesmas.

A definição de MC procura ser mais precisa do que C e Modula-2, deixando as dependências de implementação relegadas a aspectos não fundamentais (por exemplo, o número de caracteres significativos de um identificador). As dependências de configuração estão centradas em dois módulos, SYSTEM e LOW_LEVEL. Todo programa que utiliza recursos destes módulos estará se comprometendo com uma dada implementação mas de maneira explícita.

MC não é uma extensão de Modula-2 simplesmente. Incorpora também outros mecanismos encontrados em outras linguagens como o tratamento de exceções [DraftAda 83,Smedema 84], a introdução do conceito de vetores com limites abertos[1] e novos conceitos como a descrição sintática de subprogramas com um número variável de parâmetros, entre outros.

Outra contribuição importante é a definição mais precisa do problema da compatibilidade entre tipos, situando-a em quatro contextos: compatibilidade para operação, compatibilidade para atribuição, compatibilidade para passagem de parâmetros e compatibilidade para devolução de resultados de funções.

Cabe ressaltar que a preocupação principal de MC não é a inovação de conceitos mas a consolidação de soluções interessantes propostas em várias linguagens,

[1] Uma extensão do padrão ISO para o Pascal. Cf. [ISOPascal 82,Cooper 83].

na linha do que afirma C. Hoare como um dos objetivos do projeto de linguagens de programação.[2]

4.2 Características

As principais características de MC são apresentadas a seguir. A comparação de seus aspectos com suas linguagens predecessoras, Modula-2 e C, será feita nas seções subseqüentes.

- É uma linguagem de propósito geral, adequada à programação de sistemas. Apresenta construções para controle de fluxo de execução requeridas por programas estruturados e um conjunto de operadores e funções equivalentes às da linguagem C. Sua sintaxe é simples, próxima à de Modula-2.
- Permite compilação em separado através do uso de módulos, da mesma maneira que Modula-2.
- É uma linguagem de expressões: todo comando de MC devolve um resultado, que poderá até mesmo ser de valor e tipo indefinidos, podendo ou não ser utilizado, dependendo do contexto.
- Possibilita o acesso às facilidades de baixo nível de maneira ordenada e segura (por exemplo, associação de atributos a variáveis na sua declaração).
- As dependências de configuração ficam restritas à utilização de módulos especiais (SYSTEM, LOW_LEVEL).
- Apresenta regras rígidas de verificação de tipos; podem ser usadas funções de transferência de tipos, dependentes de implementação, para burlá-las de maneira explícita.
- Operações com apontadores e endereços são feitas através de funções definidas nos módulos dependentes de configuração.
- Processos são um mecanismo definido na linguagem que possibilita a programação concorrente. A comunicação entre processos deve utilizar as facilidades do módulo especial LOW_LEVEL.
- Introduz vetores com limites abertos em dois contextos: como parâmetros de um subprograma (semelhante ao conceito de vetores conformes do Pascal, podendo ser utilizados também como vetores de parâmetros em número

[2] Cf. [GhezJaz 87], pg. 333.

variável) e como variáveis dinâmicas, com a definição de seus limites (esse esquema é próximo ao Algol 60 que permite, na entrada de um bloco, a definição de vetores cujos tamanhos não são conhecidos em tempo de compilação mas são dados por variáveis definidas em algum bloco que contém este com a definição de tais vetores). Funções podem retornar qualquer valor, simples ou estruturado.

- Tipos de dados oferecidos pela linguagem: simples (inteiros e reais, enumerados e subintervalos), estruturados (vetores com limites definidos, vetores com limites abertos, registros, registros com variantes e conjuntos), apontadores e identificadores de processos.

- Subprogramas podem ter um número variável de parâmetros, descritos sintaticamente com a utilização de vetores com limites abertos.

- O excaixamento estático é restrito: procedimentos e processos não podem ser declarados em procedimentos; processos não podem ser declarados em processos.

- Provê um mecanismo para atribuição de valores iniciais a variáveis na ativação de um bloco.

- Possibilita o tratamento de exceções.

- Não inclui a definição de E/S como parte da linguagem, tampouco o tipo cadeia de caracteres mas pode-se utilizar uma biblioteca como em C e Modula-2.

4.3 Aspectos Comuns a MC, Modula-2 e C

Os principais aspectos comuns a MC, Modula-2 e C serão discutidos a seguir, com a consideração das diferentes soluções existentes em cada uma dessas linguagens.

- *Acesso às Facilidades de Baixo Nível:* MC permite a utilização dos recursos de baixo nível, dependentes de implementação, através da importação de identificadores especiais e funções do módulo LOW_LEVEL. Em particular, as funções de transferência de tipos não são definidas em módulo algum mas podem ser utilizadas mediante importação do identificador *TypeTransferFunctions*, do módulo LOW_LEVEL. Modula-2 não exige uma indicação especial para o uso destas funções; outras facilidades de baixo nível estão espalhadas em vários módulos como *SYSTEM* (tipo ADDRESS e comunicação

MC	Modula-2	C	Interpretação
BOOLEAN	BOOLEAN	—	valores lógicos FALSE e TRUE
—	CHAR	—	caracteres; código ASCII 0..127
CHAR	—	/* **unsigned** */ **char**	caracteres, código ASCII 0..255
SHORTCARD	—	**unsigned short** /* **int** */	cardinais curtos
SHORTINT	—	**short int**	inteiros curtos
CARDINAL	CARDINAL	**unsigned** /* **int** */	cardinais
INTEGER	INTEGER	**int**	inteiros
—	—	**unsigned long** /* **int** */	cardinais longos
—	LONGINT	**long int**	inteiros longos
REAL	REAL	**float**	reais
—	LONGREAL	**double, long float**	reais longos

Tabela 4.1: Tipos simples predefinidos das linguagens MC, Modula-2 e C

entre processos, por exemplo) e *Storage* (rotinas para gerenciamento de memória dinâmica). A especificação de endereços a variáveis é uma facilidade não padrão da linguagem, disponível em algumas implementações. C não exige o uso de funções de transferência de tipos mas, se forem utilizadas, não é necessária qualquer indicação especial para a utilização deste recurso. Alguns mecanismos de baixo nível são previstos na própria linguagem (por exemplo, a alocação de uma variável em um registrador de máquina) e outros são dependentes de funções de bibliotecas.

- *Tipos de Dados:* As três linguagens manipulam tipos simples e tipos estruturados. A tabela 4.1 apresenta os tipos simples predefinidos equivalentes das três linguagens com interpretações naturais para os seus tamanhos em microcomputadores comuns de 16 bits. Note-se que C admite qualificadores para os tipos básicos inteiros **char** e **int**, que se referem aos seus tamanhos e à aritmética com ou sem sinal. Modula-2 e MC apresentam, ainda, tipos simples enumerados e subintervalos. Os tipos estruturados são apresentados na tabela 4.2. Registros são semelhantes nas três linguagens, que também permitem o uso de apontadores. Dentre elas, C é a que impõe menos restrições às operações com apontadores.

Tipo Estruturado	MC	Modula-2	C
vetor (com limites definidos)	índice de qualquer tipo simples, menos REAL; limites especificados na declaração	índice de qualquer tipo simples, menos REAL; limites especificados na declaração	índice de tipo **int**; limite inferior: 0; limite superior: $n-1$, onde n (tamanho do vetor) é especificado na declaração
vetor com limites abertos	índices de qualquer tipo simples, limites especificados na criação (variável dinâmica) ou na passagem do parâmetro efetivo	índice de tipo CARDINAL; limite inferior: 0; limite superior: especificado pelo parâmetro efetivo; restrito a uma dimensão	não há; a aritmética com apontadores pode ser usada para simular esse tipo em conjunto com a característica de não verificação de estouro nos limites de um vetor
registro	componentes de qualquer tipo menos vetor com limites abertos	componentes de qualquer tipo	componentes de qualquer tipo
registro com variante	campo de marca pode ou não fazer parte do registro	campo de marca pode ou não fazer parte do registro	outra sintaxe (**union**); não existe o conceito de campo de marca mas este pode ser simulado
apontador	apontadores para tipos diferentes não são equivalentes; equivalente ao tipo ADDRESS (módulo LOW_LEVEL)	apontadores para tipos diferentes não são compatíveis; compatível com o tipo ADDRESS (módulo SYSTEM)	não verifica compatibilidade (C padrão)

Tabela 4.2: Tipos estruturados das linguagens MC, Modula-2 e C

- *E/S:* A definição de E/S não pertence às linguagens nos três casos. Funções de biblioteca em C e módulos de subprogramas em Modula-2 e MC podem ser definidos para realizarem essa tarefa.

- *Compilação em Separado:* MC e Modula-2 apresentam o mesmo conceito de módulos, possibilitando a compilação em separado (mas não independente). C possibilita compilação em separado de arquivos com definições de funções, variáveis, tipos e macros.

- *Comandos:* MC é uma linguagem de expressões, no sentido que cada construção executável da linguagem devolve um resultado, mesmo que seja de valor e tipo indefinidos, que poderá ou não ser utilizado, dependendo do contexto. Modula-2 não é uma linguagem de expressões. C é praticamente uma linguagem de expressões, apresentando estruturas de controle que a aproxima de linguagens como Pascal e Modula-2 e que não retornam resultados. Uma comparação entre os comandos das linguagens é feita a seguir:

  - Comando de Atribuição:
    * MC: Não permite conversões implícitas; permite atribuição múltipla; devolve um resultado.
    * Modula-2: Permite conversão implícita em alguns casos; não devolve resultado.
    * C: Operador; permite atribuição múltipla; devolve resultado.
  - Comando Condicional **if**
    * MC: **if–then–elsif–else**. Comando *if* encadeado sintaticamente através de cláusulas *elsif*; executa a seqüência de expressões cuja expressão associada resulte no valor lógico TRUE; seu resultado é o da seqüência de expressões executada, dependendo das várias alternativas existentes no comando.[3]
    * Modula-2: IF-THEN-ELSIF-ELSE. Comando *if* encadeado sintaticamente através de cláusulas *elsif*; executa a seqüência de comandos cuja expressão associada resulte no valor lógico TRUE.
    * C: **if–else**. Executa o comando que antecede o símbolo **else** se a expressão resultar num valor não nulo; a ambigüidade usual com o *else* é resolvida associando-o com o último símbolo **if** não

[3]Em MC, o resultado de uma seqüência de expressões é o resultado da última expressão da seqüência.

associado a símbolo **else**. O comando condicional aritmético é implementado pelo operador ternário "? :".

- Comando Condicional **case**
  - MC: **case–others**. Seleção e execução de uma seqüência de expressões; termina quando se atinge novo rótulo de caso, que é uma constante, e seu resultado é o da seqüência de expressões executada, dependendo das várias alternativas existentes no comando.
  - Modula-2: CASE-ELSE. Seleção e execução de uma seqüência de comandos; termina quando se atinge novo rótulo de caso, que é uma constante.
  - C: **switch–case–default**. Seleção e desvio para um dentre vários rótulos de comandos; termina quando se executa um comando **break**; o rótulo de caso é uma expressão constante de tipo inteiro.
- Comando Repetitivo **while**
  - MC: **while–do–end**. Execução de uma seqüência de expressões zero ou mais vezes; seu resultado é de tipo e valor indefinidos.
  - Modula-2: WHILE-DO-END. Execução de uma seqüência de comandos zero ou mais vezes.
  - C: **while**. Execução de um comando zero ou mais vezes.
- Comando Repetitivo **repeat–until**
  - MC: **repeat–until**. Execução de uma seqüência de expressões ao menos uma vez; seu resultado é de tipo e valor indefinidos.
  - Modula-2: REPEAT-UNTIL. Execução de uma seqüência de comandos ao menos uma vez.
  - C: **do–while**. Execução de um comando ao menos uma vez.
- Comando Repetitivo **loop**
  - MC: **loop–end**. Execução repetida de uma seqüência de expressões; seu resultado é de tipo e valor indefinidos.
  - Modula-2: LOOP-END. Execução repetida de uma seqüência de comandos.
  - C: **for** (;;). Execução repetida de um comando.
- Comando Repetitivo **for**
  - MC: **for–to–by–do–end**. Execução de uma seqüência de expressões enquanto uma progressão de valores é atribuída à variável de controle; seu resultado é de tipo e valor indefinidos.

  * Modula-2: FOR-TO-BY-DO-END. Execução de uma seqüência de comandos enquanto uma progressão de valores é atribuída à variável de controle.
  * C: **for** (e_1;e_2;e_3). Execução repetida de um comando. Termina quando e_2 resultar em zero.

- Comando de Desvio **continue**
  * MC: **continue**. Desvio para a avaliação da expressão de controle do comando repetitivo mais interno ou precedido pelo rótulo especificado; seu resultado é de tipo e valor indefinidos.
  * Modula-2: —
  * C: **continue**. Desvio para a avaliação da expressão de controle do comando repetitivo mais interno.

- Comando de Desvio **exit**
  * MC: **exit**. Término do comando repetitivo mais interno ou precedido pelo rótulo especificado; seu resultado é de tipo e valor indefinidos.
  * Modula-2: EXIT. Término do comando repetitivo mais interno.
  * C: **break**. Término dos comandos **while**, **do-while**, **for** ou **switch** mais internos.

- Comando de Desvio **return**
  * MC: **return**. Término da execução de um procedimento, processo (nesses casos, seu tipo e valor são indefinidos) ou função (com a devolução do resultado da avaliação da expressão associada, que é o resultado do comando).
  * Modula-2: RETURN. Término da execução de um procedimento ou corpo de um módulo e devolução do resultado da expressão associada, em se tratando de uma função.
  * C: **return**. Término da execução de uma função com a devolução de um resultado definido (se houver uma expressão associada) ou não.

- Comando de Desvio **goto**
  * MC: —
  * Modula-2: —
  * C: **goto**. Desvio para o comando precedido pelo rótulo especificado.

- Comando de Escopo de Registro
  - MC: **with-end**. Possibilita a omissão da qualificação de identificadores de campos de um registro na seqüência de expressões associada; seu resultado é da seqüência de expressões executada.
  - Modula-2: WITH-END. Possibilita a omissão da qualificação de identificadores de campos de um registro na seqüência de comandos associada.
  - C: —
- Comando de Ativação de Processo
  - MC: **start**. Inicia a execução de um processo com a especificação de atributos; seu resultado é a identificação do processo criado.
  - Modula-2: Procedimentos NEWPROCESS, TRANSFER e IOTRANSFER do módulo SYSTEM.
  - C: Facilidade não-padrão, dependente de bibliotecas de funções e sistema operacional hospedeiro.
- Comando de Ativação de Exceção
  - MC: **raise**. Levanta a exceção especificada; seu resultado é de tipo e valor indefinidos.
  - Modula-2: —
  - C: —
- Chamada de Subprogramas
  - MC: Todo procedimento retorna um resultado de tipo e valor indefinidos; uma função retorna um resultado cujo tipo é especificado no seu cabeçalho.
  - Modula-2: Procedimentos e funções são distintos, embora introduzidos pela mesma palavra reservada PROCEDURE; o que os diferencia é que uma função é um procedimento que retorna um valor (tipo especificado no seu cabeçalho).
  - C: Funções que não retornam valor se comportam como procedimentos.
- Comando Composto
  - MC: Não existe uma construção particular; uma seqüência de expressões é o seu equivalente.
  - Modula-2: Não existe uma construção particular; uma seqüência de comandos é o seu equivalente.

    * C: { $c_1 \ldots c_n$ }. Permite a declaração de variáveis locais ao bloco.
  - Comando Vazio
    * MC: Relaxamento sintático das regras de pontuação em seqüências de expressões; seu resultado é de tipo e valor indefinidos.
    * Modula-2: Relaxamento sintático das regras de pontuação em seqüências de comandos.
    * C: Útil para permitir a declaração de um rótulo imediatamente antes do final de um comando composto ou para ser o corpo vazio de um comando repetitivo.

- *Funções e Procedimentos:* permite-se a recursão nas três linguagens. Funções devolvem um resultado através de comandos de retorno de valor ("**return**").

- *Operadores:* Os operadores das três linguagens são comparados nas tabelas apresentadas no apêndice A.

4.4 Aspectos Comuns a MC e Modula-2

Os aspectos comuns mais relevantes às linguagens MC e Modula-2 são apresentados nesta seção, com considerações sobre suas particularidades.

- *Módulos:* O conceito de módulos é o mesmo em ambas as linguagens. Um programa é constituído por um módulo principal (chamado módulo de programa), que pode ser ligado a outros módulos, que são divididos em módulos de definição e módulos de implementação. Podem ser declarados módulos locais. Módulos especiais são definidos como parte integrante da linguagem, permitindo a utilização de recursos de baixo nível (por exemplo, módulo SYSTEM) ou constituindo bibliotecas de rotinas auxiliares (por exemplo, *MathLib0* em Modula-2 e MATH_LIB em MC, com as funções aritméticas).

- *Tipos Opacos:* Um tipo opaco ou privado esconde detalhes do tipo de estrutura de dados bem como os detalhes de suas operações. São usados em módulos de definição e implementados utilizando-se apontadores.

- *Tipos Estruturados:* Além de apresentarem os tipos estruturados vetor (com limites definidos) e registro (fixo e com variantes), apresentam o tipo conjunto e os operadores correspondentes.

- *Tipo* ADDRESS: Em Modula-2, esse tipo deve ser importado do módulo SYSTEM e é compatível com todos os tipos apontadores e com o tipo CARDINAL, permitindo a aritmética de apontadores como cardinais. Em MC, o tipo ADDRESS deve ser importado do módulo LOW_LEVEL, é equivalente a qualquer tipo apontador e as operações com objetos desse tipo (soma, por exemplo) devem utilizar as funções definidas no módulo LOW_LEVEL.

- *Processos:* Modula-2 oferece algumas facilidades básicas que permitem a especificação de processos quase-concorrentes e de concorrência genuína para dispositivos periféricos. O termo processo é utilizado com o significado de corrotina. Os tipos de dados e procedimentos para manipulação das corrotinas devem ser importados do módulo SYSTEM. Em MC, processos são parte integrante da linguagem e são executados concorrentemente. Variáveis do tipo identificador de processo (PROCESSID) podem conter a identificação de um processo, resultado, por exemplo, de um comando de ativação de processo (*start*). As rotinas que implementam mecanismos de comunicação entre processos devem importar as facilidades definidas no módulo LOW_LEVEL.

- *Tipos Rígidos:* Ambas as linguagens apresentam um esquema rígido de verificação de tipos. Esse mecanismo pode ser burlado através da utilização de funções de transferência de tipos e funções de compatibilização de tipos (em MC, deve ser importado um identificador do módulo LOW_LEVEL no caso das funções de compatibilização de tipos, pois são consideradas facilidades de baixo nível). MC considera a compatibilidade entre tipos em quatro contextos: operação, atribuição, passagem de parâmetros e devolução de resultado de função.

- *Cadeias de Caracteres:* Ambas não prevêem cadeias como parte integrante das mesmas exceto como vetores de caracteres constantes.

- *Mecanismo de Passagem de Parâmetros:* A passagem de parâmetros é a mesma em ambas as linguagens: por valor e por variável.

- *Especificação de Atributos:* Em MC, qualquer declaração de objeto permite a especificação de atributos, que dependem da implementação (por exemplo, o endereço onde o objeto deve ser alocado, se aplicável). Em Modula-2, a especificação de endereços é uma facilidade não-padrão, podendo ocorrer em algumas implementações.

4.5 Aspectos Comuns a MC e C

Os aspectos comuns mais relevantes às linguagens MC e C são apresentados a seguir, com considerações sobre suas particularidades:

- *Subprogramas com Número Variável de Parâmetros:* C não verifica se o número de parâmetros efetivos é igual ao número de parâmetros formais de uma função; a conseqüência deste fato é que uma função em C se comporta como se tivesse um número variável de parâmetros (no entanto, o acesso a um parâmetro formal que não tenha um parâmetro efetivo correspondente não causa erro de execução). MC dá a possibilidade de se escrever subprogramas que tenham um número variável de parâmetros, com uma descrição sintática particular.

- *Valores Iniciais em Declarações de Variáveis:* Na declaração de variáveis podem ser-lhes atribuídos valores iniciais. Em MC isso pode ser feito para qualquer tipo de variável; C não permite essa atribuição para vetores cuja classe de armazenamento não seja estática.

- *Encaixamento Sintático:* Funções (subprogramas) não podem ser encaixados.

- *Classe de Armazenamento:* Em C, variáveis podem ser estáticas, automáticas, externas ou alocadas em registradores. Variáveis estáticas em MC, somente as globais; automáticas são as variáveis locais; as externas, as que são importadas; a alocação em registradores é feita através do uso de atributos na sua declaração.[4]

4.6 Aspectos Particulares de MC

MC apresenta conceitos que não existem em Modula-2 e C mas estão presentes em outras linguagens como Algol 60 e Pascal-ISO (vetor com limite aberto), Ada e Chill (tratamento de exceções).

- *Tipo Vetor com Limites Abertos:* É uma extensão do conceito de *vetores semelhantes* ou *conformes*[5] do Pascal[6] e permite que vetores tenham seu

[4]Pode-se conseguir a alocação estática ou automática para outras variáveis através da especificação de atributos.

[5]Em inglês: *conformant arrays.*

[6]Cf. [ISOPascal 82,Cooper 83].

tamanho definido em tempo de execução. Somente variáveis dinâmicas e parâmetros de subprogramas e processos podem ser desse tipo. Vetores abertos de Modula-2 são um caso particular desse tipo vetor.

- *Tratamento de Exceções:* É possível, em MC, o levantamento de uma exceção, indicando um erro ou alguma condição excepcional na execução de um bloco e seu tratamento, através de *tratadores de exceção*,[7] ou sua propagação para o ponto de chamada.

4.7 Aspectos de Modula-2 ou C Inexistentes em MC

Algumas características presentes em Modula-2 ou C mas inexistentes em MC são apresentadas a seguir:

- *Declarações Locais em Comandos Compostos:* Um comando composto em C pode trazer a declaração local de objetos (esse comando é chamado de *bloco*). Seu escopo se estende até o final deste bloco.
- *Pré-Processador:* MC não apresenta um pré-processador ou características capazes de realizar as mesmas operações que o pré-processador de C mas poderia ser definido como um apêndice da linguagem.
- *Tipos Apontadores para Funções:* C permite a manipulação de apontadores para funções; Modula-2 apresenta o tipo procedimento. Uma variável ou parâmetro desse tipo pode receber o endereço de uma função ou procedimento.
- *Aritmética com Apontadores:* C permite a aritmética com apontadores, que é muito interessante quando se manipulam vetores em geral. Poder-se-ia obter efeito semelhante em MC utilizando-se, convenientemente, as funções predefinidas do módulo SYSTEM: INC, DEC, SIZE, TSIZE e funções de transferência de tipos.[8]

[7] Em inglês: *exception handlers.*

[8] Cf. uma possível extensão de MC em 7.3.2.

Capítulo 5

Discussão de Alguns Aspectos de MC

Este capítulo apresenta algumas características da linguagem MC contrapostas às soluções existentes em várias linguagens, procurando justificar a solução de projeto adotada, com o intuito de manter a simplicidade e funcionalidade da linguagem.

Os aspectos apresentados são:

- Exceções
- Vetores com limites abertos
- Passagem de parâmetros
- Número variável de parâmetros em subprogramas
- Processos
- Compatibilidade entre tipos

5.1 Exceções

Os eventos ou condições com que uma unidade executável se depara durante sua execução podem ser classificados como normais ou excepcionais. Exemplos destas últimas incluem divisão ilegal por zero, final de arquivo inesperado ou dado de entrada inválido, erro de protocolo durante a recepção de uma mensagem por uma linha de comunicação, entre outros. No entanto, o que é considerado um

estado normal de processamento é uma decisão do programador e é dependente da natureza da aplicação. Assim, um estado anômalo ou *exceção* não significa necessariamente a ocorrência de uma situação catastrófica mas, sim, que a unidade que está sendo executada é incapaz de continuar de maneira que seu término seja normal. Esta incapacidade de um programa lidar com situações especiais pode ser resolvida através de trechos de programa que são executados sob certas circunstâncias.

Linguagens de programação tradicionais (exceto PL/I) não oferecem mecanismos especiais para o tratamento adequado de situações excepcionais. Algumas linguagens mais recentes apresentam características que permitem, sistematicamente, o tratamento de tais condições, fazendo com que a preocupação com anomalias possa ser concentrada em um bloco e deslocada para fora do fluxo principal de execução de maneira a simplificar o algoritmo implementado.

Segundo [GhezJaz 87,Horowitz 84], as questões centrais resolvidas pelos esquemas de tratamento de exceções são:

1. Como uma exceção é declarada e qual o seu escopo?
2. Como uma exceção é levantada (ou sinalizada)?
3. Como especificar que unidades devem ser executadas quando exceções são levantadas (tratadores de exceções)?
4. Como segue o fluxo de execução após o tratamento de uma exceção?
5. Que exceções são possíveis? As que são levantadas pelo sistema ou definidas por usuários?
6. Exceções podem ser propagadas?
7. Exceções podem ser levantadas em um tratador? Quais são as conseqüências?

5.1.1 Tratamento de Exceções em CLU

CLU[1] foi projetada com o objetivo de dar suporte a uma metodologia de programação baseada no reconhecimento de abstrações. Além dos tipos predefinidos da linguagem, pode-se definir tipos abstratos de dados cujo acesso é feito via apontadores. Provê estruturas convencionais de controle a nível de comandos e

[1]Cf. [Liskov et al. 79].

permite a definição de novas estruturas de controle de repetição via iteradores. Além disso, provê facilidades para tratamento de exceções.

Em CLU, exceções são levantadas apenas por procedimentos e são normalmente tratadas na unidade invocadora, isto é, se um comando levanta uma exceção, a ativação corrente do procedimento termina e a invocação correspondente (na unidade invocadora) *levanta*, isto é, sinaliza a mesma exceção. As exceções que um procedimento pode levantar são declaradas no seu cabeçalho. Uma exceção é levantada explicitamente por meio de um comando *signal*. Operações pertencentes à linguagem podem levantar um conjunto de exceções conhecidas. Quando uma invocação levanta uma exceção, o controle é imediatamente transferido para seu tratador aplicável, se existir. Tratadores são associados a comandos; quando a execução de um tratador termina, o controle é passado para o comando seguinte àquele que o continha.

A forma geral de um comando com um tratador de exceções associado é:

comando **except** *lista_de_tratadores* [*tratador_de_outras_exceções*] **end**

A forma de uma lista de tratadores de exceções é:

when *lista_de_identificadores_de_exceção* [(*lista_de_parâmetros*)] : *comando*

e o tratador de outras exceções:

others [(*identificador* : **string**)] : *comando*

Parâmetros (em *lista_de_parâmetros* e *identificador* : **string**) são usados para a passagem de informações sobre a exceção para o tratador. A cláusula *others* trata qualquer exceção não presente em qualquer *lista_de_tratadores* do tratador.

Se a chamada de um procedimento levantar uma exceção mas não houver um tratador associado, esta é propagada, progressivamente, para os escopos estáticos maiores, dentro do procedimento. Se nenhum tratador for encontrado no procedimento, a exceção predefinida *failure* é levantada e o controle retorna à unidade invocadora deste procedimento. *failure* é a única exceção que é implicitamente propagada e não precisa ser listada nos cabeçalhos dos procedimentos.

O comando

signal *identificador_de_exceção* [(*lista_de_expressões*)]

levanta a exceção, termina a execução do procedimento e transfere o controle para a unidade invocadora. A lista de expressões do comando são os parâmetros efetivos para o tratador.

Uma exceção pode ser tratada localmente no procedimento que a levanta. Para indicar esse fato, a exceção é levantada por um comando diferente:

exit *identificador_de_exceção* [(*lista_de_expressões*)]

Este comando é similar a um comando de desvio incondicional, transferindo o controle para o tratador especificado.[2]

CLU não provê mecanismos que desabilitam exceções.

Exemplo:[3] No exemplo a seguir, a proposta do procedimento *get_number* é obter um inteiro de um arquivo de entrada. Para tanto, utiliza os procedimentos *get_field*, que devolve uma cadeia de caracteres sem delimitadores em branco do arquivo de entrada, e *s2i*, que converte uma cadeia de caracteres no valor inteiro correspondente. *get_field* e *s2i* podem levantar exceções que *get_number* pode tanto tratar quanto propagar para sua unidade invocadora.

```
get_number = proc (s: stream) returns (int)
                    signals (end_of_file,
                             unrepresentable_integer (string),
                             bad_format (string))
        field: string := get_field (s)
         except when end_of_file: signal end_of_file
         end
        return (s2i (field))
          except
                when unrepresentable_integer:
                   signal unrepresentable_integer (field)
                when bad_format, invalid_character (*):
                   signal bad_format (field)
          end % exception clause
        end get_number
```

Os cabeçalhos de definição de *get_field* e *s2i* são:

```
get_field = proc (s: stream) returns (string)
                          signals (end_of_file)

s2i = proc (s: string) returns (int)
                    signals (invalid_character (char)
                             bad_format,
                             unrepresentable_integer)
```

[2] CLU não apresenta o comando **goto**.

[3] Cf. [Liskov et al. 79].

O tratador em *get_number* para as exceções *bad_format* e *invalid_character*, que podem ser levantadas em *s2i*, usa a forma parâmetro * para indicar que nenhum dos parâmetros é usado no tratador, pois não são importantes no contexto; ambas são propagadas para a unidade invocadora como uma única exceção, *bad_format*.

5.1.2 Tratamento de Exceções em Ada

Ada[4] é uma linguagem de programação que foi desenvolvida para dar suporte, principalmente, às aplicações em tempo real, com facilidades para modelar processamento paralelo; às aplicações numéricas e à programação de sistemas. Apresenta tipos de dados predefinidos e construtores de tipos. Estruturas de controle são semelhantes às do Pascal. Possibilita o tratamento de exceções e ativações concorrentes. Apresenta também verificação rígida de tipos.

O nome de uma exceção em Ada deve ser declarado da seguinte forma, a não ser que seja o nome de uma exceção predefinida:

lista_de_identificadores : **exception**;

Uma unidade de programa pode levantar explicitamente uma exceção através do comando *raise*:

raise *identificador_de_exceção*;

Tratadores de exceção podem aparecer somente no final do corpo de um subprograma, do corpo de um módulo ou de um bloco, após a palavra reservada **exception**. Por exemplo:

begin
-- *sequência_de_comandos*
exception
    when *escolha_de_exceção* { | *escolha_de_exceção* } =>
                seqüência_de_comandos
end;

onde *escolha_de_exceção* tem a forma:

identificador_de_exceção | **others**[5]

[4] Cf. [DraftAda 83,Barnes 80].

[5] "|", neste caso, significa escolha alternativa.

identificador_de_exceção indica a que exceção o tratador se refere; a seqüência de comandos é o corpo do tratador. others se refere a todas as exceções que não foram mencionadas em qualquer *escolha_de_exceção* do bloco; é uma cláusula opcional.

Quando uma exceção é levantada em uma unidade de programa, a execução normal é suspensa e um dos seguintes eventos ocorre:

1. Se um bloco não contém um tratador para a exceção, sua execução é terminada e a mesma exceção é relevantada na seqüência de comandos que o contém. De maneira semelhante, se um subprograma não contém um tratador local, sua execução é terminada e a mesma exceção é relevantada no ponto de sua invocação. Em ambos os casos, diz-se que a exceção é *propagada*.

2. Se uma *tarefa*[6] não contém um tratador local para a exceção, ela é terminada mas a exceção não é propagada.[7]

3. Se um tratador local para a exceção é definido, sua execução substitui a do restante da unidade atual.

Exemplo: Para analisar a associação dinâmica de tratadores com exceções, considere o procedimento P a seguir:[8]

```
procedure P is
    ERROR: exception;

    procedure R;

    procedure Q is
    begin
        ...                          -- possibilidade de exceção (2)
        R;
        ...
    exception
        ...
        when ERROR =>                -- tratador E2
```

[6]Em inglês: *task;* módulo que opera em paralelo.

[7]A exceção TASKING_ERROR é levantada quando da comunicação entre duas tarefas. Cf. [DraftAda 83].

[8]Cf. [DraftAda 83].

```
        ...
    end Q;

    procedure R is
    begin
        ...                          -- possibilidade de exceção (3)
    end R;

begin -- P
    ...                              -- possibilidade de exceção (1)
    R; ... Q;
    ...
exception
    ...
    when ERROR =>                    -- tratador E1
    ...
end P;
```

Os seguintes casos podem ocorrer:

1. Se a exceção ERROR é levantada no corpo de P, o tratador E1 é usado para completar a execução de P e o controle volta ao seu invocador.

2. Se a exceção ERROR é levantada no corpo de Q, o tratador E2 é usado para completar a execução de Q. O controle retorna ao ponto de chamada de Q após a execução do tratador E2.

3. Se a exceção ERROR é levantada no corpo de R, chamado por P, sua execução é terminada e a mesma exceção é levantada no corpo de P. O tratador E1 é executado como em (1).

4. Se a exceção ERROR é levantada no corpo de R, chamado por Q, sua execução é terminada e a mesma exceção é levantada no corpo de Q. O tratador E2 é executado como em (2).

Essa identificação do tratador de uma exceção é dinâmica apesar do identificador de exceção ser conhecido estaticamente.

Dentro de um tratador, a exceção que causou a transferência para ele pode ser relevantada por um comando *raise* normal (mencionando o nome da exceção) ou por um comando *raise* sem mencionar o nome da exceção. O efeito de relevantar

a mesma exceção ou levantar uma outra dentro de um tratador é terminar a unidade de programa corrente e propagar a exceção correspondente.

Para suprimir a detecção de uma exceção em uma unidade de programa, deve-se usar a construção:

pragma SUPRESS (*identificador_de_exceção*);

Note-se que a exceção ainda pode ocorrer (ou através do comando *raise* ou pela sua propagação por um subprograma onde não foi supressa).

5.1.3 Exceções em Chill

Chill[9] foi desenvolvida com o objetivo de atender às necessidades de sistemas de comutação para telefonia como: manipulação de chamadas, teste e manutenção; sistemas operacionais; suportes "on-line" e "off-line"; testes de validação. Um programa em Chill consiste de três partes: descrição de dados,[10] descrição das ações que devem ser realizadas sobre eles e a estrutura do programa, isto é, a forma como os elementos de programa devem ser dispostos para formar uma entidade. Possibilita a execução concorrente de unidades de programa, tratamento de exceções e apresenta verificação rígida de tipos.

Exceções em Chill têm um nome. Um nome de exceção ou é predefinido pela linguagem ou definido pela implementação ou, ainda, definido pelo usuário. As duas primeiras podem ser levantadas por condições excepcionais durante a execução ou através da ação *cause*; as definidas pelo usuário podem ser levantadas apenas pela ação *cause*.

O formato de um tratador é:

ON (*lista_de_nome_de_exceção*) :
lista_de_comandos_executáveis
[ELSE *lista_de_comandos_executáveis*]
END

A cláusula ELSE trata qualquer exceção não presente em qualquer *lista_de_nomes_de_exceção* do tratador.

Tratadores de exceção podem ser associados a qualquer comando executável.[11]

Chill exige que, em qualquer comando, se possa determinar estaticamente para quais exceções um tratador apropriado pode ser encontrado e que se possa conhecer estaticamente qual exceção pode ocorrer em um comando.

[9] Cf. [Smedema 84].

[10] Em Chill, esses elementos são chamados *data objects*.

[11] Em Chill, tais comandos são chamados *action statements*.

Exceções levantadas em um procedimento podem ser tratadas a nível de sua declaração (isto é, no seu corpo) ou no ponto de chamada. No caso do tratamento ser feito na unidade invocadora, os nomes de exceção devem ser explicitamente listados no cabeçalho do procedimento (são as chamadas *exceções propagadas*).

Dado um texto de programa no qual uma exceção pode ocorrer, o tratador apropriado para ela, se existir, é determinado estaticamente como se segue, considerando primeiramente o menor comando no qual o texto está encaixado:

1. Um tratador para a exceção está associado a um comando. Quando a exceção é tratada, a execução continua a partir do próximo comando. Exemplo:

```
a(i) := 1
    ON (RANGEFAIL) : ação
       ELSE ação
    END;
```

2. Não há tratador para a exceção no comando mas este está incluído imediatamente em um comando composto: considera-se que a exceção foi levantada neste comando composto. Exemplo:

```
b: BEGIN
   a(i) := 1;
   END ON (RANGEFAIL) : ação
       ELSE ação
   END b;
```

3. O comando pertence à definição de um procedimento e não há um tratador apropriado. Há três possibilidades nesse caso:

   - Um tratador para a exceção é especificado após a definição do procedimento. A exceção é tratada por ele, o procedimento termina e a execução continua a partir do ponto de chamada.
   - Não há tratador para a exceção especificado após a definição do procedimento mas a exceção é listada no seu cabeçalho. Essa forma indica que a exceção deve ser propagada, isto é, deve ser tratada no ponto de chamada.

- Caso contrário, não há tratador.[12]

Exemplo:

```
gcd: PROC (a,b INT) (INT) EXCEPTIONS (badparameters);
     ASSERT a > 0 AND b > 0;
     DO WHILE a /= b;
        IF a > b
        THEN a := a - b;
        ELSE b := b - a;
        FI;
     OD;
     RESULT a;
END ON (ASSERTFAIL) : CAUSE badparameter;
    END gcd;
/* OVERFLOW não é tratado pelo procedimento! */
```

4. O comando pertence à definição de um processo e não há um tratador apropriado no seu corpo. Há duas possibilidades nesse caso:

   - Um tratador para a exceção é especificado após a definição do processo. O processo termina quando este é executado.
   - Caso contrário, não há tratador.

5. O comando pertence a uma alternativa ON ou à parte ELSE de um tratador. Considera-se que a exceção foi levantada na ação à qual o tratador está associado, como se este não fosse especificado para a ação. Em outras palavras, o próximo tratador para a exceção é procurado numa hierarquia de tratadores, de maneira a evitar uma repetição infinita. Exemplo:[13]

```
BEGIN
  a(i) := 1 ON (RANGEFAIL) : ação /* tratador 1 */
            END;
END ON (RANGEFAIL) : ação /* tratador 2 */
```

[12]Não há erro de compilação quando tratadores não podem ser encontrados para exceções que poderiam ocorrer. Certamente, há muitos casos em que um programa pode garantir que não ocorrerá uma exceção. No entanto, um erro dinâmico (em tempo de execução) ocorrerá se uma exceção for levantada sem o tratador apropriado.

[13]Todos os exemplos desse item foram retirados de [Smedema 84].

```
    END;
/* Quando a(i) causar RANGEFAIL, o tratador 1 será
   ativado. Se a ação nesse tratador também causar
   RANGEFAIL, o tratador 2 será ativado. */
```

5.1.4 Exceções em MC

O esquema de exceções em MC é um meio-termo entre as propostas de Clu, Ada e Chill, procurando ser bem simples, com as seguintes características principais:

- Um tratador de exceção está associado a um bloco. Caso uma exceção ocorra durante a execução de um comando, é feito um desvio para o tratador apropriado, declarado no bloco onde se encontra o comando.
- Um subprograma pode propagar uma exceção. O identificador da exceção, neste caso, deve ser listado no seu cabeçalho.
- Como conseqüência das duas anteriores, é possível determinar quais exceções um subprograma é capaz de atender (tratamento ou propagação). Esta é a sua característica principal.

Uma exceção pode ser declarada através da parte de declarações de um bloco ou através da parte de definições de um módulo, da maneira como se segue ou, então, ser uma exceção predefinida da linguagem, cujo identificador é penetrante.

exception *lista_de_identificadores*;

As unidades cujas execuções podem ser terminadas prematuramente por uma exceção são os blocos de subprogramas, processos e módulos. Exceções podem ser levantadas explicitamente, através do comando *raise* no caso das definidas em programas ou das predefinidas, ou pelo sistema, no caso das predefinidas:

raise *identificador_de_exceção*;

Tratadores de exceção podem ser especificados no final de um bloco, introduzidos pela palavra reservada **exceptions**. Por exemplo:

begin
    ...
exceptions
    when *lista_de_identificadores_de_exceção*
        do *seqüência_de_expressões* **end**;
    [**others** *seqüência_de_expressões* **end**]
end

Um tratador diz respeito às exceções cujos identificadores seguem o símbolo **when** e é invocado quando a exceção correspondente é levantada no bloco onde ocorre sua declaração. Se nenhum tratador é especificado para uma exceção, o tratador invocado é o que está associado à cláusula *others*, se esta ocorrer.

Uma exceção pode ser tratada por um tratador declarado em um bloco; pode ser propagada, somente no caso de subprogramas, se o identificador da exceção estiver presente no cabeçalho do subprograma a fim de que o tratamento seja provido pela unidade invocadora ou, ainda, pode interromper a execução da unidade.

O tratador de uma exceção é determinado como se segue:

1. O tratador é especificado no bloco onde ocorreu a exceção: a exceção é tratada por ele e a execução do bloco termina, com o retorno ao ponto de chamada, em se tratando de um subprograma, ou com o término da unidade executável, nos casos de processo e programa. Note-se que a exceção pode ser relevantada no tratador, através do comando *raise*, da forma:

   raise;

2. A exceção é especificada na lista de exceções do cabeçalho do subprograma: esta é a maneira de indicar que a exceção pode (também) ser tratada no ponto de chamada do subprograma. Neste caso, diz-se que a exceção é propagada. Note-se que uma exceção pode ser propagada mesmo que haja um tratador correspondente no bloco que estava sendo executado, através do relevantamento da mesma, usando-se o comando *raise*.

3. Caso contrário, não há tratador. A unidade executável termina com a condição de exceção obtida.

Exemplo:

```
function BuscaSequencial (var V: array [L..U: CARDINAL] of
                          INTEGER; x: INTEGER): CARDINAL;
    var i: CARDINAL;
begin
      i := V.L;
      while V[i] # x do i++ end;
      return i
exceptions
      when INDEX_ERROR do return V.L - 1 end
end BuscaSequencial;
```

ou, então:

```
function BuscaSequencial (var V: array [L..U: CARDINAL] of
                                    INTEGER; x: INTEGER): CARDINAL;
        exception (INDEX_ERROR);
    var i: CARDINAL;
begin
      i := V.L;
      while V[i] # x do i++ end;
      return i
end BuscaSequencial;
```

Note-se que, na primeira versão de *Busca_Sequencial*, a exceção INDEX_ERROR é tratada no bloco da função; na segunda versão, a exceção é propagada para a unidade invocadora da função. Qualquer outra exceção causa o término de execução da unidade à qual *BuscaSequencial* pertence.

Para suprimir a detecção de exceções em um subprograma, processo ou programa, devem-se utilizar atributos de supressão na sua declaração. A especificação destes é um atributo de compilação.

Comparação com Clu, Ada e Chill

Uma comparação entre exceções em MC e nas linguagens Clu, Ada e Chill é apresentada a seguir, nos seus aspectos principais:

- Declaração de identificadores de exceção em programas:
  - MC: Permitido em blocos e cabeçalhos de subprogramas.
  - Clu: Somente em cabeçalhos de subprogramas.
  - Ada: Permitido em blocos de subprograms.
  - Chill: Permitido em blocos e cabeçalhos de subprogramas.
- Declaração de identificadores de exceção no cabeçalho de subprogramas - significado:
  - MC: Exceções propagadas pelo subprograma.
  - Clu: Exceções que o subprograma trata.
  - Ada: —
  - Chill: Exceções que são propagadas.

- Declaração de tratador de exceções:
  - MC: No final de um bloco, não permite parâmetros.
  - Clu: No final de um comando, permite parâmetros.
  - Ada: No final de um bloco, não permite parâmetros.
  - Chill: No final de qualquer ação (comando executável ou declaração), não permite parâmetros.
- Unidade de programa que pode levantar uma exceção:
  - MC: Expressão ou declaração.
  - Clu: Subprogramas; o tratamento é feito na unidade invocadora.
  - Ada: Comando ou declaração.
  - Chill: Ação (comando executável ou declaração).
- Ação realizada quando o tratador conveniente não é encontrado no subprograma:
  - MC: Exceção propagada, se possível, ou erro.
  - Clu: Exceção predefinida *failure* é levantada.
  - Ada: Exceção relevantada, implicitamente, no ponto de chamada.
  - Chill: Exceção propagada, se possível, ou erro.

5.2 Vetores com Limites Abertos

Linguagens como Pascal e Modula-2 podem apresentar alguns inconvenientes quando se escrevem procedimentos que manipulam parâmetros efetivos que sejam vetores.

Em Pascal, os tipos t_1 e t_2, declarados a seguir, são vetores com tipos de índices diferentes e, portanto, são de tipos diferentes:

```
type t1 = array [1..50] of integer;
     t2 = array [1..70] of integer;
```

Pelo fato de procedimentos requererem parâmetros formais de um tipo específico, não é possível, por exemplo, escrever um procedimento genérico de ordenação de vetor que aceite um parâmetro efetivo tanto de tipo t_1 quanto de tipo t_2.

Como soluções para essa limitação, a padronização de Pascal pela ISO[14] inclui a característica chamada *vetor conforme*[15] e Modula-2, como uma solução mais restritiva, apresenta a característica chamada *vetor aberto*.[16] MC apresenta uma solução semelhante à do Pascal, através do conceito de *vetor com limites abertos*, com uma restrição de sua aplicação a variáveis dinâmicas e parâmetros de subprogramas.

5.2.1 Vetor Conforme – Solução de Pascal

Um parâmetro formal pode ser do tipo vetor conforme. Os parâmetros efetivo e formal, nesse caso, devem ter o mesmo número e tipo de índices e o mesmo tipo de componente.[17] Diz-se que o parâmetro conforma-se ao parâmetro efetivo em termos de tamanho.

Exemplo:[18]

```
procedure Sort (var a: array [low..high: integer] of Ctype);
    var i: integer;
        more: boolean;
begin { Sort }
    more := true;
    while more do begin
        more := false;
        for i := low to high-1 do begin
            if a[i] > a[i+1] then begin
              move_right (i);
              more := true
            end;
        end;
    end;
end { Sort };
```

Quando o procedimento *Sort* é chamado com um parâmetro de tipo vetor unidimensional, *low* e *high* recebem os valores dos limites inferior e superior, respectivamente, do parâmetro efetivo.

[14] Cf. [ISOPascal 82].

[15] Em inglês: *conformant array*.

[16] Em inglês: *open array*.

[17] Este tipo é distinto de qualquer outro tipo. Assim, duas ou mais especificações de vetor conforme absolutamente idênticas definem parâmetros formais com tipos distintos.

[18] Cf. [GhezJaz 87].

Vetores conformes não são uma solução genérica. Por exemplo, não é possível declarar um vetor local no procedimento *Sort* de tamanho *low..high* porque os limites de um subintervalo devem ser constantes conhecidas em tempo de compilação. Em MC, pode-se superar essa restrição com a alocação de vetores dinâmicos com limites abertos. Note-se que o tipo de componente de um vetor conforme pode ser qualquer, inclusive um vetor conforme.

5.2.2 Vetor Aberto – Solução de Modula-2

Um parâmetro formal pode ser do tipo vetor aberto. Os parâmetros efetivo e formal, nesse caso, devem ter o mesmo tipo de componente mas o intervalo de índices do vetor é dependente do parâmetro efetivo. O tipo do índice do parâmetro formal é CARDINAL; o limite inferior do parâmetro formal é sempre 0; seu limite superior é obtido através da função padrão HIGH, que devolve o número de elementos do vetor menos 1. Assim, se um vetor declarado como

```
a: ARRAY [m..n] OF CHAR
```

é parâmetro efetivo de um procedimento cujo parâmetro formal é da forma

```
s: ARRAY OF CHAR
```

então s[i] denota a[m+i] para i = 0,...,HIGH(s), onde HIGH(s) = n − m.

Exemplo:

```
PROCEDURE Sort (VAR a: ARRAY OF Ctype);
    VAR i: CARDINAL;
        more: BOOLEAN;
BEGIN /* Sort */
    more := TRUE;
    WHILE more DO
        more := FALSE;
        FOR i := 0 TO HIGH(a)-1 DO
            IF a[i] > a[i+1] THEN
              Move_Right (i);
              more := TRUE
            END
        END
    END
END /* Sort */;
```

Note-se que o tipo de componente de um vetor aberto não pode ser, por sua vez, um outro tipo vetor.

5.2.3 Vetor com Limites Abertos – Solução de MC

Parâmetros formais e variáveis dinâmicas podem ser do tipo vetor com limites abertos. A especificação dos limites de um vetor desse tipo é incompleta, sendo os limites definidos em tempo de execução. Registros e vetores comuns (com limites definidos) não podem apresentar, como componentes, vetores com limites abertos.[19] Note-se que o mesmo conceito de vetores abertos é aplicável a parâmetros formais e a variáveis dinâmicas.

A seguir, apresenta-se o conceito de *conformidade* entre dois tipos vetores que será utilizado pelas regras de passagem de parâmetros.

Conformidade

Sejam T_v um tipo vetor (com limites definidos ou abertos), com um índice de tipo T_i, e T_l o tipo dos identificadores de limites de um parâmetro formal de tipo vetor com limites abertos. Um vetor de tipo T_v é *semelhante a* ou *conforme com* um parâmetro formal de tipo vetor com limites abertos se todas as condições a seguir forem verificadas:

1. T_i é compatível para atribuição[20] com T_l.
2. Os limites de T_i, quando definidos, pertencem ao intervalo fechado definido por T_l.
3. O tipo de componente de T_v é equivalente[21] ao tipo de componente do parâmetro formal ou o tipo do componente de T_v é conforme com o tipo do componente do parâmetro formal.

Vetores com Limites Abertos como Parâmetro Formal

Um parâmetro formal de tipo vetor com limites abertos tem o seu tamanho estabelecido quando da associação do parâmetro efetivo correspondente na chamada do subprograma. O parâmetro efetivo deve ser *conforme* com o parâmetro formal.

Se o mecanismo de passagem é por valor, [22] a declaração de parâmetro formal

[19] Exceto através de apontadores.
[20] Cf. 5.6.6.
[21] Idem.
[22] Cf. 5.3.3.

p: **array** [L_1..U_1: T_1, L_2..U_2: T_2,..., L_n..U_n: T_n] **of** T

é funcional mas não sintaticamente equivalente à declaração de um tipo registro da forma:

type T' = **record**
L_1,U_1: T_1;
L_2,U_2: T_2;
⋮
L_n,U_n: T_n;
"Vetor_Componente": **array** [L_1..U_1], [L_2..U_2],...,
[L_n..U_n] **of** T
end

e à declaração do parâmetro formal p como

p: T'

Os valores dos limites efetivos L_i, U_i e dos elementos de "Vetor_Componente" [i_1,...,i_n] são copiados quando da associação entre os parâmetros efetivo e formal. O acesso a cada componente da estrutura, no subprograma, segue as mesmas regras utilizadas para acesso aos campos de registros. O pseudo-identificador "Vetor_Componente", no entanto, não deve ser usado para se fazer acesso aos elementos do vetor.

Exemplo:

```
var v1: array [1..Maxv1] of REAL;

function Add_Vector (p: array [low..high: INTEGER] of REAL): REAL;
    var i: INTEGER;
        s: REAL := 0.0;
begin
    for i := p.low to p.high do s := s + p[i] end;
    return s
end Add_Vector;
```

Na chamada `Add_Vector(v1)`, p.low = 1, p.high = Maxv1. No corpo do subprograma, `p[i]` pode ser visto como uma abreviação de

p."Vetor_Componente"[i]. Os limites do vetor se comportam como constantes cujos valores são definidos quando da associação entre os parâmetros efetivo e formal.

Se o mecanismo de passagem é por variável,[23] a declaração do parâmetro formal

$$\textbf{var pp: array } [L_1..U_1\text{: } T_1, L_2..U_2\text{: } T_2, \ldots, L_n..U_n\text{: } T_n] \textbf{ of } T$$

é funcional mas não sintaticamente equivalente à declaração de um tipo registro da forma:

type T'' = **record**
L_1,U_1: T_1;
L_2,U_2: T_2;
⋮
L_n,U_n: T_n;
"Apontador_Componente": **pointer to array** $[L_1..U_1]$, $[L_2..U_2], \ldots, [L_n..U_n]$ **of** T
end

e à declaração do parâmetro formal pp como

$$\textbf{var pp: } T''$$

Assim como no caso de parâmetro por valor, os valores dos limites efetivos L_i, U_i são copiados quando da associação entre os parâmetros efetivo e formal; no entanto, os elementos do vetor não são copiados mas é passado o endereço deste como parâmetro efetivo. O acesso aos componentes da estrutura, no subprograma, também segue as regras utilizadas para acesso aos campos de registros. O pseudo-identificador "Apontador_Componente" e o operador de derreferenciação ↑ não devem ser usados para se fazer acesso aos elementos do vetor.

Exemplo:

```
var v2: array [1..Maxv2] of INTEGER;

procedure Sort (var a: array [low..high: INTEGER] of INTEGER);
   var i: INTEGER;
```

[23] Cf. 5.3.3.

```
        more: BOOLEAN := TRUE;
begin /* Sort */
    while more do
        more := FALSE;
        for i := a.low to a.high-1 do
            if a[i] > a[i+1] then
              Move_Right (i);
              more := TRUE
            end
        end
    end
end Sort;
```

Na chamada Sort(v2), a.low = 1, a.high = Maxv2. No corpo do subprograma, a[i] pode ser visto como uma abreviação de a."Apontador_Componente"↑[i]. Também como no caso de parâmetros por valor, os limites do vetor se comportam como constantes cujos valores são definidos quando da associação entre os parâmetros efetivo e formal.

A especificação de parâmetros formais

$$p_1,p_2,\ldots,p_m\text{: } \textbf{array } [L_1..U_1\text{: } T_1, L_2..U_2\text{: } T_2,\ldots, L_n..U_n\text{: } T_n] \textbf{ of } T$$

é permitida e equivalente a

$$\begin{aligned} p_1&\text{: } \textbf{array } [L_1..U_1\text{: } T_1, L_2..U_2\text{: } T_2,\ldots, L_n..U_n\text{: } T_n] \textbf{ of } T;\\ p_2&\text{: } \textbf{array } [L_1..U_1\text{: } T_1, L_2..U_2\text{: } T_2,\ldots, L_n..U_n\text{: } T_n] \textbf{ of } T;\\ &\vdots\\ p_m&\text{: } \textbf{array } [L_1..U_1\text{: } T_1, L_2..U_2\text{: } T_2,\ldots, L_n..U_n\text{: } T_n] \textbf{ of } T; \end{aligned}$$

mas os tipos dos parâmetros p_i não são equivalentes entre si, conforme apresentado em 5.6.6.

Finalmente, note-se que esta solução é próxima à adotada pelo Pascal-ISO.[24]

Vetor com Limites Abertos como Variável Dinâmica

A criação do vetor, como variável dinâmica, dá uma instância aos seus limites.

Uma declaração de tipo

[24] Cf. [ISOPascal 82].

array [$L_1..U_1$: T_1, $L_2..U_2$: T_2,..., $L_n..U_n$: T_n] **of** T

é funcional mas não sintaticamente equivalente à declaração de um tipo registro da forma:

```
record
  L1,U1: T1;
  L2,U2: T2;
    :
  Ln,Un: Tn;
  "Vetor_Componente": array [L1..U1], [L2..U2],...,
                             [Ln..Un] of T
end
```

Os valores dos limites do vetor são obtidos quando da criação da variável dinâmica deste tipo, através de um procedimento de alocação adequado. No caso de se utilizar o procedimento predefinido NEW, do módulo SYSTEM, além do parâmetro que indica o tipo da variável a ser criada, deve-se passar uma lista de expressões que serão usadas para dar os limites do vetor aberto.

Exemplo:

```
type
    T1 = array [LT1..UT1: INTEGER] of INTEGER;
    T2 = array [L1T2..U1T2: CHAR, L2T2..U2T2: INTEGER] of CHAR;
    ApT1 = pointer to T1;
    ApT2 = pointer to T2;
var
    p,q: ApT1;
    r,s: ApT2;
      :
    NEW(p,1,10);
    NEW(q,-1,1);
      :
    NEW (r,'A','Z',1,26);
    NEW (s,'0','9',0,9);
```

Uma vez criado o vetor, os componentes do registro que dão os seus limites se comportam como constantes (no exemplo, p↑.LT1 = 1, p↑.UT1 = 10, r↑.L1T2

= 'A', todos constantes). De maneira semelhante ao parâmetro por valor, o pseudo-identificador "Vetor_Componente" não deve ser usado para se fazer acesso aos elementos do vetor. No exemplo acima, p↑[i] pode ser visto como uma abreviação de p↑."Vetor_Componente"[i] e r↑[c,j] como uma abreviação de r↑."Vetor_Componente"[c,j].

Pelas regras de compatibilidade de MC,[25] a atribuição p := q é permitida mas p↑ := q↑ não é. Observe-se, ainda, que variáveis dinâmicas que são vetores com limites abertos podem ser usadas como parâmetros efetivos.

5.3 Passagem de Parâmetros

A passagem de parâmetros permite a comunicação de dados entre unidades de programa, com a transferência de diferentes valores em cada ponto de chamada, proporcionando vantagens em termos de legibilidade e facilidade para modificações.

A maioria das linguagens usa o método posicional para a associação dos parâmetros efetivos[26] aos formais[27] nas chamadas de subprogramas. Assim, se um procedimento é declarado como

procedure P (f_1,...,f_n);

e a chamada é

P (e_1,...,e_n);

o método posicional implica que o parâmetro formal f_i é associado ao parâmetro efetivo e_i, $i = 1, \ldots, n$.

5.3.1 Passagem de Dados em Geral

A discussão a seguir diz respeito à passagem de dados em geral como parâmetros. Outras entidades passadas como parâmetros, que não serão discutidas no presente texto, são subprogramas e tipos.

---

[25] Cf. 5.6.6.

[26] Em inglês: *actual parameters.*

[27] Em inglês: *formal parameters.*

Chamada por Referência (ou Compartilhamento)

Neste método, é passado o endereço do parâmetro efetivo. Uma referência ao parâmetro formal correspondente é tratada como uma referência à posição cujo endereço foi passado. Uma variável que é passada dessa maneira é compartilhada entre a unidade de programa que fez a chamada e o subprograma que foi chamado. Se o parâmetro efetivo é uma expressão (que não uma variável) ou uma constante, o subprograma recebe o endereço da posição temporária que contém o valor do parâmetro efetivo. Algumas linguagens tratam esse caso como erro.

Chamada por Cópia

Neste método, o parâmetro formal se comporta como uma variável local. É possível classificar a chamada por cópia em três modos, de acordo com a maneira que as variáveis locais que correspondem aos parâmetros efetivos recebem seus valores iniciais e o modo como seus valores afetam os valores efetivos no retorno à unidade invocadora. Esses modos são:

- *Chamada por Valor:* O valor do parâmetro efetivo, calculado na unidade que faz a chamada, é usado para dar o valor inicial para o parâmetro efetivo correspondente. Neste método não se permite que qualquer informação seja passada de volta para a unidade invocadora; modificações no parâmetro formal não afetam a unidade invocadora.

- *Chamada por Resultado:* A variável local correspondente ao parâmetro formal não recebe valor na entrada do subprograma, mas seu valor, ao seu término, é copiado de volta na posição correspondente ao parâmetro efetivo. Neste método não se permite que qualquer informação seja passada para a unidade invocada.

- *Chamada por Valor-Resultado*: A variável local que denota o parâmetro formal recebe um valor inicial na entrada do subprograma (como na chamada por valor) e devolve um valor ao seu término (como na chamada por resultado).

Chamada por Nome

O termo *chamada por nome* foi introduzido em Algol 60 como o método adotado implicitamente[28] para manipulação de parâmetros.

[28]Tradução para o termo *default*.

Na chamada por nome, o parâmetro formal denota não uma variável local do subprograma mas uma expressão no contexto da unidade invocadora. Basicamente, neste mecanismo, cada ocorrência do parâmetro formal é substituída textualmente pela expressão que representa o parâmetro efetivo e essa expressão é avaliada no seu contexto original.

Embora a chamada por nome seja um mecanismo poderoso mas difícil de implementar de maneira eficiente, pode produzir resultados inesperados, como mostra o exemplo a seguir:[29]

```
procedure swap (a,b: integer);
    var t: integer;
begin
    t := a; a := b; b := t
end swap;
```

Quando o comando de chamada `swap (i,A[i])` é encontrado, supondo chamada por nome, a seqüência de comandos resultante é:

$$t := i;\ i := A[i];\ A[i] := t$$

que não produz o resultado desejado.

Outra armadilha para o programador é que o parâmetro efetivo que é, conceitualmente, substituído no texto da unidade invocada, pertence ao ambiente da unidade invocadora. O problema é a dificuldade encontrada pelo programador em antever a associação, em tempo de execução, dos parâmetros formais e efetivos devido, principalmente, à possível duplicidade de nomes.

5.3.2 Mecanismos de Passagem em Algumas Linguagens

Modula-2

Há dois tipos de parâmetros, chamados *parâmetros por valor* e *parâmetros por variável*. Pela descrição de [Wirth 85], pode-se concluir que os mecanismos de passagem são passagem por valor e por referência, respectivamente, mas isso não está estabelecido no texto, bem como a ordem de avaliação dos parâmetros efetivos.[30] A passagem dos parâmetros é posicional.

[29] Cf. [Horowitz 84,GhezJaz 87].

[30] Cf. [Torbett 87a].

C

Parâmetros são passados por valor somente, com o parâmetro formal recebendo uma cópia do parâmetro efetivo na chamada da função. Para obter o efeito de uma passagem por referência deve-se passar, explicitamente, o endereço de uma variável (ou o valor de um apontador; cf. [KernRit 78]). Quando um nome de vetor aparece como um argumento de uma função, seu endereço inicial é passado como parâmetro; seus elementos não são copiados. A passagem de parâmetros é posicional mas sua ordem de avaliação não está determinada.

Ada

Há três classes de parâmetros, que são definidas em termos de seu comportamento abstrato, isto é, sem se referirem ao mecanismo de passagem utilizado:[31]

in: Dentro do subprograma, o parâmetro formal é considerado como uma constante local cujo valor é fornecido pelo parâmetro efetivo correspondente.

out: Dentro do subprograma, o parâmetro formal é considerado como uma variável local; seu valor é atribuído ao parâmetro efetivo correspondente como um resultado da execução do subprograma.

in out: Dentro do subprograma, o parâmetro formal é considerado como uma variável local e permite acessos e atribuições ao parâmetro efetivo correspondente.

Embora as três classes estejam relacionadas com a chamada por cópia (chamadas por valor, resultado e valor-resultado, respectivamente), discutida em 5.3.1, para cada uma delas uma implementação pode escolher o mecanismo a ser utilizado. Essa escolha pode ser influenciada pelos tamanhos dos objetos considerados. Para objetos grandes, uma implementação por referência é freqüentemente mais eficiente. Objetos pequenos são mais apropriados para passagem por cópia. Em situações normais, a semântica de um programa não será afetada pelo fato da implementação da passagem de parâmetros ser por referência ou por cópia. Situações anormais são: variáveis compartilhadas, exceções e sinônimos,[32] como no exemplo a seguir:

```
procedure P (X: in out INTEGER) is
begin
```

[31] Cf. [Ichbiah et al. 79].

[32] Em inglês: *aliasing*.

```
        x := x + 1;
        x := x + A
    end;
```

A chamada P(A) pode ser ilegal pois o acesso a A pode ser feito de duas maneiras (diretamente ou via o parâmetro x). Assim, se A = 2 antes da chamada P(A), o valor de A após o retorno será 5 em uma implementação por cópia e 6 em uma implementação por referência.

A passagem de parâmetros pode ser feita tanto de maneira posicional quanto baseada no nome do parâmetro formal. Como exemplo, o procedimento P acima poderia ser chamado de duas maneiras: P(A) e P(x := A). A convenção por nomes pode ser usada em conjunção com a convenção posicional, com esta aparecendo primeiramente. A ordem de avaliação dos parâmetros não é especificada.

Chill

Há, basicamente, dois mecanismos de passagem de parâmetros, chamados *passagem por valor* e *passagem por endereço*:[33]

- *Passagem por Valor:* É a chamada por cópia discutida em 5.3.1; o atributo IN se refere à chamada por valor; OUT, à chamada por resultado e IN OUT, à chamada por valor-resultado. Nos casos de OUT e IN OUT, o parâmetro efetivo deve ser um endereço.
- *Passagem por Endereço:*[34] É a chamada por referência discutida em 5.3.1; o parâmetro formal deve vir qualificado pelo atributo LOC e é passado o endereço do parâmetro efetivo. Se este é uma expressão que não tem um endereço, uma posição contendo o resultado da expressão é implicitamente criada e passada para o procedimento no ponto de chamada. O tempo de vida dessa posição é a chamada do procedimento.

Exemplo:[35]

```
DCL aa,bb,cc,ee,d INT := 2;

ex: PROC (a INT, b INT INOUT, c INT OUT, e INT LOC) (INT);
```

[33]Note-se que a acepção de Chill difere da apresentada nesse texto para a passagem por valor; cf. [GhezJaz 87].

[34]Em inglês: *location*.

[35]Cf. [Smedema 84].

```
        a := 2 * a;
        b := 2 * b;
        e := 2 * e;
        RESULT a;
END ex;

d := ex (aa,bb,cc,ee);
```

Após a chamada, obtêm-se os seguintes valores:

$$aa = 2,\ bb = 4,\ cc = \textit{indefinido},\ ee = 4,\ d = 4$$

A passagem de parâmetros é posicional. A ordem de avaliação dos parâmetros não é especificada.

5.3.3 Passagem de Parâmetros em MC

Devido à sua simplicidade, o mecanismo de passagem de parâmetros adotado em MC é semelhante ao de Modula-2.

A discussão a seguir diz respeito à parte fixa dos parâmetros de um subprograma em MC. A discussão sobre a parte variável de parâmetros é feita na seção 5.4.

Parâmetros formais são considerados locais ao subprograma. Há duas classes de parâmetros:

- *Parâmetro por Valor:* É uma variável local cujo valor inicial é dado pelo parâmetro efetivo correspondente. Atribuições ao parâmetro por valor não têm efeito no parâmetro efetivo.
- *Parâmetro por Variável* (identificado pela palavra reservada **var**): É um nome local para o parâmetro efetivo correspondente, que deve ser uma variável ou um componente de uma variável. Assim, nenhuma variável é alocada para o parâmetro formal e seu identificador denota a variável que é passada como parâmetro efetivo. Qualquer atribuição ao parâmetro por variável é equivalente a uma atribuição ao parâmetro efetivo correspondente.

O parâmetro por valor é implementado através do esquema de passagem por valor; o parâmetro por variável, através do esquema de passagem por referência, conforme apresentado em 5.3.1. A associação entre parâmetros formais e efetivos é feita antes da chamada do subprograma e de maneira posicional. A avaliação dos parâmetros é feita da esquerda para a direita; efeitos laterais são propagados nesta ordem.

5.4 Número Variável de Parâmetros em Subprogramas

A principal aplicação de um subprograma que aceita um número variável de parâmetros é, sem dúvida, a implementação de rotinas de E/S. A seguir são apresentadas soluções em algumas linguagens, não somente para o problema de E/S, mas aplicáveis, em muitas delas, a outros casos mais genéricos.

5.4.1 Pascal

Pascal não apresenta uma solução genérica que permita escrever subprogramas que aceitam um número variável de parâmetros. O caso particular de E/S é tratado pelos procedimentos predefinidos *read* e *write* que, quebrando a convenção da linguagem, aceitam um número variável de parâmetros e de tipos variados.[36]

read (f,v) é equivalente a x := f↑; get (v)

write (f,e) é equivalente a f↑ := e; put (f)

Observação: As equivalências acima não se verificam se f não é um arquivo de tipo texto.[37]

Os procedimentos *read* e *write* também admitem múltiplos argumentos. Assim,

$$\text{read } (f,v_1,\ldots,v_n)$$

é equivalente a

$$\textbf{begin read } (f,v_1);\ \ldots;\ \text{read } (f,v_n)\ \textbf{end}$$

e

$$\text{write } (f,e_1,\ldots,e_n)$$

é equivalente a

$$\textbf{begin write } (f,x_1);\ \ldots;\ \text{write } (f,x_n)\ \textbf{end}$$

[36] Cf. [Horowitz 84]. Alguns autores consideram os procedimentos *read* e *write* como pseudo-procedimentos, justamente por quebrarem as regras de parâmetros da linguagem.

[37] Cf. [Cooper 83].

5.4.2 C

[KernRit 78] afirma que não há maneira satisfatória de se escrever uma função portável que aceite um número variável de parâmetros porque não existe maneira portável da função invocada determinar quantos argumentos foram realmente passados para ela na chamada. No entanto, geralmente não há problemas se uma função invocada não usa um argumento que não foi passado e se os tipos dos parâmetros anteriores (da esquerda para a direita) são consistentes com os tipos dos parâmetros formais da função.

A função *printf* é o exemplo típico de uma função em C com um número variável de parâmetros. Ela utiliza a informação contida no primeiro parâmetro (uma cadeia com formatos de saída para expressões) para determinar quantos são os argumentos seguintes, ou seja, quantos se espera estejam presentes na chamada, e seus tipos. Ela não funcionará corretamente se a chamada não fornecer um número de parâmetros suficiente, de acordo com o primeiro parâmetro, ou se os seus tipos forem inconsistentes com a informação dada por ele. É uma função não portável e deve ser modificada para os diferentes ambientes onde deve ser utilizada.

Uma outra solução alternativa é se os parâmetros são de tipos conhecidos, pode-se marcar o final de uma lista de parâmetros efetivos com um valor especial de um parâmetro (por exemplo, zero no caso de inteiros).

5.4.3 Ada

Não é possível, em Ada, escrever um procedimento com um número de parâmetros variável; no entanto, a chamada pode apresentar um número variável de parâmetros efetivos.

A um parâmetro formal na declaração de um subprograma pode ser associado um valor a ser-lhe atribuído em caso de omissão do parâmetro efetivo na chamada do subprograma.[38] Essa facilidade é possível apenas para parâmetros de classe *in*. Exemplo:[39]

```
procedure ACTIVATE ( PROCESS: in PROCESS_NAME;
                     AFTER: in PROCESS_NAME := NO_PROCESS;
                     WAIT: in TIME := 0.0;
                     PRIOR: in BOOLEAN := FALSE );
```

O parâmetro PROCESS precisa estar presente em todas as chamadas, pois nenhum valor lhe foi associado para ser atribuído numa chamada. Por outro lado,

[38] Em inglês: *default value*.
[39] Cf. [Ichbiah et al. 79].

os parâmetros AFTER, WAIT e PRIOR podem ser omitidos. Assim, as chamadas a seguir de ACTIVATE são equivalentes:

```
ACTIVATE (PROCESS := X, AFTER := NO_PROCESS, WAIT := 0.0,
          PRIOR := FALSE);

ACTIVATE (PROCESS := X);
```

Outras exemplos de chamadas são:

```
ACTIVATE (X, AFTER := Y);
ACTIVATE (X, WAIT := 50.0*SECONDS, PRIOR := TRUE);
```

5.4.4 MC

Um subprograma, em MC, pode apresentar um número variável de parâmetros, que devem ser especificados após os parâmetros formais fixos.

A parte de parâmetros variáveis de um subprograma consiste, basicamente, na declaração de um vetor com limites abertos, considerado da mesma maneira que no caso de parâmetro por valor em termos de acesso,[40] e na especificação de funções de conversão que devem ser aplicadas automaticamente aos parâmetros efetivos na chamada do subprograma ou ao seu término, na dependência do vetor ser de entrada ou de saída. Note-se que é possível especificar, em um subprograma, apenas uma das classes de parâmetros variáveis.

O vetor aberto de parâmetros variáveis pode ser de entrada (**inparray**) ou de saída (**outparray**). Os identificadores de limites do vetor se comportam como constantes e recebem valores na chamada do subprograma, sendo permitido somente um par de limites (ou seja, um vetor de apenas uma dimensão). O valor do limite inferior será o menor valor dentre os possíveis de seu tipo; o limite superior será o valor do inferior mais o número de expressões (menos um) que serão consideradas como parâmetros efetivos.[41] Podem ser especificadas, entre os símbolos '(' e ')', funções de conversão a serem aplicadas aos parâmetros efetivos na entrada do subprograma ou na saída do mesmo, conforme a classe de parâmetro formal variável. O identificador qualificado que segue o símbolo **of**

[40] Cf. 5.2.3.

[41] O termo "mais" deve ser entendido, nesse contexto, como relação de ordem entre os valores de um tipo ordinal. Assim, caso declarado um procedimento S (p: **inparray** [1..u: BOOLEAN] **of** INTEGER), a chamada S (4,12) resultará em p.l = FALSE e p.u = TRUE no corpo de S.

deve ser um sinônimo de um tipo qualquer, a menos de vetor com limites abertos. Seja esse tipo T_{pv} na discussão a seguir.

Na chamada do subprograma é criado um vetor com limites abertos com tipo de componente T_{pv}; seu tamanho é dependente do número de expressões especificadas como parâmetros efetivos, descontadas as que foram consideradas como parâmetros efetivos fixos.

As funções de conversão listadas no cabeçalho do subprograma devem ter um único argumento. No caso de parâmetros de entrada, as funções não podem ter argumentos de tipos equivalentes entre si[42] e seus tipos de resultados devem ser equivalentes ao tipo T_{pv}. No caso de parâmetros de saída, as funções devem ter seus argumentos equivalentes ao tipo T_{pv} mas seus resultados não podem ser de tipos equivalentes entre si.

Se o parâmetro variável é de entrada, a cada parâmetro efetivo da parte variável cujo tipo resultante, T_e, não é equivalente ao tipo T_{pv}, é aplicada a função especificada no seu cabeçalho cujo argumento é de tipo equivalente ao tipo T_e, caso contrário, não há aplicação de função. Os resultados obtidos dessa forma são usados, então, para dar os valores iniciais ao vetor aberto da parte variável de parâmetros.

Se o parâmetro variável é de saída, cada parâmetro efetivo da parte variável deve ser uma variável ou o componente de uma variável, de tipo T_v. Na chamada do subprograma, o vetor aberto não tem os conteúdos de seus elementos definidos. À saída, a cada elemento do vetor, de tipo T_{pv}, é aplicada a função de conversão especificada no seu cabeçalho cujo resultado é de tipo equivalente ao tipo T_v se T_v e T_{pv} não forem equivalentes; caso contrário, não há aplicação de função. Os resultados obtidos dessa forma são atribuídos aos parâmetros efetivos correspondentes.

Exemplo:

```
type String = array [1..MAXSTR] of CHAR;

function ShortCard_to_Str (SHORTCARD): String;
function ShortInt_to_Str (SHORTINT): String;
function Cardinal_to_Str (CARDINAL): String;
function Integer_to_Str (INTEGER): String;
function Str_to_ShortCard (String): SHORTCARD;
function Str_to_ShortInt (String): SHORTINT;
function Str_to_Cardinal (String): CARDINAL;
```

[42] Cf. 5.6.6.

```
function Str_to_Integer (String): INTEGER;

                    ⋮

procedure Leitura_de_Inteiros (arq: TextFile; vl: outparray [l..n: SHORTCARD]
   (Str_to_ShortCard,Str_to_ShortInt,Str_to_Cardinal,Str_to_Integer) of String);
   var j: SHORTCARD;
begin
      for j := vl.l to vl.n do
          SkipSeparators (arq);
          ReadStr (arq, vl[j])
      end
end Leitura_de_Inteiros;

procedure Escrita_de_Inteiros (arq: TextFile; ve: inparray [l..n: SHORTCARD]
   (ShortCard_to_Str,ShortInt_to_Str,Cardinal_to_Str,Integer_to_Str) of String);
   var j: SHORTCARD;
begin
   for j := ve.l to ve.n do
       WriteStr (arq, ve[j]);
       WriteStr (arq, ' ')
   end
end Escrita_de_Inteiros;
       ⋮
var
    entr,sai: TextFile;
    i,j: INTEGER;
    l,m: SHORTCARD;
    n: SHORTINT;
```

Chamadas corretas dos procedimentos *Leitura_de_Inteiros* e *Escrita_de_Inteiros* seriam:[43]

```
Leitura_de_Inteiros (entr,i);
                /* função aplicada: Str_to_Integer;
```

[43]Neste exemplo, supõe-se que *ReadStr(f,v)* devolve, em *v*, a próxima cadeia de caracteres, separada de outra por algum conjunto predefinido de caracteres delimitadores, do arquivo de entrada *f* enquanto que *WriteStr(f,v)* escreve a cadeia de caracteres dada por *v* no arquivo de saída *f*.

```
                limites: 0,0 */
Leitura_de_Inteiros (entr,i,m,n);
        /* funções aplicadas: Str_to_Integer,
                Str_to_ShortCard, Str_to_ShortInt;
                limites: 0,2 */
Escrita_de_Inteiros (sai,i+2,i,7);
        /* funções aplicadas: Integer_to_Str,
                Integer_to_Str, ShortCard_to_Str;
                limites: 0,2 */
Escrita_de_Inteiros (sai,n*2,INTEGER(n)*j);
        /* funções aplicadas: ShortInt_to_Str, Integer_to_Str;
                limites: 0,1 */
```

O mecanismo de passagem utilizado para implementar esse esquema é a passagem por cópia (passagem por *valor*, no caso de parâmetros de entrada e passagem por *resultado*, no caso de saída).[44]

5.5 Processos

Em muitas aplicações, principalmente naquelas que tratam de programação de sistemas, é interessante modelar um sistema como um conjunto de unidades, chamadas *unidades concorrentes*, cujas execuções ocorrem em paralelo.[45]

Há várias classes de controle de unidades concorrentes. Uma delas é o *modelo de unidade simétrica*;[46] neste modelo, um certo número de unidades de programa, chamadas *corrotinas*, podem cooperar para sua execução inter-relacionada mas apenas uma delas pode ser executada em um dado instante. A execução de unidades simétricas é chamada, às vezes, *quase-concorrência*. Uma outra classe de concorrência, exibida em sistemas de computação maiores, supondo que mais de um processador está disponível, várias unidades de programa são executadas de maneira simultânea, literalmente. Esta é a *concorrência física*. Um pequeno relaxamento neste conceito de concorrência permite a suposição de múltiplos processadores quando, na realidade, apenas um está disponível, sendo conhecida como *concorrência lógica*.

Na discussão apresentada nesta seção, não é relevante o número de processadores. O termo *concorrência* será usado na sua conotação geral, tanto de concorrência física quanto lógica.

[44] Cf. 5.3.1.

[45] Embora possam ou não ser executadas realmente em paralelo.

[46] Cf. [Sebesta 89].

Conceitos Fundamentais

Uma corrotina é um subprograma que apresenta múltiplos pontos de entrada, que são controlados pela própria corrotina e os meios de manter seu contexto entre ativações. Uma corrotina, freqüentemente, é executada de maneira parcial, transferindo seu controle para outra corrotina (através de um comando *resume*). Quando o controle volta para ela, a execução continua a partir do último comando executado (isto é, *resume*) quando estava ativa.

Processo é uma unidade que é executada de maneira concorrente (ou paralela).

Um sistema concorrente pode ser considerado como um conjunto de processos, cada processo sendo representado por uma unidade de programa. Processos são *concorrentes* se suas execuções podem ser (conceitualmente) sobrepostas no tempo. Processos são *disjuntos* se descrevem atividades que não interagem com outras, isto é, não precisam fazer acesso a objetos compartilhados ou comunicar-se. Neste caso, o resultado da execução de um processo é independente de outros. São mais freqüentes, no entanto, os processos que devem interagir entre si. Interações podem ocorrer por *competição* por um recurso compartilhado, que deve ser usado em exclusão mútua ou por *cooperação*, para se atingir uma meta comum. Uma interação correta pode ser garantida pelo uso de *comandos* (ou *primitivas*) *de sincronização* (comunicação), providos pela linguagem de programação.

Três mecanismos básicos para sincronização serão apresentados a seguir, a saber: *semáforos*, *monitores* e *rendezvous*.[47] Segundo [GhezJaz 87], eles representam, dentre várias outras propostas presentes na literatura, a evolução histórica do conceito de sincronização de unidades concorrentes e têm sido incorporados em linguagens de programação largamente disponíveis. Todos eles supõem um modelo de processos concorrentes com memória compartilhada, isto é, tais processos devem ter acesso a uma memória comum.

A seguir, uma breve descrição dos três mecanismos:

Semáforos: Semáforos foram criados por Edsger Dijkstra em 1965 e posteriormente introduzidos como primitivas de sincronização em Algol 68. Um semáforo é um objeto que só pode assumir valores inteiros e que pode ser operado pelas primitivas P e V. Estas são operações indivisíveis e atômicas, isto é, dois processos não podem executar as operações P e V no mesmo semáforo ao mesmo tempo. Associada a um semáforo está uma lista de processos que esperam ser completados. As definições de P e V são:

[47] Não há uma tradução para esse termo no contexto de programação concorrente. O termo em francês significa *encontro*.

P(s): **if** s > 0 **then** s := s − 1
else *suspenda o processo atual*
V(s): **if** *há um processo suspenso no semáforo s*
then *libere o processo para continuar*
else s := s + 1

Programação com semáforos requer que seja associado um semáforo a cada condição de sincronização. Sua utilização, no entanto, requer boa disciplina por parte do programador para que execute P antes de fazer acesso a um recurso compartilhado e V após liberá-lo.

Monitores: Monitores foram propostos por Brinch Hansen e Hoare como um mecanismo de sincronização de alto nível e foram implementados em Pascal Concorrente e Mesa.[48] Descrevem tipos abstratos de dados em um ambiente concorrente. A exclusão mútua em acessos às operações que manipulam as estruturas de dados compartilhadas são garantidas automaticamente pela implementação. A cooperação no acesso às estruturas de dados compartilhadas deve ser programada explicitamente através das primitivas do monitor, *delay* e *continue*. Um monitor contém dados locais, um conjunto de procedimentos e, geralmente, alguns comandos para dar valores iniciais a seus dados. Se um processo invoca um procedimento do monitor e não há outro executando qualquer procedimento seu, poderá, então, seguir e executar o procedimento invocado. Caso contrário, entrará numa fila de processos do monitor para esperar sua vez.

Rendezvous: Rendezvous é um esquema provido por Ada para descrever a sincronização entre processos concorrentes (*tarefa*). Uma tarefa é muito similar a um módulo, tendo em sua parte de declarações várias *declarações de entradas*. *Entradas* podem ser invocadas por outras tarefas da mesma maneira que procedimentos. No entanto, diferentemente destes, o corpo da uma entrada não é executado imediatamente após sua chamada, mas somente quando a tarefa a quem pertence a entrada estiver pronta para aceitar a chamada através da execução de um comando *accept* correspondente. Nesse ponto, ambas as tarefas podem ser vistas como que se encontrando em um *rendezvous*. Se a tarefa invocadora executa a chamada antes que a tarefa invocada execute um comando *accept*, a primeira é suspensa até que o rendezvous ocorra. De maneira similar, uma suspensão da tarefa invocada ocorre se um comando *accept* é executado antes da chamada

[48] Cf. [Hansen 75] e [Mitchell et al. 79], respectivamente.

correspondente. Assim, as duas tarefas que se encontraram no rendezvous podem prosseguir paralelamente.

5.5.1 Modula-2

A filosofia do desenvolvimento da linguagem de N. Wirth está centrada no conceito que simplicidade é uma característica essencial para uma boa linguagem de programação. Modula-2 procura refletir esse conceito. Não há construções, tipos de dados ou operadores nela para controle de unidades concorrentes. Várias implementações, no entanto, como sugerido por Wirth,[49] incluem um módulo que provê as facilidades para a construção e uso de corrotinas. Este módulo é chamado *Process*.

As corrotinas de Modula-2 são chamadas processos. São compostas por procedimentos sem parâmetros usando as facilidades do módulo SYSTEM.

O módulo de definição Process, apresentado por Wirth, é:

```
DEFINITION MODULE Process;

   TYPE SIGNAL;

PROCEDURE StartProcess (P: PROC; n: CARDINAL);
   /* inicia um processo concorrente com o procedimento P e
   uma área de trabalho de tamanho n (palavras) */

PROCEDURE SEND (VAR s: SIGNAL);
   /* um processo que espera por s é liberado */

PROCEDURE WAIT (VAR s: SIGNAL);
   /* espera por algum outro processo enviar s */

PROCEDURE Awaited (s: SIGNAL): BOOLEAN;
   /* Awaited (s) = "no mínimo um processo espera por s" */

PROCEDURE Init (VAR s: SIGNAL);

END Process;
```

Se um processo é executado em quase-concorrência ou concorrência genuína, é dependente de implementação. A comunicação entre processos se dá de duas

[49] Cf. [Wirth 85].

maneiras distintas: *variáveis compartilhadas* e *sinais*. O acesso às variáveis compartilhadas deve ser feito de maneira exclusiva. Isso é feito através de um módulo *monitor*, que garante exclusão mútua de processos. Sinais são usados para sincronização entre processos. Um módulo é designado a ser um monitor pela especificação de uma *prioridade* no seu cabeçalho.[50]

Como as corrotinas são consideradas facilidades de baixo-nível, seus tipos e operadores devem ser importados do módulo SYSTEM ou de um outro módulo com facilidades de baixo nível. Há, em particular, o tipo ADDRESS e os procedimentos NEWPROCESS, TRANSFER e IOTRANSFER.

```
PROCEDURE NEWPROCESS (P: PROC; A: ADDRESS; n: CARDINAL
                      VAR P1: ADDRESS);
    /* Cria um novo processo com o procedimento P e uma área
    de trabalho de tamanho n e endereço inicial A é atribuído
    a P1. Esse processo é alocado mas não ativado. P deve ser
    um procedimento declarado no nível 0. */

PROCEDURE TRANSFER (VAR P1,P2: ADDRESS);
    /* Suspende o processo atual, atribui seu endereço a P1 e
    continua o processo designado por P2. A P2 deve ter sido
    atribuído um processo através de uma chamada anterior de
    NEWPROCESS ou TRANSFER. */

PROCEDURE IOTRANSFER (VAR P1,P2: ADDRESS; VA: ADDRESS);
    /* Análogo a TRANSFER, suspende a execução do processo de
    dispositivo, atribui-o a P1, continua a execução do processo
    interrompido P2 e, além disso, faz com que a interrupção
    ocorrida após a operação completada pelo dispositivo atri-
    bua o processo interrompido a P2 e continue a execução de
    P1. VA é o endereço de interrupção atribuído (associado) ao
    dispositivo. Considerada, originalmente, como dependente
    de implementação do PDP-11. */
```

Note-se que o módulo *Process* pode ser implementado usando-se as operações e tipos de dados do módulo SYSTEM.

[50] A prioridade é especificada através de um cardinal entre colchetes após o nome do módulo, na sua declaração.

5.5.2 Mesa

Mesa[51] provê mecanismos na linguagem para a execução concorrente de múltiplos processos, bem como facilidades para sincronização, por meio de monitores.

Os comandos FORK e JOIN permitem a execução paralela de dois procedimentos. Seu uso requer o tipo de dados PROCESS. FORK cria[52] um subprocesso de mesmo tipo que o processo ou o procedimento que o executou (chamado procedimento ou processo *raiz* do novo processo). Posteriormente, os resultados são recuperados pelo comando JOIN, que também apaga o subprocesso criado. Isto não deve ocorrer até que ambos os processos estejam prontos, isto é, tenham atingido os comandos JOIN e RETURN, respectivamente; este "rendezvous" é sincronizado automaticamente pelas facilidades de processos.

O mecanismo de sincronização entre processos é uma variante de monitor, implementado como uma instância de um módulo. A execução mútua é garantida através do mecanismo de *trava de monitor*.[53] A trava é um tipo predefinido (MONITORLOCK) e privado. Uma variável de condição c é de tipo predefinido CONDITION e seu conteúdo é privado. É sempre associada a alguma expressão lógica, que descreve um estado do monitor. O acesso ao seu conteúdo é feito através das operações WAIT, NOTIFY e BROADCAST.

Exemplo: Um processo esperando por uma condição faria:

```
WHILE ~Expressão_Lógica DO
    WAIT c
    ENDLOOP;
```

e um processo tornando uma condição verdadeira:

```
Torna Expressão_Lógica verdadeira
    -- isto é, como efeito lateral de modificar
    -- uma variável global
NOTIFY c;
```

Para notificar todos os processos que esperam uma variável de condição, deve-se usar:

```
BROADCAST c;
```

[51] Cf. [Mitchell et al. 79].
[52] Em inglês: *spawn*.
[53] Em inglês: *monitor lock*.

5.5.3 Ada

Uma tarefa é uma unidade de programa distinta de outras que pode ser executada em paralelo com outras tarefas. A comunicação e sincronização entre tarefas são feitas através do conceito de rendezvous entre uma tarefa que realiza uma chamada a uma entrada e outra tarefa que aceita a chamada, através do comando *accept*.

O comando

initiate *tarefa*

inicia a execução paralela de uma tarefa. Uma tarefa termina quando executa o último comando de seu corpo (antecedendo a palavra reservado **end** de seu final) ou através da execução do comando *abort*.

A parte visível de uma tarefa pode conter várias especificações de entradas. Externamente, uma entrada parece um procedimento, recebe parâmetros e é chamada da mesma maneira. No caso de procedimento, uma tarefa que o invoca executa o seu corpo imediatamente. No caso de uma entrada, é a tarefa onde foi declarada que a executa, não a que a invoca. Além disso, essas ações são executadas somente quando a tarefa está pronta para executar um comando *accept* correspondente.

A tarefa invocadora deve saber o nome da entrada e esta é específica da tarefa invocada. Esta, por outro lado, aceitará chamadas de qualquer tarefa. Assim, há a comunicação muitos-para-um; como conseqüência, cada entrada de uma tarefa tem, potencialmente, uma fila de tarefas que a invocam. Cada comando *accept* retira um item desta fila.

O comando *select* permite que uma tarefa aceite uma de várias alternativas de comandos *accept*. No caso mais geral, cada alternativa do comando select pode incluir uma *condição de guarda* seguindo a palavra reservada **when**. Todas as condições de guarda são avaliadas no início do comando select e somente as alternativas cujas condições resultam em verdadeiras é que são consideradas para seleção subseqüente. A alternativa cuja guarda é verdadeira (ou está ausente) é dita *aberta*; caso contrário, *fechada*.

Exemplo:[54]

```
task BUFFERING is
    entry READ (V: out ITEM);
```

[54] Cf. [Ichbiah et al. 79].

```
    entry WRITE (E: in ITEM);
end;

task body BUFFERING is
    SIZE        : constant INTEGER := 10;
    BUFFER      : array (1..SIZE) of ITEM;
    INX,OUTX    : INTEGER range 1..SIZE := 1;
    COUNT       : INTEGER range 0..SIZE := 0;
begin
    loop
        select
          when COUNT < SIZE =>
                accept WRITE (E: in ITEM) do
                        BUFFER (INX) := E;
                end;
                INX := INX mod SIZE + 1;
                COUNT := COUNT + 1;
        or
          when COUNT > 0 =>
                accept READ (V: out ITEM) do
                        V := BUFFER (OUTX);
                end;
                OUTX := OUTX mod SIZE + 1;
                COUNT := COUNT - 1;
        end select;
    end loop;
end BUFFERING;
```

As variáveis INX e OUTX são os índices extremos da parte correntemente utilizada de BUFFER e COUNT indica quantos itens estão nele armazenados. READ pode somente ser aceito quando BUFFER não está vazio (COUNT > 0) e WRITE somente pode ser aceito quando BUFFER não está cheio (COUNT < SIZE).

É interessante notar que as variáveis INX, OUTX, COUNT são atualizadas fora do rendezvous. Isso permite que a tarefa chamadora continue tão logo quanto possível.

O comando

delay *especificação_de_tempo*

adia a execução da tarefa por, no mínimo, o intervalo de tempo especificado (múltiplo de unidades básicas do relógio de tempo real – dependente de máquina), de tempo predefinido TIME.

Um comando *delay* pode ser usado no lugar de um comando *accept* fazendo a parte de sincronização de uma alternativa de um comando *select*. Tal comando pode ser usado para prover um tempo máximo de espera para o comando *select*. Após o estouro desse intervalo de tempo,[55] se não tiver ocorrido um rendezvous, o comando seguinte ao *delay* é executado. Se, ao contrário, o rendezvous ocorre antes do intervalo expirar, o comando *delay* é interrompido e a execução do *select* continua normalmente.

5.5.4 MC

MC oferece facilidades de multiprogramação através da utilização de processos. Um processo é uma entidade que apresenta parâmetros, declarações locais e um corpo que pode ser executado em paralelo com outros processos.

A declaração de um processo tem o seguinte formato:

process *identificador_com_atributos* '(' *lista_de_parâmetros_formais* ')' ';'
bloco identificador

O identificador que segue o símbolo **process** dá o nome ao processo podendo ser seguido de atributos (lista de expressões entre os símbolos '[' e ']') como, por exemplo, o endereço onde o processo deve residir. A utilização de atributos é um recurso de baixo nível que é permitida através da importação do identificador *SpecialAttributes*, do módulo LOW_LEVEL. Os atributos e sua forma de especificação são dependentes de implementação.

A ativação de um processo é feita pelo comando *start*, com a indicação dos seus atributos na ativação e com uma lista de parâmetros efetivos:

start *especificação_de_atributo identificador_qualificado*
(lista_de_parâmetros_efetivos)

Processos admitem apenas parâmetros por valor na parte fixa de parâmetros com as mesmas regras de passagem de parâmetros de subprogramas;[56] a parte de parâmetros variáveis só permite vetor de parâmetros de entrada.[57] O resultado dessa ativação é uma identificação de processo, de tipo predefinido PROCESSID. A

[55] Em inglês, esse estouro de tempo é chamado *time-out*.
[56] Cf. 5.3.
[57] Cf. 5.4.

falha na ativação de um processo levanta a exceção PROCESS_ACTIVATION_ERROR. O bloco de um processo não pode conter a declaração ou o cabeçalho de outros processos; subprogramas obedecem à mesma restrição.

A ativação de um processo implica na alocação de recursos necessários à sua execução (por exemplo, área de memória), conforme as possibilidades da implementação a ser utilizada. O término de um processo (quando o final de seu bloco é atingido ou quando é executado algum comando *return*) libera esses recursos e torna inválida a identificação do processo que lhe foi associada quando de sua ativação. A referência a um processo que não está mais ativo levanta a exceção NO_PROCESS_ID.

Se um processo é interrompido por alguma exceção que não pode ser tratada no seu bloco, este termina sem indicação do erro.

Seguindo a filosofia de N. Wirth, procurando desenvolver uma linguagem simples e devido ao fato de existirem vários mecanismos distintos de sincronização sendo, portanto, difícil de escolher um conjunto básico, simples, coerente e geral de primitivas, MC não se compromete com qualquer forma de implementação de processos. Estas facilidades podem ser criadas através dos recursos providos pelo módulo LOW_LEVEL.

Exemplo de subprogramas que poderiam ser definidos no módulo LOW_LEVEL para uma dada implementação:

```
type
    Message;
    /* esquema de sinalização entre processos: troca de mensagens */

procedure Send (PROCESSID; Message); exception (NO_PROCESS_ID);
    /* Envia a mensagem passada no segundo argumento para o pro-
    cesso cuja identificação é dada pelo primeiro argumento. */

procedure Receive (var PROCESSID; var Message);
    /* Recebe e devolve, no segundo parâmetro, uma mensagem pen-
    dente enviada pelo processo cuja identificação é devolvida no pri-
    meiro parâmetro. */

function Number_of_Messages (PROCESSID): CARDINAL;
        exception (NO_PROCESS_ID);
    /* Devolve o número de mensagens pendentes para o processo cuja
    identificação é passada como argumento. */
```

5.6 Compatibilidade entre Tipos

Há um consenso entre muitos usuários de linguagens que um esquema de tipos rígidos é uma característica desejável em linguagens de programação.[58] Numa linguagem com essa propriedade é possível determinar, com precisão, o tipo de cada expressão em um programa pois tanto variáveis quanto constantes têm tipos bem especificados em suas declarações.

Uma questão importante que surge é quando dois tipos são considerados equivalentes ou compatíveis num dado contexto.[59] A resposta a essa questão determina a utilização de valores de uma certa classe de tipos. Em geral, um contexto que exige um valor de um determinado tipo, aceitará qualquer valor de tipo equivalente ou compatível. Com a possibilidade de um programa poder definir novos tipos em termos de outros já existentes, a equivalência de tipos é não-trivial.

As regras de equivalência ou compatibilidade de tipos indicam a exata especificação de que mecanismos de verificação devem ser aplicados. [GhezJaz 87] define compatibilidade da seguinte maneira:

> Dois tipos, T_1 e T_2 são *compatíveis* se qualquer valor de tipo T_1 pode ser atribuído a uma variável de tipo T_2 e vice-versa e se parâmetros efetivos de tipo T_1 podem corresponder aos parâmetros formais de tipo T_2 e vice-versa.

Além disso, o texto supra-citado define duas noções de compatibilidade entre tipos: *equivalência por nome* e *equivalência estrutural*.[60] Considerem-se as seguintes declarações em Pascal:

```
type t = array [1..20] of integer;
var  a,b: array [1..20] of integer;
     c: t;
     d: record
          a: integer;
          b: t;
        end
```

Equivalência por Nome – Duas variáveis são de tipos compatíveis se os seus tipos têm o mesmo nome (predefinido da linguagem ou definido em um

[58] Cf. [BerSch 79,Popeck et al. 77].

[59] Vários autores usam ambos os termos de maneira intercambiável. Não será feita a distinção entre eles a não ser nas seções que lhes dão interpretação distinta.

[60] Cf. também [BerSch 79].

programa). Assim, nas declarações do exemplo, `c` e `d.b` são de tipos compatíveis mas não `a` e `b` ou `a` e `c`. O termo "equivalência por nome" reflete o fato que duas variáveis são de tipos compatíveis somente se os nomes de seus tipos são o mesmo. Alguns autores, no entanto, consideram que duas variáveis são de tipos compatíveis se aparecerem na mesma declaração. Assim, de acordo com essa definição, `a` e `b` são de tipos compatíveis.

Equivalência Estrutural – Duas variáveis são de tipos compatíveis se têm a mesma estrutura. De acordo com essa definição, os nomes de tipos definidos em um programa são usados apenas como uma abreviação (ou comentário) para a estrutura que representam e não para introduzir qualquer nova característica semântica. Para verificar a equivalência estrutural, nomes de tipos definidos em um programa são substituídos por suas definições. O processo é repetido até que não devam mais ocorrer substituições de nomes.[61] Então, os tipos são considerados estruturalmente equivalentes se têm, exatamente, a mesma descrição. No exemplo anterior, `a`, `b`, `c` e `d.b` têm tipos compatíveis.

5.6.1 Pascal

Pascal utiliza uma equivalência mista (em alguns casos, por nome e em outros, estrutural) de duas maneiras: *compatibilidade* e *compatibilidade para atribuição*.

Dois nomes de tipos denotam o *mesmo tipo* se e somente se eles derivam do mesmo identificador de tipo. Então, se T_1 é um nome de tipo,

$$\textbf{type } T_2 = T_1;\ T_3 = T_2;$$

define T_3 e T_2 como o mesmo tipo que T_1.

Dois tipos, T_1 e T_2,[62] são *compatíveis* se qualquer uma das condições a seguir é verdadeira:

- T_1 e T_2 são o mesmo tipo.
- T_1 é subintervalo de T_2 (ou vice-versa) ou ambos são subintervalos do mesmo tipo hospedeiro.[63]

[61] Esta definição de equivalência estrutural pode levar a uma repetição infinita de substituições quando apontadores são usados para criar definições de tipos recursivos. Linguagens que adotam essa formas de equivalência contornam esse problema através de regras ou algoritmos apropriados.

[62] Nesta e nas subseções seguintes, quando T_i é um nome de tipo, a frase "tipo T_i" indica o tipo associado com o nome T_i, como definido acima.

[63] Um tipo pode ser definido como um intervalo de qualquer tipo ordinal definido previamente, chamado seu *tipo hospedeiro* (*host type* ou *underlying type*).

- T_1 e T_2 são tipos conjuntos, seus tipos base[64] são compatíveis e ou ambos são compactados ou ambos não compactados.
- T_1 e T_2 são tipos *string*[65] com o mesmo número de componentes.

Um valor de tipo T_2 é *compatível para atribuição*, com o tipo T_1 se qualquer uma das condições a seguir é verdadeira:

- T_1 e T_2 são o mesmo tipo mas não um tipo arquivo (**file of...**) ou um tipo com componentes de tipo arquivo.
- T_1 é *Real* e T_2 é *Integer*.
- T_1 e T_2 são tipos ordinais compatíveis ou tipos conjuntos compatíveis e o valor é um membro do conjunto de valores determinado por T_1.
- T_1 e T_2 são tipos *string* compatíveis.

5.6.2 Algol 68

Algol 68 utiliza o esquema de equivalência estrutural. A determinação de equivalência estrutural necessita da expansão de definições de tipos recursivos em árvores potencialmente infinitas onde as arestas são marcadas por seletores e as folhas por tipos básicos. Árvores idênticas implicam em tipos equivalentes.[66] De acordo com a descrição de equivalência estrutural em Algol 68 em [GhezJaz 87], pode-se concluir que os seletores de registros não são considerados na determinação de equivalência entre registros.

5.6.3 Modula-2

A definição de Modula-2 [Wirth 85] não define precisamente qual mecanismo de equivalência é usado. As considerações feitas sobre compatibilidade são:

1. Variáveis declaradas numa mesma lista são todas de mesmo tipo.

[64] Um tipo conjunto define o *conjunto potência* (conjunto de todos os subconjuntos de valores possíveis desse tipo, incluindo o conjunto vazio) de um tipo ordinal, chamado o *tipo base* do conjunto.

[65] Um tipo vetor é chamado um tipo *string* se é compactado (**packed**), o tipo de seus componentes é o tipo predefinido *Char* e seu tipo de índice é um subintervalo de tipo predefinido *Integer*, de **1** a n, $n > 1$.

[66] Cf. [Horowitz 84,BerSch 79].

2. Na designação de uma variável indexada a, a expressão a[e] é válida se o tipo de índice de a é compatível para atribuição com o tipo de e.

3. Numa atribuição da forma v := e, o tipo de v deve ser compatível para atribuição com o tipo da expressão e.

4. Os tipos de um parâmetro efetivo e do formal correspondente devem ser idênticos no caso de parâmetros por variável ou compatíveis para atribuição no caso de parâmetros por valor.

Um tipo T_1 é *compatível* com um tipo T_0 se é declarado como $T_1 = T_0$ ou como um intervalo de T_0 ou, ainda, se T_0 e T_1 são ambos subintervalos do mesmo tipo base.

Tipos de operandos são *compatíveis para atribuição* se ou são compatíveis ou se ambos são dos tipos INTEGER ou CARDINAL ou subintervalos com tipos base INTEGER ou CARDINAL.

Extensões

[PedroJr 83] procura clarificar os conceitos de equivalência e compatibilidade. A primeira é considerada de maneira absoluta enquanto a segunda é relativa a algum contexto.

Devido à idiossincrasia dos processadores da família Intel 8086, no seu trabalho, [PedroJr 83] estende a linguagem Modula-2 com a introdução dos tipos predefinidos BYTE, SHORT_INTEGER, SHORT_CARDINAL; endereços e apontadores longos (LONG_ADDRESS e LONG_POINTER, respectivamente) e a retirada do tipo REAL.

A compatibilidade é definida em três classes:

1. Compatibilidade em operação.

2. Compatibilidade em atribuição.

3. Compatibilidade em passagem de parâmetro.

Define-se *equivalência de tipos* através de uma relação de equivalência "≡", dada por:

1. Tipos sinônimos são equivalentes (isto é, se existe uma declaração **type** $T_1 = T_2$, então $T_1 \equiv T_2$).

2. $T_1 \equiv T_2$ se T_1 e T_2 são intervalos de tipos base equivalentes, com os mesmos limites.

Define-se *compatibilidade de tipos* como uma relação reflexiva, simétrica mas não transitiva, da seguinte maneira: dois tipos, T_1 e T_2 são *compatíveis* se alguma das condições a seguir for satisfeita:

1. $T_1 \equiv T_2$.
2. T_1 e T_2 são subintervalos de um tipo T_3.
3. T_1 é um endereço longo (curto) e T_2 é apontador longo (curto) para algum tipo T_3.
4. T_1 é endereço curto e T_2 é o tipo CARDINAL.
5. T_1 e T_2 são apontadores longos (curtos) para tipos compatíveis.
6. T_1 e T_2 são conjuntos com tipos base equivalentes.
7. T_1 e T_2 são vetores dinâmicos de elementos compatíveis.
8. T_1 e T_2 são vetores estáticos de tipos de índices equivalentes e elementos compatíveis.
9. T_1 e T_2 são subprogramas de mesma categoria (procedimento ou função), parâmetros concordando em número, de tipos respectivamente compatíveis, tendo respectivamente os mesmos mecanismos de passagem e tipos equivalentes de resultado, se função.

O tipo de uma constante explícita tal como NIL, 20, etc. pode, a priori, pertencer a mais de um tipo (por exemplo, 20 pode ser de qualquer um dos tipos numéricos SHORT_INTEGER, SHORT_CARDINAL, INTEGER ou CARDINAL). A decisão sobre o tipo de tais constantes faz-se pelo contexto em que elas ocorrem em operações, chamadas de subprogramas e comandos de atribuição. Nos casos de uma definição da forma **const** c = 20, a discussão em [PedroJr 83] apresenta alguns casos especiais de sua implementação:

1. UNIV_INT ("Universal Integer") – atribuído a toda expressão numérica de tipo não especificado (através de funções de transferência de tipos) e de contexto não diferenciado.
2. UNIV_PTR ("Universal Pointer") – atribuído ao valor NIL.

5.6.4 C

Como definido por [KernRit 78], C não é uma linguagem com tipos rígidos como Pascal ou Algol 68. É relativamente permissiva em relação à conversão automática de expressões. Não se definem regras de compatibilidade entre tipos da linguagem. Uma versão separada do compilador, denominada *lint*, pode ser usada para fazer verificação de tipos e outras situações de inconsistência como, por exemplo, na passagem de argumentos de uma função.

A padronização de C[67] afirma que, em relação à compatibilidade, vários operadores podem converter, implicitamente, operandos de um tipo para outro e que uma constante inteira 0 pode ser convertida para um apontador de tipo apropriado, cujo valor é considerado como apontador nulo.

5.6.5 Ada

A compatibilidade, em Ada, é definida via equivalência por nome, da seguinte maneira:

1. Cada definição de tipo introduz um tipo distinto. Como conseqüência, cada nome de tipo denota um tipo distinto. Objetos com tipos distintos não podem ser intercambiados.

2. Em contraste, objetos pertencentes a diferentes subtipos de um mesmo tipo são compatíveis.

3. Tipos definidos como derivados de um outro tipo são chamados de *conformes* com ele. Conversões explícitas são possíveis entre tipos conformes.

5.6.6 MC

MC procura usar a filosofia de Modula-2 e Pascal em termos de flexibilidade nas regras de compatibilidade. No entanto, tais regras são rígidas, de maneira algo semelhante ao trabalho de [PedroJr 83].

MC diferencia as noções de equivalência e compatibilidade: equivalência é um conceito absoluto e a compatibilidade é relativa a alguma operação ou contexto.

Três tipos especiais, não passíveis de utilização por um programa, são acrescentados aos tipos predefinidos na linguagem para compatibilização de tipos inteiros, conjuntos e apontadores: UNIV_INTEGER, UNIV_SET e UNIV_POINTER, respectivamente. Toda expressão de MC tem um tipo associado, determinável em tempo de compilação.

[67] Cf. [DraftC 85].

A compatibilidade entre tipos é considerada em quatro situações em que os objetos da linguagem são confrontados:

1. Compatibilidade para operação.
2. Compatibilidade para atribuição.
3. Compatibilidade para passagem de parâmetros.
4. Compatibilidade para devolução de resultado de função.

Em algumas situações, expressões de MC produzem resultados cujos tipos são indefinidos. Um tipo indefinido nunca é compatível com qualquer outro tipo.

Tipos Anônimos

Para cada especificação de tipo sem um identificador, isto é, um nome de tipo, MC associa um nome interno como se fosse feita uma nova declaração, chamada *declaração de tipo anônimo*. Note-se que esta introduz um *novo* identificador de tipo anônimo.

Exemplo: A seqüência de declarações a seguir

```
type TipoInteiro = INTEGER;
     String = array [1..MAXSTR] of CHAR;
var i,j: INTEGER;
    k,l: TipoInteiro;
    s1: String;
    s2: array [1..MAXSTR] of CHAR;
```

é considerada como

```
type TipoInteiro = INTEGER;
type MC$TA0001 = array [1..MAXSTR] of CHAR;
     String = MC$TA0001;
var i,j: INTEGER;
var k,l: TipoInteiro;
var s1: String;
type MC$TA0002 = array [1..MAXSTR] of CHAR;
var s2: MC$TA0002;
```

Tipos Sinônimos e Tipos Equivalentes

Dois identificadores de tipo, T_1 e T_2, são *sinônimos* se:

- T_1 e T_2 são o mesmo identificador de tipo.
- T_2 é declarado como **type** $T_2 = T_1$.
- existe um tipo T' que é sinônimo de T_1 e também de T_2.

Dois tipos, T_1 e T_2, são *equivalentes* se:

- T_1 é sinônimo de T_2 e nenhum deles é sinônimo de um vetor com limites abertos.
- T_1 e T_2 são equivalentes a tipos subintervalos de tipo base[68] equivalentes, com os mesmos limites.
- T_1 e T_2 são equivalentes a tipos conjuntos com tipos base equivalentes.
- T_1 e T_2 são equivalentes a tipos vetores com limites definidos, com tipos de índices equivalentes e os tipos de componentes equivalentes.
- T_1 e T_2 são equivalentes a tipos apontadores para tipos equivalentes ou sinônimos.

Compatibilidade para Operação

Um operador n-ário define os tipos $T_1,\ldots,T_n$ de seus operandos. Alguns operadores podem ser *sobrecarregados*,[69] isto é, um mesmo símbolo (operador) pode denotar operações distintas, com uma seqüência de tipos de seus operandos para cada uma delas. Cada operador n-ário, aplicado a seus operandos, produz um valor de um certo tipo como resultado. As tabelas de operadores de MC, os tipos de seus operandos e o tipo de resultado são apresentados em [Carvalho 89a].

Para que uma operação seja semanticamente aceita em MC, os tipos de seus operandos, $T'_1,\ldots,T'_n$, devem ser *compatíveis para operação* com os tipos $T_1,\ldots,T_n$ aceitos pelos operadores, conforme descrito no referido manual.

T'_i é compatível para operação com T_i se:

[68] Um tipo T pode ser definido como um subintervalo de um tipo ordinal T' pela especificação de seus valores mínimo e máximo. O tipo T' especificado por esses limites é chamado de *tipo base* de T.

[69] Em inglês: *overloaded operators.*

- T_i e T'_i são equivalentes.
- T_i e T'_i são equivalentes a tipos subintervalos com o mesmo tipo raiz.[70]
- T_i é equivalente ao tipo ADDRESS, definido no módulo LOW_LEVEL, e T'_i é equivalente a um tipo apontador.

Compatibilidade para Atribuição

A atribuição de um valor de tipo T_e a uma variável de tipo T_v é possível se T_e é *compatível para atribuição* com T_v.

T_e é compatível para atribuição com T_v se:

- T_e é compatível para operação com T_v.
- T_e é equivalente ao tipo SHORTINT (SHORTCARD) e T_v é equivalente ao tipo INTEGER (CARDINAL).
- T_e é do tipo UNIV_POINTER e T_v é equivalente a um tipo apontador.
- T_e é do tipo UNIV_SET e T_v é equivalente a um tipo conjunto.

Note-se que um vetor com limites abertos não é compatível com qualquer tipo; portanto, não pode ser atribuído como uma unidade.

Compatibilidade para Passagem de Parâmetros

A parte fixa de parâmetros de um subprograma tem as regras de compatibilidade apresentadas a seguir. A parte de parâmetros variável já foi discutida em 5.4.

Parâmetro por Valor: Sejam T_f o tipo do parâmetro formal e T_e o tipo do parâmetro efetivo correspondente. O tipo T_e é *compatível para passagem de parâmetros por valor* com T_f se:

- T_e é compatível para atribuição com T_f.
- T_f é do tipo vetor com limites abertos, T_e é um tipo vetor (com limites definidos ou abertos) e T_e é conforme[71] com T_f.

[70] O *tipo base* de um conjunto ou subintervalo designa o tipo de seus elementos. Numa declaração **type** $T_* = T_o$, o *tipo raiz* de T_* é o tipo raiz de T_o. Se T_o for um tipo subintervalo, o tipo raiz de T_* é tipo base de T_o. Se T_* não tiver sido declarado como sinônimo de outro tipo, T_* será seu próprio tipo raiz.

[71] Cf. 5.2.3.

Parâmetro por Variável: Sejam T_f o tipo do parâmetro formal e T_e o tipo do parâmetro efetivo correspondente. O tipo T_e é *compatível para passagem de parâmetros por variável* com T_f se:

- T_e e T_f são equivalentes.
- T_e e T_f são equivalentes a tipos subintervalos com o mesmo tipo raiz.
- T_f é do tipo vetor com limites abertos, T_e é um tipo vetor (com limites definidos ou abertos) e T_e é conforme com T_f.

Compatibilidade para Devolução de Resultado de Função

Sejam T_f o tipo especificado no cabeçalho da função e T_r o tipo de um comando "**return** *expressão*" da parte de expressões de uma função (*expressão* não pode ser vazia).

T_r é *compatível para devolução de resultado de função* com T_f se T_r é compatível para atribuição com T_f.

Capítulo 6

Uma Implementação de MC

O principal objetivo de MC ao unir os aspectos positivos de Modula-2 e C é resultar numa linguagem que seja simples, com implementações também simples e eficientes.

Este capítulo apresenta um esboço de um sistema de execução[1] com tais características. Os principais aspectos discutidos são processos e exceções.

6.1 Sistema de Execução para Processos

Processos são unidades sintáticas de MC que são executadas em paralelo. A ativação de um processo aloca[2] uma região de memória para sua execução e ao seu término há a desalocação da memória anteriormente alocada. O programa principal é considerado como um processo especial, chamado *processo principal.*

A memória alocada para um processo em MC, não necessariamente contígua, consiste de duas regiões:

- Área de Código
- Área de Dados

A área de código contém o código executável dos blocos do processo e subprogramas nele encaixados sintaticamente[3] e a área de dados é utilizada para

[1] Em inglês: *run-time system.*

[2] *Alocar* (em inglês: *allocate*): atribuir determinado recurso ou espaço de memória para o uso na execução de um programa ou rotina específicos ou para o armazenamento de dados ou arquivos. *Desalocar* (em inglês: *deallocate*): liberar um recurso que já esteja alocado ou atribuído a uma tarefa específica. (*A. H. Fragomeni. Dicionário Enciclopédico de Informática. Prefácio de Antônio Houaiss. Ed. Campus/Livraria Nobel S/A*).

[3] Note-se que o processo principal inclui também os blocos dos processos nele declarados.

armazenar a identificação do processo (PROCESSID) e para a alocação de variáveis locais, parâmetros para invocação de subprogramas ou ativação de processos, e outras informações de controle como endereços de retorno de subprogramas. Esta área é manipulada como uma pilha a fim de armazenar os registros de ativação do processo.[4] A região da área de dados do processo principal onde são alocadas suas variáveis globais é chamada de *área de dados globais*. O gerenciamento de memória dinâmica de um processo não faz parte da definição da linguagem; subprogramas definidos nos módulos SYSTEM ou LOW_LEVEL podem ser usados para esse fim como NEW ou ALLOCATE_PROCESS_MEMORY.

Cada uma das regiões de memória é referenciada por um particular registrador: *registrador de código* (r_c), *registrador de identificação* (r_i), *registrador de dados* (r_d). Este último indica a área de dados da unidade sintática (isto é, módulo, processo ou subprograma) correntemente executada. A área de dados globais de um processo é também referenciada pelo *registrador de processo* (r_p) e a do processo principal também pelo *registrador de dados globais* (r_g). A figura 6.1 mostra a configuração de memória, na sua parte de dados, para um programa genérico em MC.

Estruturas auxiliares para o gerenciamento de processos não serão tratadas nesse texto.[5]

6.1.1 Criação de Processos

A criação de um processo é feita através do comando *start*. As ações mínimas que devem ser realizadas são:

- Alocação de memória para o processo, utilizando-se o procedimento ALLOCATE_PROCESS_MEMORY conhecido lexicamente no ponto de sua ativação.
- Criação do registro de ativação do processo na sua área de dados.[6]
- Devolução da identificação do processo (PROCESSID) como resultado do comando *start*.

Se um processo não puder ser criado, a unidade executável responsável pela sua ativação será interrompida, como resposta ao comando *start*, pelo levantamento da exceção PROCESS_ACTIVATION_ERROR.

[4] Cf. 6.2.

[5] Implementações mais comuns associam uma estrutura chamada *bloco de controle de processo*, BCP, a cada processo e seu gerenciamento é feito através do esquema de filas.

[6] Se a área de código do processo é uma cópia da sua definição no processo principal ou se é utilizada reentrância é dependente de implementação.

```
module m;
    ...
    process p;
        ...
        procedure a;
            ...
        begin
            ...
        end a;
        ...
    begin /* p */
    ... a; ...
    end p;
    ...
begin /* m */
    ...
    start p;
    ...
end m.
```

Figura 6.1: Configuração de memória (área de dados) na execução do procedimento *a*.

6.1.2 Finalização de Processos

Um processo termina quando o final de seu bloco é atingido ou através de um comando *return* ou, ainda, pela ocorrência de uma exceção a nível de bloco de processo (quer possa ser tratada ou não).

As ações mínimas que devem ser realizadas na finalização de um processo são:

- Liberação da memória alocada para o processo, utilizando-se o procedimento DEALLOCATE_PROCESS_MEMORY conhecido lexicamente no ponto de sua ativação.
- Remoção da identificação de processo (PROCESSID) associada.

Note-se que um processo principal que apresenta subprocessos não poderá ter sua área de dados liberada enquanto apresentar algum subprocesso ativo. Neste caso, deve-lhe ser associado um estado de *processo inativo*, que perdurará até a não existência de subprocessos ativos quando, então, será finalizado realmente.

6.1.3 Comunicação entre Processos

A comunicação entre processos é feita através das facilidades providas pelo módulo LOW_LEVEL. Implementações diferentes poderão apresentar recursos diversos para tal comunicação.

6.2 Registro de Ativação

Esta seção trata de registros de ativação para subprogramas. Um processo pode ser considerado como um procedimento especial declarado em nível 0.

As informações necessárias para a execução de um subprograma genérico são gerenciadas por um bloco de armazenamento chamado *registro de ativação*,[7] que consiste de uma coleção de campos como mostra a figura 6.2. Quando um subprograma é invocado, seu registro de ativação é empilhado na área de dados do processo onde foi declarado; quando o controle retorna à unidade invocadora, este registro é desempilhado.[8]

Os componentes de um registro de ativação são descritos a seguir:

Apontador Dinâmico: Essa região é usada para armazenar o endereço de registro de ativação da unidade invocadora. Seu conteúdo, no caso de processos, é dependente de implementação.

[7] Em inglês: *Activation record* ou *frame*.

[8] No caso de término de processos, além disso, outras ações devem ser realizadas; cf. 6.1.

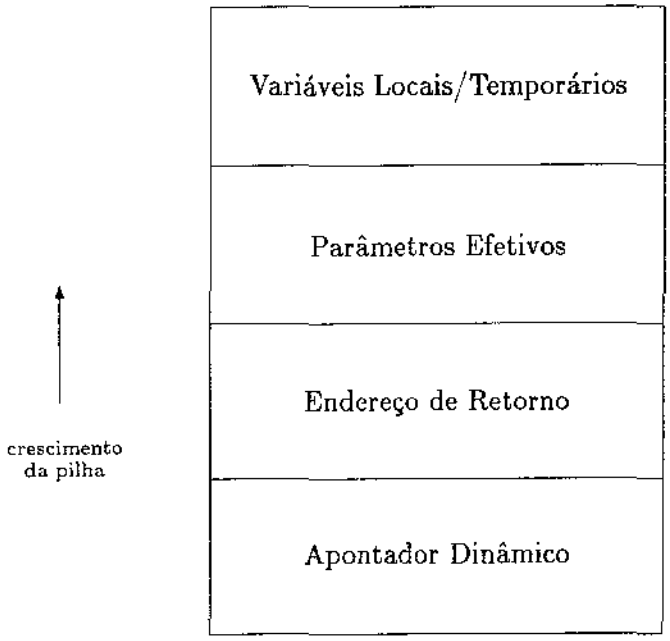

Figura 6.2: Registro de ativação de um subprograma genérico de MC

Endereço de Retorno: Essa região consiste de um registro com uma cópia do registrador de código no instante em que foi feita a invocação do subprograma.[9] Não aplicável a processos.

Parâmetros Efetivos: Essa região é usada para a unidade invocadora fornecer os parâmetros efetivos para o subprograma invocado. Se o subprograma for uma função, o valor a ser devolvido será considerado como um parâmetro a mais e alocado nessa região.[10]

Variáveis Locais: Essa região é usada para armazenar os dados que são locais à execução do subprograma bem como outros temporários.

Observação: Quando um vetor com limites abertos é passado como parâmetro, seus limites serão dispostos na região de parâmetros efetivos e seus componentes, no caso de parâmetros por valor, serão dispostos na região de variáveis locais (no caso, serão consideradas como valores temporários).

As figuras 6.3 e 6.4 trazem exemplos de registros de ativação e comportamento da área de dados, com chamadas encadeadas.

A coleção de apontadores dinâmicos apresentados nas referidas figuras, presentes na área de pilha de um processo em um determinado instante, é chamada de *cadeia dinâmica*.[11]

6.3 Memória Dinâmica

Um processo pode alocar variáveis de maneira dinâmica numa região chamada *memória dinâmica* ou *heap*. O esquema de gerenciamento desta área não está previsto na definição de MC.

Uma variável dinâmica pode ser criada por uma unidade executável através do procedimento predefinido NEW e ser desalocada através do procedimento predefinido DISPOSE. Procedimentos outros para realizar o gerenciamento do heap, alocando e desalocando variáveis dinâmicas, podem ser escritos utilizando-se as facilidades do módulo LOW_LEVEL.[12]

[9] Por exemplo, no microprocessador Intel 8086, armazenaria os valores dos registradores apontador de programa (PC) e de segmento de código (CS).

[10] Pode-se ter, numa dada implementação, os parâmetros passados em registradores de máquina por questões de eficiência.

[11] Em inglês: *dynamic chain*, também podendo ser chamada de *cadeia de chamada* (*call chain*).

[12] O procedimento NEW, do módulo SYSTEM, utiliza a função de alocação de memória ALLOCATE_MEMORY conhecida lexicamente no ponto de chamada; caso não haja a definição de tal função, será utilizada a que está declarado no módulo LOW_LEVEL. De maneira semelhante,

```
procedure Max (a: array [low..high] of INTEGER);
    var i,m: INTEGER;
begin
    m := a[low];
    for i := a.low+1 to a.high do
        if m > a[i] then m := a[i] end;
    end;
    Write (m); Writeln
end Max;
```

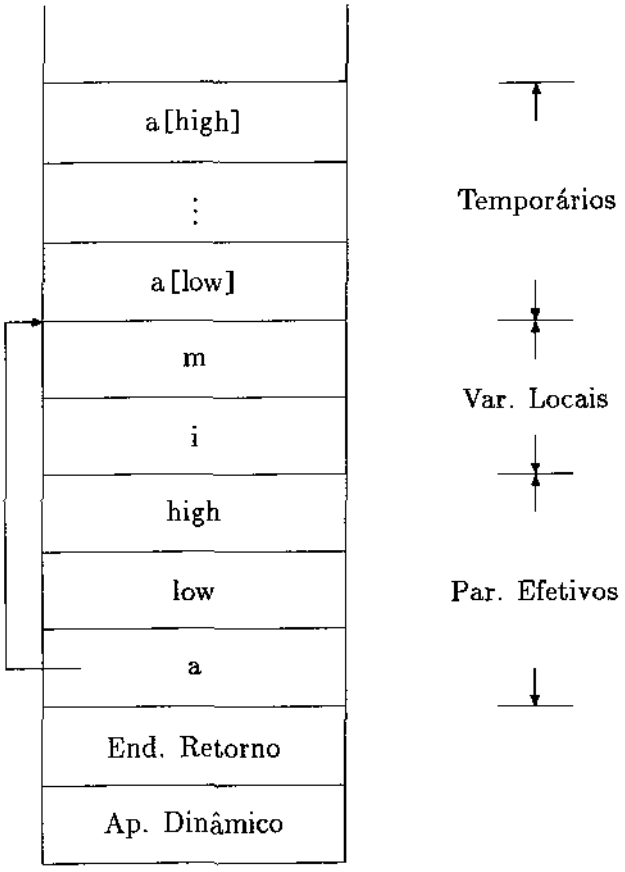

Figura 6.3: Procedimento *Max* e seu registro de ativação

```
module p;
    var z: REAL;

    procedure c (BOOLEAN);

    procedure a (x: INTEGER);
        var y: BOOLEAN;
    begin /* a */
        ...
        c (y);
        ...
    end a;

    procedure b (r: REAL);
        var s,t: INTEGER;
    begin /* b */
        ...
        a (s);
        ...
    end b;

    procedure c (q: BOOLEAN);
    begin /* c */
        ...
    end c;
    ...
begin /* p */
    ...
    b (z);
    ...
end p.
```

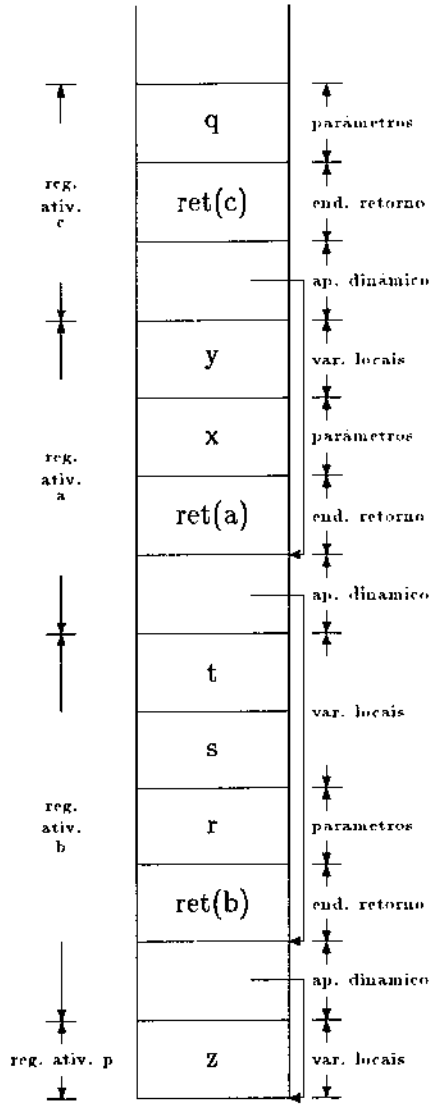

Figura 6.4: Módulo p e registros de ativação na chamada do procedimento c

6.4 Referências Não Locais

Como MC restringe a declaração de subprogramas, não permitindo que os mesmos sejam declarados de maneira encaixada sintaticamente, o esquema de *cadeia estática* ou *vetor de registradores de base*[13] pode ser simplificado através da utilização dos registradores r_g, r_p e r_d (global, processo e dados, respectivamente).

Módulos locais apresentam problemas de visibilidade apenas; a alocação de suas variáveis deve seguir à da unidade executável que os contém.

6.5 Sistema de Execução para Exceções

As exceções de um programa podem ser numeradas seqüencialmente, da seguinte maneira:

- Exceções predefinidas: $[0..e_s - 1]$, onde e_s é o número de exceções predefinidas de MC.
- Exceções definidas em um subprograma genérico:[14] $[e_n..e_n + e_b - 1]$, onde e_n é o número da última exceção que antecede o subprograma e e_b é o número de exceções do mesmo.

Assim, se se tratar do módulo principal, $e_n = e_s$. Em definições excaixadas, a numeração, em cada uma delas, inicia a partir do mesmo número e_n. Exemplo:

```
module p;
    exception p1, /* es */
              p2, /* es + 1 */
              p3; /* es + 2 */
    process r;
        exception r1; /* es + 3 */
        procedure a exception (p1,r1);
            /* p1,r1: já numerados */
            exception a1, /* es + 4 */
                      a2; /* es + 5 */
                      ...
        end a;
```

DISPOSE utiliza o procedimento DEALLOCATE_MEMORY conhecido lexicamente no ponto de chamada ou o que está definido no módulo LOW_LEVEL.

[13]Em inglês, esse vetor é chamado *display*.

[14]Procedimento, função ou processo.

```
        procedure b;
            exception b1; /* es + 4 */
                        ...
        end b;
        ...
    end r;
    procedure c exception (p1,p2);
        /* p1,p2: já numerados */
        exception c1, /* es + 3 */
                  c2; /* es + 4 */
        ...
    end c;
end p.
```

As exceções que o sistema pode levantar, interrompendo a execução do programa, são apenas as predefinidas e, funcionalmente, se comportam como se o comando

raise *exceção_predefinida*

tivesse sido executado.

O comando

raise *exceção*

causa o desvio para o tratador definido no bloco. O tratador pode ser considerado como sendo funcionalmente equivalente a uma versão particularizada do comando *case*:

```
case e of
        /* e: número (interno) de uma exceção */
    i: ...
    k: ...
    ...
    others ...
end
```

onde $i, k, \ldots$ são os números das exceções especificadas em cada *tratador_de_exceção* de um bloco. Pode-se definir SYSTEM$ERROR como o identificador de um tratador do sistema que causa o término da unidade executável

onde o subprograma se encontra e SYSTEM$PROPAGATE como o identificador de um tratador do sistema que causa o término da unidade executável e o relevantamento da mesma exceção na unidade invocadora.

Exemplo: Considere as duas versões do procedimento *BuscaSequencial* a seguir e o comando *case* relativo a parte de exceções, colocado após cada uma delas. INDEX_ERROR é o nome de uma exceção predefinida externamente ao procedimento.

```
/* Versão 1 */

function BuscaSequencial (var V: array [L..U: CARDINAL] of
                          INTEGER; x: INTEGER): CARDINAL;
    var i: CARDINAL;
begin
      i := V.L;
      while V[i] # x do i++ end;
      return i
exceptions
      when INDEX_ERROR do return V.L - 1 end
end BuscaSequencial;

/*
      Implementação do tratador:

      case Exceção$BuscaSequencial of
        INDEX_ERROR: return V.L - 1;
        others SYSTEM$ERROR
      end
*/

/* Versão 2 */

function BuscaSequencial (var V: array [L..U: CARDINAL] of
                          INTEGER; x: INTEGER): CARDINAL;
        exception (INDEX_ERROR);
    var i: CARDINAL;
begin
      i := V.L;
      while V[i] # x do i++ end;
```

```
        return i
end BuscaSequencial;

/*
        Implementação do tratador:

        case Exceção$BuscaSequencial of
          INDEX_ERROR: SYSTEM$PROPAGATE;
          others SYSTEM$ERROR
        end
*/
```

Capítulo 7

Conclusões

O objetivo deste capítulo é avaliar, de acordo com critérios de desenvolvimento de linguagens apresentados por alguns autores, a linguagem MC, bem como discutir algumas alternativas possíveis. Uma série de critérios foi considerada para seu projeto, alguns mais relevantes do que outros em função dos objetivos a serem atingidos; dentre eles podem-se citar os conceitos de tipos de dados, construções adequadas para programação estruturada, facilidades para acesso a recursos de baixo nível de maneira relativamente independente da máquina-objeto e o desenvolvimento de programas grandes de maneira modular.

Hoare[1] faz considerações interessantes para o projeto de uma linguagem:

> O projetista da linguagem deveria estar familiarizado com as várias características alternativas desenvolvidas por outros e deveria ter um excelente julgamento na escolha das melhores e rejeitar quaisquer que sejam mutuamente inconsistentes. Deve ser capaz de harmonizar, através de boa engenharia de projeto, quaisquer inconsistências ou justaposições menores entre características projetadas separadamente. Deve ter uma idéia clara de escopo, propósito, intervalo de aplicação de sua nova linguagem e a que ponto atingiria em tamanho e complexidade... Algo que não deveria ser feito é a inclusão de idéias suas que não foram testadas. Sua tarefa é consolidação, não inovação.

[1] C. A. R. Hoare, *Hints on Programming Language Design*, ACM SIGACT/SIGPLAN Conference on Principles of Programming Language. Oct. 1973; transcrito em [GhezJaz 87], página 333, tradução livre.

7.1 Critérios para Desenvolvimento

Segundo [GhezJaz 87] e [Horowitz 84], os principais critérios para um projeto de linguagem são:

- *Redigibilidade.*[2] Linguagens devem facilitar o desenvolvimento de programas oferecendo construções que tornam possível a implementação de um algoritmo de maneira simples, de modo que a preocupação esteja centrada no entendimento e solução do problema ao invés dos truques usados para implementá-la. As características que contribuem para essa capacitação são:

  - *Simplicidade.* Uma linguagem deve ser simples de dominar e todas as suas características diferentes devem ser facilmente lembradas; o efeito de qualquer combinação entre elas deve ser previsível e facilmente inteligível. A simplicidade é prejudicada se a linguagem provê várias formas alternativas para especificar o mesmo conceito (isso favorece a formação de "dialetos" que usam apenas subconjuntos da linguagem). Também é prejudicada se a linguagem permite conceitos semânticos diferentes a serem expressos pela mesma notação sintática.
  - *Expressividade.* A expressividade de uma linguagem é uma medida de quão naturalmente uma estratégia para a resolução de um problema pode ser mapeada em uma estrutura de programa. É função do tipo de problema que deve ser expresso.
  - *Ortogonalidade.* Ortogonalidade significa que qualquer composição das primitivas básicas da linguagem deve ser permitida, aumentando o seu grau de generalidade, sem quaisquer restrições ou casos especiais. No entanto, não deve ser considerada como uma substituta para a simplicidade, principalmente quando levar a soluções difíceis de serem entendidas e implementadas.
  - *Definibilidade.* Uma linguagem deve ter sua sintaxe e semântica definidas de maneira acurada. Omissões e definições vagas são ruins sob dois aspectos. Primeiramente, o programador não pode confiar completamente na linguagem e qualquer tentativa de encontrar uma resposta para uma definição pode resultar em mera frustração. Em segundo lugar, diferentes implementações podem escolher soluções diferentes para características ambíguas, com conseqüências desagradáveis para a portabilidade do programa.

[2] Em inglês: *writability.*

- *Legibilidade.*[3] A legibilidade de um programa é o fator principal a influenciar sua modificabilidade e manutenebilidade, diminuindo os custos de produção. Está relacionada à redigibilidade de programas. Em particular, abstração de dados e modularidade são os itens principais para atingir esse objetivo. Convenções léxicas (por exemplo, a disposição de comentários ou os caracteres de um identificador), sintáticas (delimitadores explícitos de comandos são preferíveis aos parênteses **begin** ... **end** ou similares, encontrados em várias linguagens) e semânticas (mecanismos de passagem de parâmetros) também exercem influência.

- *Confiabilidade.* Relacionada com redigibilidade e legibilidade de programas, pode ser aumentada se a linguagem faz distinção rigorosa entre as verificações estáticas e dinâmicas que devem ser feitas em programas. Quanto mais verificações estáticas houver, mais atraente será a linguagem porque verificações em tempo de execução diminuem sua velocidade. Há um relação profunda entre confiabilidade e rigorosa definição da semântica da linguagem.

- *Implementação.* De acordo com N. Wirth, "Na prática, uma linguagem de programação é tão boa quanto seu(s) compilador(es)".[4] Esse item envolve um compilador confiável, rápido e que produza um código-objeto eficiente. A interação com outras ferramentas do sistema deve ser facilitada.

7.2 Avaliação de MC

De acordo com os critérios para desenvolvimento da linguagem apresentados em 7.1, pode-se fazer a seguinte avaliação de MC:

- *Simplicidade*: MC é uma linguagem simples; suas estruturas são próximas às de Modula-2 e a possibilidade do uso de "dialetos" está restrita a alguns operadores (++, INCR, etc). O esquema de tratamento de exceções, importante característica introduzida, é de fácil interpretação. A impossibilidade de encaixamento sintático de subprogramas simplifica as regras de visibilidade de nomes.[5] Vetores com limites abertos são um recurso interessante

[3] Em inglês: *readability.*

[4] N. Wirth, *On the Design of Programming Languages*, Information Processing 74, Amsterdam, 1975; transcrito em [GhezJaz 87], página 333, tradução livre.

[5] Talvez haja uma incoerência neste ponto devido à existência de módulos locais em MC. Ainda assim, estes foram mantidos por não trazerem maiores complicações.

e não complicam a linguagem; mesmo procedimentos com número variável de parâmetros utilizam-se deles.

- *Expressividade*: MC, através do uso de módulos, tipos predefinidos e possibilitando a definição de novos tipos em um programa, estruturas de controle e processos, permite a resolução de problemas relacionados a várias áreas.

- *Ortogonalidade*: MC não é completamente ortogonal pois há muitas restrições semânticas para invalidar construções que foram permitidas na tentativa de simplificação da sintaxe.

- *Definibilidade*: Procura-se definir MC de maneira precisa. Um caso importante é a sua definição de compatibilidade entre tipos, que não é feita de maneira acurada na maioria das linguagens.

- *Legibilidade*: Alguns pontos são deixados à guisa de implementação como o tamanho de um identificador. Isso pode afetar a legibilidade de um programa. MC não apresenta delimitadores explícitos de comandos.

- *Confiabilidade*: A compatibilidade entre tipos precisamente definida colabora para esse item sendo possível, assim, realizar várias verificações em tempo de geração de código.

- *Implementação*: Nenhuma implementação de MC está disponível até o momento mas a definição da linguagem e o sistema de execução proposto sugerem implementações eficientes. Os módulos predefinidos SYSTEM e LOW_LEVEL são de particular interesse inclusive para se obter maior portabilidade.

7.3 Extensões

Esta seção traz sugestões para possíveis extensões da linguagem MC. Primeiramente são apresentadas algumas modificações incorporadas nas linguagens Oberon e Modula-3. A seguir são discutidos mecanismos sintáticos para a manipulação de apontadores para vetores em MC e a generalização das funções de conversão aplicadas a parâmetros efetivos. Finalmente, sugere-se um macroprocessador para a linguagem.

7.3.1 Modificações Incorporadas a Oberon e Modula-3

Oberon[6] e *Modula-3*[7] (não necessariamente ambas as linguagens) trazem algumas modificações em relação a Modula-2 que poderiam ser introduzidas em MC:

- Tipos de Dados
  - *Inclusão de Tipos*: permite o relaxamento das regras de compatibilidade entre tipos, da seguinte forma: um tipo T inclui ($\supset$) um tipo T' se os valores de T' também são valores de T. Por exemplo, postula-se, em Oberon, que:

    LONGREAL $\supset$ REAL $\supset$ LONGINT $\supset$ INTEGER $\supset$ SHORTINT

    As regras de atribuição devem ser relaxadas de acordo.
  - *Extensão de Tipos*: permite a construção de novos tipos baseados em outros já existentes e estabelece um certo grau de compatibilidade entre os novos e os antigos. Por exemplo:

    ```
    T  = RECORD x,y: INTEGER END
    T0 = RECORD (T) z: REAL END
    T1 = RECORD (T) w: LONGREAL END
    ```

    Assim, diz-se que T0 é uma *extensão* (*direta*) de T e T é o *tipo base* (*direto*) de T0. Da mesma forma, T1 é uma extensão (direta) de T. A regra de compatibilidade para atribuição estabelece que valores de um tipo estendido podem ser atribuídos a uma variável de seu tipo base T. No exemplo acima, a atribuição envolveria apenas os campos x e y.
    Para descrição completa dessa característica e agregadas, cf. [Wirth 88,Wirth 88a,Wirth 88b].
    Com essa extensão, registros com variantes poderiam ser retirados da linguagem.

- Módulos e Regras de Importação/Exportação
  - *Módulos Locais*: como raramente são utilizados, trazendo complicações nas regras de visibilidade na definição da linguagem e para os compiladores, podem ser eliminados.

[6] Cf. [Wirth 88].
[7] Cf. [Cardelli et al. 88].

- Comandos
  - *Comando* **with**: estendido para ser usado como *guarda* em tipos estendidos.

7.3.2 Apontadores para Vetores

Um dos recursos mais utilizados em C são os apontadores; parte por serem, às vezes, a única maneira de realizar certas operações (como a alteração de valores de parâmetros), parte por, normalmente, levarem a um código mais compacto e eficiente.[8]

Em C, há um forte relacionamento entre apontadores e vetores, de tal forma que ambos podem ser tratados de maneira similar. Qualquer operação que pode ser realizada através de indexação de vetores também pode ser feita com apontadores sendo esta versão, usualmente, mais rápida.

Assim, se um vetor a de inteiros for declarado como **int** a[10], os seguintes trechos de programa produzem resultados completamente equivalentes:

```
/* Versão com Indexação */

{
  int i;

  for (i=0; i<10; i++) printf("%d\n",a[i]);
}

/* Versão com Apontadores */

{
  int i; int *pa;

  for (i=0, pa=a; i<10; i++,pa++) printf("%d\n",*pa);
}
```

MC poderia prover semelhante facilidade com a introdução de um novo construtor de tipos predefinido, **arraypointer**, da seguinte maneira:

type PT = **arraypointer to** T;

[8] Cf. [KernRit 78].

O tipo T deve ser equivalente a um tipo vetor.

Uma declaração desse tipo apontador para vetor é funcional mas não sintaticamente equivalente à declaração de um registro da forma:

```
record
   Ptr: pointer to tipo_de_componente_de_T;
   Lower: ADDRESS;
      /* Endereço do primeiro elemento do vetor de tipo T */
   Upper: ADDRESS;
      /* Endereço do último elemento do vetor de tipo T */
end
```

O campo *Ptr* armazena o apontador para o elemento do vetor de tipo T; os campos *Lower* e *Upper* são usados para indicar a validade do apontador *Ptr*. A relação entre esses valores é:

$$\text{Lower} \leq \text{Ptr} \leq \text{Upper}$$

A não verificação dessa condição levanta a (nova) exceção predefinida PTR_BOUNDS_ERROR.[9]

A declaração de uma variável do tipo apontador para vetor introduz um registro dessa forma. A única operação de atribuição permitida é a do endereço de um vetor de tipo equivalente a T sendo dados, assim, os valores iniciais para os campos *Ptr, Lower, Upper* (Ptr = Lower, em particular).[10]

As operações que resultam em novos valores do apontador devem ser realizadas através das seguintes funções predefinidas:

`IncrPtr(p,i)` — Devolve o apontador para o elemento do vetor que se encontra i posições à frente do elemento apontado por p.

`DecrPtr(p,i)` — Devolve o apontador para o elemento do vetor que se encontra i posições antes do elemento apontado por p.

`NextPtr(p,i)` — Equivalente a `p := IncrPtr(p,i)`, atualizando o apontador p.

[9]Note-se que a verificação de condições excepcionais pode ser desativada através da especificação de atributos. Neste caso, poderiam ser omitidos os campos *Lower* e *Upper*, com uma implementação mais eficiente.

[10]A função predefinida ADDR pode ser usada para fornecer esse endereço.

`PrevPtr(p,i)` — Equivalente a `p := DecrPtr(p,i)`, atualizando o apontador p.

Observação: A omissão do parâmetro efetivo correspondente ao parâmetro i, em cada uma das funções acima, indica $i = 1$.

Assim, apresentam-se três trechos de programa em MC que produzem o mesmo resultado, de maneira semelhante à C:

```
type VetInt10 = array [1..10] of INTEGER;
var a: VetInt10;

/* Versão com Indexação */

var i: INTEGER;

for i := 1 to 10 do Write(a[i]) end

/* Versão com Apontadores */

var i: INTEGER;
    pa: arraypointer to VetInt10;

pa := ADDR(a);
for i := 1 to 10 do
    Write(pa↑);
    NextPtr(pa)
end

/* Versão com Apontadores e exceção PTR_BOUNDS_ERROR */

var pa: arraypointer to VetInt10;

pa := ADDR(a);
loop
    Write(pa↑);
    NextPtr(pa)
end
:
exceptions
```

when PTR_BOUNDS_ERROR **do return end**

7.3.3 Funções de Conversão Aplicadas a Parâmetros Efetivos

A seção 5.4.4 apresenta a maneira como MC possibilita a escrita de subprogramas que aceitam um número variável de parâmetros. O mecanismo adotado poderia ser generalizado, pois trata-se de dois conceitos distintos que foram acoplados numa única construção: vetor aberto (de parâmetros) e a aplicação de funções de conversão aos parâmetros efetivos na entrada do subprograma ou na saída do mesmo, conforme a classe do parâmetro.

Poderiam ser introduzidos mais dois mecanismos de passagem de parâmetros: *in* e *out*, com a mesma conotação de Ada.[11] A especificação de um parâmetro formal seria, então:

parâmetro_formal : *mecanismo_de_passagem lista_de_identificadores*
':' *tipo*
;
mecanismo_de_passagem : /* vazio */
| *mecanismo_de_passagem* **var**
| *mecanismo_de_passagem* **in**
'(' *lista_de_identificadores_qualificados* ')'
| *mecanismo_de_passagem* **out**
'(' *lista_de_identificadores_qualificados* ')'
;

lista_de_identificadores_qualificados denotaria as funções de conversão a serem aplicadas aos parâmetros efetivos na entrada do subprograma (mecanismo *in*), à sua saída (mecanismo *out*) ou na entrada e na saída (se os mecanismos *in* e *out* fossem especificados com suas respectivas listas de funções de conversão), seguindo regras semelhantes de compatibilidade e outras restrições apresentadas em 5.4.4.

Exemplo: função MaxVetNum que determina o máximo dentre um vetor de valores de tipo REAL:

```
function MaxVetNum (m: array [l..u: SHORTCARD] of REAL): REAL;
    var i: SHORTCARD := m.l + 1;
        max: REAL := m[m.l];
```

[11] Cf. 5.3.

```
begin
    loop
        if max < m[i]
          then max := m[i]
        end;
        ++i
    end;
exceptions
    when INDEX_ERROR do return max end
end MaxVetNum;
```

Essa função poderia ser chamada para determinar o máximo dentre listas de valores numéricos;[12] a seguir apresentam-se algumas delas, acompanhadas de declarações convenientes:

```
var
    b: array [1..10] of INTEGER;
    c: array [5..20] of REAL;
    d1,d2: SHORTCARD;
    d3: CARDINAL;
    d4,d5: REAL;

function MaxValNum (in (TOREAL) x: pararray [min..max: SHORTCARD]
                        of REAL): REAL;
begin
    return MaxVetNum (x)
end MaxValNum;

    :

    d5 := MaxVetNum (c);
    d4 := MaxValNum (d1,d2,d3) + MaxValNum (b[1],b[2],b[7], b[10]);
    d5 := MaxValNum (MaxVetNum (c), MaxValNum (d2,d3));
```

Outro exemplo é o procedimento `OrdemValNum`, que ordena uma série de valores de qualquer tipo numérico, cujo cabeçalho é:

[12] Constituem os tipos numéricos em MC os tipos inteiros predefinidos SHORTINT, SHORTCARD, INTEGER, CARDINAL, tipos subintervalos com o tipo raiz de tipo inteiro e o tipo predefinido REAL.

```
procedure OrdemValNum (in (TOREAL)
                       out (TO.HORTINT,TOSHORTCARD,TOINTEGER,TOCARDINAL)
    p: pararray [l..u: SHORTCARD]of REAL);
```

Utilizando as mesmas declarações do exemplo anterior, a seguinte chamada poderia ser feita:

```
OrdemValNum (d1,d3,d5);
        /* Parâmetro formal p no corpo de OrdemValNum:
                p[p.l] := TOREAL(d1);
                p[p.l+1] := TOREAL(d3);
                p[p.l+2 /* p.u */ ] := d5;
        Valores de d1,d3,d5 após o término de OrdemValNum:
                p[p.l] ≤ p[p.l+1] ≤ p[p.l+2]
                d1 := TOSHORTCARD(p[p.l]);
                d3 := TOCARDINAL(p[p.l+1]);
                d5 := p[p.l+2];
        */
```

Note-se que essa generalização não aumentaria essencialmente a linguagem e torná-la-ia mais ortogonal.

7.3.4 Macro-Processador

De maneira semelhante à existente na linguagem C, MC poderia prover extensões da linguagem através de um macro-processador com as seguintes funções principais:

- Definição/Expansão de Macros com ou sem parâmetros
- Compilação condicional
- Inclusão de arquivos

Note-se que a existência de um macro-processador pode ser ortogonal à definição da linguagem, não trazendo maiores conseqüências.

7.4 Considerações Finais

MC procura ser uma linguagem clara e concisa, com o objetivo principal de acrescentar as facilidades de baixo nível de C à maneira estruturada de Modula-2 de

forma a se obter a generalidade da primeira aliada à certa rigidez da segunda. Pode-se perguntar o porquê de tais objetivos, se seria válido colocar uma "camisa-de-força" na linguagem C com a perda de sua simplicidade no acesso aos recursos de baixo nível ou de manipulações não convencionais de variáveis, fortemente dependentes de implementação. A resposta a essa questão é que não se obtém uma linguagem mais restritiva; ao contrário, grande parte do que pode ser feito em C também o pode em MC, somente que de maneira mais ordenada, com indicações explícitas de utilização de recursos mais dependentes de implementação. É incentivada, portanto, a legibilidade que pode ser obtida no processo e o transporte facilitado entre diferentes implementações da linguagem. A contribuição de MC também pode ser considerada interessante em algumas propostas apresentadas; em particular, o esquema de vetores com limites abertos.

A nova linguagem Oberon é um parâmetro interessante para ajudar na análise de MC. Algumas de suas características já foram, inclusive, incorporadas; outras foram encontradas na definição de MC e solucionadas de maneira diferente a fim de se manter mais próximo de Modula-2.

Nenhuma implementação de um compilador para a linguagem está disponível até o momento por questões de finalidade de projeto. Sua gramática, no entanto, já foi submetida à análise do gerador de analisador sintático Yacc[13] com resultados bastante satisfatórios. É certo que o autor deste trabalho considera importante a implementação para verificação das viabilidades das soluções propostas. Espera-se uma evolução do projeto nesse sentido.

[13]Cf. [Johnson 75].

Apêndice A

Operadores de MC, Modula-2 e C

As tabelas deste apêndice trazem uma comparação entre os operadores das linguagens MC, Modula-2 e C, com seu respectivo significado.

Quando não houver o operador em uma linguagem, poderá ser considerado o nome de uma função equivalente, predefinida ou de biblioteca. Acompanhando o operador pode vir uma explicação sobre o mesmo, entre os símbolos '(' e ')'.

MC	Modula-2	C	Significado
↑ (designador)	↑ (designador)	* (unário)	indireção
ADDR (LOW_LEVEL)	ADR (SYSTEM)	& (unário)	endereço do argumento
+ (unário)	+ (unário)		identidade
− (unário)	− (unário)	− (unário)	negativo do argumento
		!	negação lógica
		~	complemento de um
++		++	incremento (prefixo ou posfixo)
−−		−−	decremento (prefixo ou posfixo)
SIZE (SYSTEM)	SIZE (predefinido)	**sizeof**	tamanho do argumento
+ (binário)	+ (binário)	+	adição
− (binário)	− (binário)	− (binário)	subtração
*	*	* (binário)	multiplicação
/	DIV	/	divisão inteira
/	/	/	divisão real
	MOD		módulo
%		%	resto
ASH (SYSTEM)		<<	deslocamento aritmético à esquerda
LSH (SYSTEM)		<<	deslocamento lógico à esquerda
ASH (SYSTEM)			deslocamento aritmético à direita
RSH (SYSTEM)		>>	deslocamento lógico à direita

Tabela A.1: Operadores e funções das linguagens MC, Modula-2 e C

MC	Modula-2	C	Significado
<	<	<	menor que
<=	<=	<=	menor que ou igual
>	>	>	maior que
>=	>=	>=	maior que ou igual
<=	<=		inclusão de conjuntos
>=	>=		inclusão de conjuntos
=	=	==	igual
#	#	!=	diferente
=	=		igualdade de conjuntos
#	#		diferença de conjuntos
in	IN		pertinência a conjuntos
notin			pertinência a conjuntos
+ (binário)	+ (binário)		união de conjuntos
− (binário)	− (binário)		diferença de conjuntos
*	*		intersecção de conjuntos
/	/		diferença simétrica de conjuntos
not	NOT		negação
and	AND	&&	operação lógica E
or	OR	||	operação lógica OU
xor			operação lógica OU-Exclusivo

Tabela A.2: Operadores e funções das linguagens MC, Modula-2 e C (continuação)

MC	Modula-2	C	Significado
		&	operação lógica E bit a bit
		|	operação lógica OU bit a bit
		^	operação lógica OU-Exclusivo bit a bit
if–then–else		? :	operação condicional
		=	atribuição
INC (SYSTEM)	INC (SYSTEM)	+=	atribuição com soma
DEC (SYSTEM)	DEC (SYSTEM)	-=	atribuição com subtração
		*=	atribuição com multiplicação
		/=	atribuição com divisão
		%=	atribuição com cálculo de resto
		>>=	atribuição com deslocamento aritmético à direita
		<<=	atribuição com deslocamento aritmético à esquerda
		&=	atribuição com operação lógica E
		^=	atribuição com operação lógica OU-Exclusivo
		|=	atribuição com operação lógica OU
		,	concatenação de expressões

Tabela A.3: Operadores e funções das linguagens MC, Modula-2 e C (continuação)

MC	Modula-2	C	Significado
&			concatenação de cadeias de caracteres
ABS (SYSTEM)	ABS (predefinido)		valor absoluto
CHR (SYSTEM)	CHR (predefinido)		representação ASCII do argumento
TOREAL (SYSTEM)	FLOAT (predefinido)	(**float**) (operador *cast*)	representação do argumento como real
EVEN (SYSTEM)			indicação lógica do argumento ser par
ODD (SYSTEM)	ODD (predefinido)		indicação lógica do argumento ser ímpar
ORD (SYSTEM)	ORD (predefinido)		número ordinal do argumento no conjunto definido pelo seu tipo
EXCL (SYSTEM)	EXCL (predefinido)		exclusão de elementos de conjuntos
INCL (SYSTEM)	INCL (predefinido)		inclusão de elementos em conjuntos
PRED (SYSTEM)			predecessor do argumento
SUCC (SYSTEM)			sucessor do argumento
TRUNC (SYSTEM)	TRUNC (predefinido)		truncamento de reais
CEIL (SYSTEM)			menor inteiro maior ou igual ao argumento
FLOOR (SYSTEM)			maior inteiro menor ou igual ao argumento

Tabela A.4: Operadores e funções das linguagens MC, Modula-2 e C (continuação)

Apêndice B

Exemplos de Módulos em MC

Este apêndice apresenta três exemplos de módulos em MC, a saber:

- Manipulação de matrizes de duas dimensões: procedimentos para soma e subtração de matrizes; multiplicação de matriz por escalar; multiplicação de matriz por um vetor; multiplicação de duas matrizes.
- Gerador de referências cruzadas: versão do gerador para Modula-2, apresentado em [Wirth 85], que lê um arquivo com um programa fonte em Modula-2 e gera um arquivo com uma listagem do programa com numeração de linhas e uma tabela de todos os identificadores em ordem lexicográfica,[1] cada um deles seguido por uma lista dos números das linhas onde ocorrem no programa.
- Produtor/Consumidor: uma coleção de processos gera tarefas a serem realizadas por uma coleção de processos consumidores. Os processos são executados de maneira concorrente com velocidades independentes entre si. O sincronismo é feito através de semáforos. As tarefas são descritas como mensagens (tipo importado do módulo LOW_LEVEL). Processos produtores colocam mensagens em uma "caixa de correio"[2] e processos consumidores retiram mensagens dele. Apresentado em [HaberPerry 83].

[1] Uma extensão de ordem alfabética, segundo a ordem dos caracteres da tabela ASCII.
[2] Em inglês: *mailbox*.

B.1 Manipulação de Matrizes de Duas Dimensões

```
definition module Manipula_Matr_2D;

    /* Manipulação de matrizes de duas dimensões com tipos
       de componentes REAL */

type
    Matr_2D = array [MinLin..MaxLin: INTEGER;
                     MinCol..MaxCol: INTEGER] of REAL;
type
    Vetor_1D = array [Min..Max: INTEGER] of REAL;

exception DimIncompativeis;


procedure Soma_Matr_2D (Matr_2D; Matr_2D; var Matr_2D);
                exception (DimIncompativeis);

procedure Sub_Matr_2D (Matr_2D; Matr_2D; var Matr_2D);
                exception (DimIncompativeis);

procedure Mult_Escalar_Matr_2D (REAL; Matr_2D; var Matr_2D);
                exception (DimIncompativeis);

procedure Mult_Vetor_Matr_2D (Vetor_1D; Matr_2D; var Vetor_1D);
                exception (DimIncompativeis);

procedure Mult_Matr_2D (Matr_2D; Matr_2D; var Matr_2D);
                exception (DimIncompativeis);

end Manipula_Matr_2D.

implementation module Manipula_Matr_2D;

    /* Rotinas para manipulação de matrizes de duas dimensões
       com tipos de componentes REAL */

function Lim_Diferentes (l1,u1,l2,u2: INTEGER): BOOLEAN;
```

```
                /* x: array [l1..u1]; y: array [l2..u2] */
begin
    return (l1 # l2) or (u1 # u2)
end Lim_Diferentes;

procedure Soma_Matr_2D (x,y: Matr_2D; var z: Matr_2D);
            exception (DimIncompativeis);

                    /* z_{p×q} = x_{p×q} + y_{p×q} */
var
   i,j: INTEGER;

begin
    /* verifica se as dimensões são compatíveis */
    if Lim_Diferentes (x.MinLin,x.MaxLin,y.MinLin,y.MaxLin)
          or Lim_Diferentes (x.MinLin,x.MaxLin,z.MinLin,z.MaxLin)
          or Lim_Diferentes (x.MinCol,x.MaxCol,y.MinCol,y.MaxCol)
          or Lim_Diferentes (x.MinCol,x.MaxCol,z.MinCol,z.MaxCol)
      then raise DimIncompativeis
    end;
    /* soma propriamente dita */
    for i := z.MinLin to z.MaxLin do
        for j := z.MinCol to z.MaxCol do
            z[i,j] := x[i,j] + y[i,j];
        end
    end
end Soma_Matr_2D;

procedure Sub_Matr_2D (x,y: Matr_2D; var z: Matr_2D);
            exception (DimIncompativeis);

                    /* z_{p×q} = x_{p×q} - y_{p×q} */

begin
    Mult_Escalar_Matr_2D (-1.0,y,y);
    Soma_Matr_2D (x,y,z)
end Sub_Matr_2D;
```

```
procedure Mult_Escalar_Matr_2D (s: REAL; x: Matr_2D; var z: Matr_2D);
            exception (DimIncompativeis);

                    /* z_{p×q} = s * x_{p×q} */
var
   i,j: INTEGER;

begin
    /* verifica se as dimensões são compatíveis */
    if Lim_Diferentes (x.MinLin,x.MaxLin,z.MinLin,z.MaxLin)
          or Lim_Diferentes (x.MinCol,x.MaxCol,z.MinCol,z.MaxCol)
      then raise DimIncompativeis
    end;
    /* multiplicação propriamente dita */
    for i := z.MinLin to z.MaxLin do
        for j := z.MinCol to z.MaxCol do
            z[i,j] := s * x[i,j];
        end
    end
end Mult_Escalar_Matr_2D;

procedure Mult_Vetor_Matr_2D (v: Vetor_1D; x: Matr_2D;
                                    var z: Vetor_1D);
            exception (DimIncompativeis);

                    /* z_{1×p} = v_{1×q} + x_{q×p} */
var
   i,j: INTEGER;
   s:REAL;

begin
    /* verifica se as dimensões são compatíveis */
    if Lim_Diferentes (1,v.Max,x.MinLin,x.MaxLin)
          or Lim_Diferentes (1,z.Max,x.MinCol,x.MaxCol)
      then raise DimIncompativeis
    end;
    /* multiplicação propriamente dita */
    for i := 1 to z.Max do
        s := 0.0;
```

```
        for j := 1 to v.Max do
            s := s + v[j] * x[j,i];
        end;
        z[i] := s
    end
end Mult_Vetor_Matr_2D;

procedure Mult_Matr_2D (x,y: Matr_2D; var z: Matr_2D);
            exception (DimIncompativeis);

                /* z_{p×r} = x_{p×q} + y_{q×r} */
var
   i,j,k: INTEGER;
   s: REAL;

begin
    /* verifica se as dimensões são compatíveis */
    if Lim_Diferentes (x.MinLin,x.MaxLin,z.MinLin,z.MaxLin)
          or Lim_Diferentes (x.MinCol,x.MaxCol,y.MinLin,y.MaxLin)
          or Lim_Diferentes (z.MinCol,z.MaxCol,y.MinCol,y.MaxCol)
      then raise DimIncompativeis
    end;
    /* multiplicação propriamente dita */
    for i := z.MinLin to z.MaxLin do
        for j := z.MinCol to z.MaxCol do
            s := 0.0;
            for k := x.MinCol to x.MaxCol do
                s := s + x[i,k] * y[k,j]
            end;
            z[i,j] := s
        end
    end
end Mult_Matr_2D;


end Manipula_Matr_2D.
```

B.2 Gerador de Referências Cruzadas

```
definition module TableHandler;

const
      LineLength = 80;
      WordLength = 24;

type Table;
      overflow_type = (none, over_alloc_asc, over_alloc_Word,
                       over_alloc_Item);

var overflow: overflow_type;

procedure InitTable (var Table);

procedure Insert (Table; var array [: SHORTCARD] of CHAR; INTEGER);

procedure Tabulate (Table);

end TableHandler.

implementation module TableHandler;

from InOut import Write, Writeln;

const TableLength = 3000;

type
     TreePtr = pointer to Word;
     ListPtr = pointer to Item;
     Item = record
                num: INTEGER;
                next: ListPtr
            end;
     Word = record
                 key: CARDINAL;
                 first: ListPtr;
                 left,right: TreePtr
```

```
            end;
    Table = TreePtr;

var
    id: array [0..WordLength] of CHAR;
    ascinx: CARDINAL := 0;
    asc: array [0..TableLength-1] of CHAR;


procedure InitTable (var t: Table);
begin
    New (t);
    t↑.right := NIL
end InitTable;

type Relation = (less,equal,greater);

function rel (CARDINAL): Relation;

function Search (p: TreePtr): TreePtr;
    var
        q: TreePtr := p↑.right;
        r: Relation := greater;
        i: SHORTCARD := 0;
begin
    while q # NIL do
        p := q;
        r := rel (p↑.key);
        if r = equal
            then return p
            else q := if r = less then p↑.left else p↑.right end
        end
    end;
    New(q);
    with q↑ do
        key := ascinx;
        first := left := right := NIL
    end;
    if r = less
```

```
            then p↑.left := q
            else p↑.right := q
        end;
        while id[i] > ' ' do
            asc[ascinx++] = id[i++];
        end;
        asc[ascinx++] := ' ';
        return q;
    exceptions
        when STORAGE_OVERFLOW do return NIL end;
        when INDEX_ERROR do overflow := over_alloc_asc; return q end
    end Search;

    function rel (k: CARDINAL): Relation;
        var
            i: SHORTCARD := 0;
            R: Relation := equal;
            x,y: CHAR;
    begin
        loop
            x := id[i++]; y := asc[k++];
            if x <= ' ' then return R end;
            if x < y then R := less
                elsif x > y then R := greater
            end
        end;
        return (if x > y then greater else less end)
    end rel;

    procedure Insert (t: Table; var x: array [lx..ux: SHORTCARD] of CHAR;
                      n: INTEGER);
        var
            p: TreePtr;
            q: ListPtr;
            i: SHORTCARD := x.lx;
    begin
        repeat
            id[i] := x[i++]
        until id[i-1]= ' ' or i = WordLength;
```

```
    p := Search (t);
    if p = NIL then overflow := over_alloc_Word
    else
        New (q);
        q↑.num := n;
        q↑.next := p↑.first;
        p↑.first := q
    end
exceptions
    when STORAGE_OVERFLOW do overflow := over_alloc_Item end
end Insert;

procedure PrintItem (p: TreePtr);
    const
        L = 6;
        N = (LineWidth - WordLength) / L;
    var
        ch: CHAR;
        i: SHORTCARD := WordLength + 1;
        k: CARDINAL := p↑.key;
        q: ListPtr := p↑.first;
begin
    repeat
        ch := asc[k];
        --i; ++k;
    until ch <= ' ';
    while i-- > 0 do Write (' ') end;
    i := N;
    while q # NIL do
        if i = 0 then
            Writeln ( );
            i := WordLength + 1;
            repeat
                Write (' ')
            until --i = 0;
            i := N
        end;
        Write (q↑.num);
        q := q↑.next; --i
```

```
    end;
    Writeln ( )
end PrintItem;

procedure TraverseTree (p: TreePtr);
begin
    TraverseTree (p↑.left);
    PrintItem (p);
    TraverseTree (p↑.right);
exceptions
    when ACCESS_ERROR do return end
end TraverseTree;

procedure Tabulate (t: Table);
begin
    Writeln ( ); TraverseTree (t↑.right)
end Tabulate;


begin /* TableHander */
    id[WordLength] := ' ';
    overflow := none
end TableHandler.


module XREF;

from InOut import
        Open, Close, input, output, Open_Input_Error, Open_Output_Error,
        EOL, EOF, Read, Write;
from TableHandler import
        WordLength, Table, overflow_type, overflow, InitTable, Insert, Tabulate;

type Alfa = array [0..9] of CHAR;

exception End_of_File;

const N = 45;
```

```
var
    ch: CHAR;
    i,k,l,m,r: SHORTCARD;
    lno: INTEGER := 0;
    T: Table;
    id: array [0..WordLength-1] of CHAR;
    key: array [1..N] of Alfa :=
        ("AND       ","ARRAY     ","BEGIN     ",
        "BITSET     ","BOOLEAN   ","BY        ",
        "CASE       ","CARDINAL  ","CHAR      ",
        "CONST      ","DIV       ","DO        ",
        "ELSE       ","ELSIF     ","END       ",
        "EXIT       ","EXPORT    ","FALSE     ",
        "FOR        ","FROM      ","IF        ",
        "IMPORT     ","IN        ","INTEGER   ",
        "LOOP       ","MOD       ","MODULE    ",
        "NOT        ","OF        ","OR        ",
        "POINTER    ","PROCEDURE ","QUALIFIED ",
        "RECORD     ","REPEAT    ","RETURN    ",
        "SET        ","THEN      ","TO        ",
        "TRUE       ","TYPE      ","UNTIL     ",
        "VAR        ","WHILE     ","WITH      ");

procedure copy;
begin
    Write (ch); Read (ch)
end copy;

procedure heading;
begin
    Write (lno++,' ')
end heading;



begin /* XREF */
    InitTable (T);
    Open (input,"MOD");
    Open (output,"XREF");
```

```
Read (ch);
heading;
repeat
    case ch of
        'A'..'Z' | 'a'..'z':
            k := 0;
            repeat
                id[k++] := ch; copy
            until ch notin {'0'..'9','A'..'Z','a'..'z'};
            l := 1; r := N; id[k] := ' ';
            repeat
                m := (l + r) / 2; i := 0;
                while (id[i] = key[m,i]) and (id[i] > ' ') do i++ end;
                if id[i] <= key[m,i] then r := m - 1
                elsif id[i] >= key[m,i] then l := m + 1 end
            until l > r;
            if l = r + 1 then Insert (T,id,lno) end
        '0'..'9':
            repeat
                copy
            until ch notin {'0'..'9'}
        '(':
            copy;
            if ch = '*'
                repeat
                    repeat
                        if ch = EOL
                            then copy; heading;
                            else copy
                        end
                    until ch = '*';
                    copy
                until ch = ')';
                copy
            end
        '\'':
            repeat
                copy
            until ch = '\';
```

```
                copy
            '\"':
                repeat
                    copy
                until ch = '\"';
                copy
            EOL:
                copy;
                heading
            EOF:
                raise End_of_File
            others copy
        end
    until overflow # none;
    /* overflow */
    Write ("Table overflow");
    Writeln ( );
    raise End_of_File;
    return
exceptions
    when Open_Input_Error do
        Write ("Can't open input file") end
    when Open_Output_Error do
        Write ("Can't open output file") end
    when End_of_File do
        Tabulate (T);
        Close (input,"MOD"); Close (output,"XREF")
    end
end XREF.
```

B.3 Produtor/Consumidor

```
definition module Semaphore_Manager;

    from LOW_LEVEL import Message;

    type Semaphore;
```

```
procedure P (var Semaphore);

procedure V (var Semaphore);

procedure InitSemaphore (var Semaphore; INTEGER);

procedure Copy (Message; var Message);

end Semaphore_Manager.

definition module Producer_Consumer;

    from LOW_LEVEL import Message;

    type mailbox;

    const numbox = 10;

    var Mail: array [1..numbox] of mailbox;

procedure Deposit (Message; var mailbox);

procedure Remove (var Message; var mailbox);

end Producer_Consumer.

implementation module Producer_Consumer;

    from Semaphore_Manager import
                    Semaphore,InitSemaphore,P,V,Copy;


    const capacity = 24;

    type
        slotindex = SHORTCARD [0..capacity - 1];
        mailbox = record
```

```
                    head, tail: slotindex;
                    dsem, rsem: Semaphore;
                    mesnum, slotnum: Semaphore;
                    box: array slotindex of Message
                end;

    var
        i: SHORTCARD;

procedure Deposit (m: Message; var mb: mailbox);
begin
    with mb do
        P (dsem);
        P (slotnum);
        Copy (m,box[head]);
        head := (head + 1) % capacity;
        V (mesnum);
        V (dsem)
    end
end Deposit;

procedure Remove (var m: Message; var mb: mailbox);
begin
    with mb do
        P (rsem);
        P (mesnum);
        Copy (box[head],m);
        tail := (tail + 1) % capacity;
        V (slotnum);
        V (rsem)
    end
end Remove;


begin /* Producer_Consumer */
    for i := 1 to numbox do
        with Mail[i] do
            head := tail := 0;
            InitSemaphore (dsem,1);
```

```
            InitSemaphore (rsem,1);
            InitSemaphore (mesnum,0);
            InitSemaphore (slotnum,capacity)
        end
    end
end Producer_Consumer.
```

Bibliografia

[AhoSetUll 86] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers — Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[Anderson 80] B. Anderson. *Type Syntax in the Language "C" — an object lesson in syntactic innovation.* SIGPLAN Notices, vol. 15 #3, March 1980. Pg. 21–27.

[Anderson 86] T. L. Anderson. *The Scope of Imported Identifiers in Modula-2.* SIGPLAN Notices, vol. 21 #9, September 1986. Pg. 17–21.

[Barnes 80] J. G. P. Barnes. *An Overview of Ada.* Software–Practice and Experience, vol. 10, 1980. Pg. 851–887.

[BerSch 79] D. M. Berry, R. L. Schwartz. *Type Equivalence in Strongly Typed Languages: One More Look.* SIGPLAN Notices, vol. 14 #9, September 1979. Pg. 35–41.

[Bielak 85] R. Bielak. *Ada vs. Modula-2: A View from the Trenches.* SIGPLAN Notices, vol. 20 #12, December 1985. Pg. 13–17.

[Bondy 87] J. Bondy. *Uninitialized Modula-2 Abstract Object Instances, Yet Again.* SIGPLAN Notices, vol. 22 #5, May 1987. Pg. 58–63.

[Cardelli et al. 88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, G. Nelson. *Modula-3 Report.* Digital Research Center. August 1988.

[Carvalho 89a] C. S. R. Carvalho. *Linguagem MC — Manual de Referência.* DCC—IMECC—UNICAMP. Em preparação.

[Christ. et al. 86] H. P. Christiansen, A. Lacroix, K. Lundberg, A. McKeeman, P. V. D. van der Stock. *More or Less Concerns About Modula-2.* SIGPLAN Notices, vol. 21 #9, September 1986. Pg. 27–31.

[Coar 84] D. Coar. *Pascal, Ada, and Modula-2.* Byte, August 1984. Pg. 215–232. Reprinted in E. Horowitz, *Programming Languages: A Grand Tour.* Computer Science Press. Pg. 291–298.

[Cooper 83] D. Cooper. *Standard Pascal - User Reference Manual.* W. W. Norton & Company, 1983.

[Cornelius 88] B. J. Cornelius. *Problems with the Language Modula-2.* Software–Practice and Experience, vol. 18 (6), 1988. Pg. 529–543.

[Crawford 85] A. L. Crawford. *High Level Input/Output in Modula-2.* SIGPLAN Notices, vol. 20 #12, December 1985. Pg. 18–25.

[CzyzIgl 85] J. Czyzowicz, M. Iglewski. *Implementing Generic Types in Modula-2.* SIGPLAN Notices, vol. 20 #12, December 1985. Pg. 26–32.

[DecVaxPascal 85] Digital Equipment Corporation. *Programming in VAX Pascal.* Order Number: AA-L369B-TE. March 1985.

[DraftAda 83] United States Department of Defense. *Ada Programming Language.* Draft ANSI/MIL-STD 1815A. January 1983.

[DraftC 85] ANSI (American National Standards Institute) J. J. Brodie, T. Plum, P. J. Plauger, L. Rosler, R. A. Phraner. *C Information Bulletin.* Draft, Ref. Doc.: X3J11/85-008. April 1985.

[Feuer 82] A. R. Feuer. *The C Puzzle Book.* Prentice-Hall, 1982.

[FitzJohn 81] P. A. Fitzhorn, G. R. Johnson. *C: Toward a Concise Syntactic Description.* SIGPLAN Notices, vol. 16 #12, December 1981. Pg. 14–21.

[GhezJaz 87] C. Ghezzi, M. Jazayeri. *Programming Language Concepts. 2/E.* John Wiley & Sons, 1987.

[Goldberg 85] M. Goldberg. *A response to "Some Concerns About Modula-2".* SIGPLAN Notices, vol. 20 #8, August 1985. Pg. 71–72.

[Greenwood 86] J. R. Greenwood. *Comments on "A View from the Trenches": Ada vs. Modula-2 vs. Praxis.* SIGPLAN Notices, vol. 21 #5, May 1986. Pg. 45–49.

[Habermann 73] A. N. Habermann. *Critical Comments on the Programming Language Pascal.* Acta Informatica 3, 1973. Pg. 47–57.

[HaberPerry 83] A. N. Habermann, D. E. Perry. *Ada for Experienced Programmer's.* Addison-Wesley, 1983.

[HanKrie 85] L. Hancock, M. Krieger. *Manual de Linguagem C.* Editora Campus, 1985.

[Hansen 75] P. B. Hansen. *The Programming Language Concurrent Pascal.* IEEE Transactions on Software Engineering SE-1, 2. June 1975. Pg. 199–207.

[HarbSteele 84] S. P. Harbison, G. L. Steele Jr. *C: A Reference Manual.* Prentice-Hall, 1984.

[Hayward 86] V. Hayward. *Compared Anatomy of the Programming Languages Pascal and C.* SIGPLAN Notices, vol. 21 #5, May 1986. Pg. 50–60.

[HoltCordy 88] R. C. Holt, J. R. Cordy. *The Turing Programming Language.* Communications of the ACM, vol. 31 #12. December 1988. Pg. 1410–1423.

[Horowitz 84] E. Horowitz. *Fundamentals of Programming Languages.* Second Edition. Springer-Verlag, 1984.

[Ichbiah et al. 79] J. D. Ichbiah, J. C. Heliard, O. Roubine, J. G. P. Barnes, B. Krieg-Brueckner, B. A. Wichmann. *Rationale for the Design of the ADA Programming Language*. SIGPLAN Notices, vol. 14 #6, June 1979. Part B.

[ISOPascal 82] ISO (International Organization for Standardization). *Specification for Computer Programming Language - Pascal*. ISO 7185-1982-08-12.

[Jameson 85] D. Jameson. *Correspondence: "Some Concerns About Modula-2"*. SIGPLAN Notices, vol. 20 #5, May 1985. Pg. 6–7.

[JensenWirth 85] K. Jensen, N. Wirth. *Pascal User Manual and Report*. Third Edition, revised by A. B. Mickel, J. F. Miner. Springer-Verlag, 1985.

[Johnson 75] S. C. Johnson. *Yacc: Yet Another Compiler Compiler*. Computer Science Techical Report, Nº 32. Bell Laboratories, Murray Hill, 1975.

[Johnson 78] S. C. Johnson. *Lint, a Program Checker*. Computer Science Technical Report, Nº 65. Bell Laboratories, Murray Hill, 1978.

[Kenah et al. 88] L. J. Kenah, Ruth E. Goldenberg, S. F. Bate. *VAX/VMS Internals and Data Structures. Version 4.4*. Digital Press, 1988.

[KernRit 78] B. W. Kernighan, D. M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[KernRit 88] B. W. Kernighan, D. M. Ritchie. *The State of C*. Byte. August 1988. Pg. 205–210.

[Kowaltowski 83] T. Kowaltowski. *Implementação de Linguagens de Programação*. Guanabara Dois, 1983.

[Lampson et al. 77] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, G. J. Popeck. *Report on the Programming Language Euclid*. SIGPLAN Notices, vol. 12 #2, February 1977.

[Liskov et al. 79] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, B. Scheifler, A. Snyder. *Clu Reference Manual.* Laboratory for Computer Science, MIT. MIT/LCS/TR-255. October 1979.

[Liskov et al. 87] B. Liskov, M. Day, M. Herlihy, P. Johnson, G. Leavens, R. Scheifler, W. Weihl. *Argus Reference Manual.* Laboratory for Computer Science, MIT. March 1987.

[McKee et al. 85] W. M. McKeeman, N. Martin, T. D. Gill, J. Gehling, S. Trager. *Expression Side Effects in C.* Wang Institute of Graduate Studies, Technical Report TR-85-04. May 1985.

[Meissner 82] M. Meissner. *Correspondence: "C: Toward a Concise Syntactic Description".* SIGPLAN Notices, vol. 17 #8, August 1982. Pg. 84–88.

[Metz 86] S. J. Metz. *Correspondence: "Compared Anatomy of the Programming Languages Pascal and C".* SIGPLAN Notices, vol. 21 #9, September 1986. Pg. 13–15.

[Mitchell et al. 79] J. G. Mitchell, W. Maybury, R. Sweet. *Mesa Language Manual - Version 5.0.* Xerox. April 1979.

[Moffat 84] D. V. Moffat. *Some Concerns About Modula-2.* SIGPLAN Notices, vol. 19 #12, December 1984. Pg. 41–47.

[Muller 84] H. A. Muller. *Differences between Modula-2 and Pascal.* SIGPLAN Notices, vol. 19 #10, October 1984. Pg. 32–39.

[Nelson 85] R. Nelson. *Another Solution to Modula-2's I/O.* SIGPLAN Notices, vol. 20 #11, November 1985. Pg. 9.

[Pase 85] D. M. Pase. *System Programming in Modula-2.* SIGPLAN Notices, vol. 20 #11, November 1985. Pg. 49–53.

[PedroJr 83] J. Pedro Jr. *Uma implementação de Modula-2: Análise e Representação Intermediária.* Tese de Mestrado. DCC–IMECC–UNICAMP. Novembro 1983.

[Popeck et al. 77] G. J. Popeck, J. J. Horning, B. W. Lampson, J. G. Mitchell, R. L. London. *Notes on the Design of Euclid.* Proceedings of an ACM Conference on Language Design for Reliable Software. SIGPLAN Notices, vol. 12 #3, March 1977.

[RitThomp 74] D. M. Ritchie, K. Thompson. *The Unix Time-Sharing System.* Communications of the ACM, vol. 17 #7. July 1974. Pg. 365–375.

[RitThomp 78] D. M. Ritchie, K. Thompson. *The Unix Time-Sharing System – revised version.* The Bell System Technical Journal, vol. 57 #6. July-August 1978. Pg. 1905–1929.

[Sebesta 89] R. W. Sebesta. *Concepts of Programming Languages.* Benjamin/Cummings Publishing Company, 1989.

[SegreStanton 85] L. Segre, M. Stanton. *"Some Concerns About Modula-2" Considered Unwarranted.* SIGPLAN Notices, vol. 20 #5, May 1985. Pg. 31–35.

[Sewry 84] D. A. Sewry. *Modula-2 Process Facilities.* SIGPLAN Notices, vol. 19 #11, November 1984. Pg. 23–32.

[SilvaAssis 88] J. C. G. Silva, F. S. G. Assis. *Linguagem de Programação – Conceitos e Avaliação.* McGraw-Hill – Embratel, 1988.

[Smedema 84] C. H. Smedema. *Chill User Manual.* Chill Bulletin, vol. 4, nº 1. AT&T and Philips Telecommunications. March 1984.

[Spector 82] D. Spector. *Ambiguities and Insecurities in Modula-2.* SIGPLAN Notices, vol. 17 #8, August 1982. Pg. 43–51.

[Spector 83] D. Spector. *Lexing and Parsing Modula-2.* SIGPLAN Notices, vol. 18 #10, October 1983. Pg. 25–32.

[Stroustrup et al. 84] B. Stroustrup et al. *C++ (Release E). Includes C++ Reference Manual.* AT&T Bell Laboratories. November 1984.

[Stroustrup 86] B. Stroustrup. *An Overview of C++*. SIGPLAN Notices, vol. 21 #10, October 1986. Pg. 7–18.

[Stroustrup 88] B. Stroustrup. *A Better C?* Byte. August 1988. Pg. 215–216D.

[Tennent 78] R. D. Tennent. *Another Look at Type Compatibility in Pascal.* Software–Practice and Experience, vol. 8, 1978. Pg. 429–437.

[Thompson 78] K. Thompson. *Unix Implementation.* The Bell System Technical Journal, vol. 57 #6. July–August 1978. Pg. 1931–1946.

[Torbett 87] M. A. Torbett. *A Note on "Protecting against Uninitialized Abstract Objects in Modula-2".* SIGPLAN Notices, vol. 22 #5, May 1987. Pg. 8–10.

[Torbett 87a] M. A. Torbett. *More Ambiguities and Insecurities in Modula-2.* SIGPLAN Notices, vol. 22 #5, May 1987. Pg. 11–17.

[Welsh et al. 77] J. Welsh, W. J. Sneeringer, C. A. R. Hoare. *Ambiguities and Insecurities in Pascal.* Software–Practice and Experience, vol. 7, 1977. Pg. 685–696.

[WienerSinc 85] R. S. Wiener, R. F. Sincovec. *Two Approaches to Implementing Generic Data Structures in Modula-2.* SIGPLAN Notices, vol. 20 #6, June 1985. Pg. 56–64.

[Wiener 86] R. S. Wiener. *Protecting against Uninitialized Abstract Objects in Modula-2.* SIGPLAN Notices, vol. 21 #6, June 1986. Pg. 63–69.

[Wirth 77] N. Wirth. *Modula: A Language for Modular Multiprogramming.* Software–Practice and Experience, vol. 7, 1977. Pg. 3–35.

[Wirth 77a] N. Wirth. *The Use of Modula.* Software–Practice and Experience, vol. 7, 1977. Pg. 37–65.

[Wirth 85] N. Wirth. *Programming in Modula-2. Third, Corrected Edition.* Springer-Verlag, 1985.

[Wirth 88] N. Wirth. *From Modula to Oberon.* Software–Practice and Experience, vol. 18 (7), 1988. Pg. 661–670.

[Wirth 88a] N. Wirth. *The Programming Language Oberon.* Software–Practice and Experience, vol. 18 (7), 1988. Pg. 671–690.

[Wirth 88b] N. Wirth. *Type Extensions.* ACM Transactions on Programming Languages and Systems, vol. 10, #2. April 1988. Pg. 204–214.

[Zimmer 85] J. A. Zimmer. *A Modest Modula with List.* SIGPLAN Notices, vol. 20 #11, November 1985. Pg. 69–77.

Linguagem MC - Manual de Referência

Claudio Sergio Da Rós de Carvalho

DCC - IMECC - UNICAMP

21 de agosto de 1989

Sumário

A linguagem de programação MC incorpora algumas facilidades da linguagem C à estrutura da linguagem Modula-2. Além disso, procura ser mais versátil, introduzindo os conceitos de *exceções, vetores com limites abertos, subprogramas com número variável de parâmetros* e *processos*, entre outros, resultando numa linguagem de propósito geral, mas adequada à programação de sistemas. O objetivo desse documento é servir como um manual de referência para programadores e implementadores dessa linguagem.

Conteúdo

Capítulo 1

Introdução

A linguagem de programação MC surgiu com o intuito de unir as características principais da linguagem C [KernRit 78] (facilidades de baixo nível, falta de restrições e sua generalidade) à estrutura da linguagem Modula-2 [Wirth 85] (conceito de módulos, forte verificação de tipos e acesso ordenado às facilidades de baixo nível). Além disso, inclui o conceito de exceções, como uma solução intermediária entre as propostas por Ada [DraftAda 83] e Chill [Smedema 84]. O resultado é uma linguagem versátil, de propósito geral mas adequada à programação de sistemas.

Os tipos de dados definidos na linguagem são praticamente os mesmos oferecidos por Modula-2 e C: simples (inteiros e reais, enumerados e subintervalos), estruturados (vetores, vetores com limites abertos, registros, registros com variantes e conjuntos), apontadores e identificadores de processos.

MC é uma linguagem de expressões, no sentido que cada comando resulta num valor, com um certo tipo associado. Uma questão importante é a definição de compatibilidade entre tipos, que alguns autores criticam desde o Pascal [JenWirth 85]. MC define precisamente o que é compatibilidade entre tipos em quatro situações: compatibilidade para operação, para atribuição, para passagem de parâmetros e para devolução de resultado de função. Nesse sentido, a verificação da compatibilidade, em cada um dos aspectos citados, é estrita. No entanto, a linguagem provê mecanismos para burlar esse esquema de verificação, através de funções de transferência e compatibilização, seguindo a proposta de Modula-2 e se afastando da versão original de C, que não realiza consistência estrita entre tipos. Com isso, a versatilidade em tratar um objeto de um certo tipo como se fosse de outro ainda é mantida; a vantagem desse esquema é que essa manipulação deve ser explicitada, deixando claro que essa facilidade da linguagem está sendo utilizada. No entanto, esse mecanismo especial não diminui a

eficiência de execução.

C apresenta outra característica importante que é a não verificação do número de parâmetros efetivos na chamada de uma função; esta se comporta como se tivesse um número variável de parâmetros. MC provê uma descrição sintática que descreve subprogramas efetivamente com um número variável de parâmetros.

A linguagem MC permite compilação em separado através do uso de módulos, da mesma forma que em Modula-2: módulos de programa, de definição e implementação. Módulos especiais fazem parte da definição da linguagem como SYSTEM, LOW_LEVEL e MATH_LIB, que trazem os elementos que são dependentes de implementação, os mecanismos para a utilização de recursos de baixo nível e, finalmente, as funções aritméticas. A ativação de processos, diferentemente de Modula-2, é um mecanismo previsto na linguagem; a comunicação entre processos deve, em última instância, utilizar as facilidades do módulo LOW_LEVEL.

A definição de entrada e saída não faz parte da linguagem, assim como em Modula-2 e C. Com os recursos providos por MC, é possível escrever módulos que resolvam esse problema. O mesmo ocorre com a manipulação de cadeias de caracteres.[1]

Exceções são um mecanismo incorporado à linguagem que possibilita o tratamento de situações excepcionais (possivelmente erros) de maneira mais simples e segura.

O resultado é uma linguagem simples, de propósito geral, que permite acesso às facilidades de baixo nível, mas de maneira ordenada e segura. A sintaxe está mais próxima de Modula-2, incorporando algumas facilidades das linguagens C, Ada e Chill.

[1] Em inglês: *strings*.

Capítulo 2

Sintaxe

Uma linguagem é um conjunto possivelmente infinito de sentenças, que pode ser descrita por uma gramática livre de contexto. Como em Modula-2, sentenças bem formadas, de acordo com sua sintaxe, são chamadas *unidades de compilação*. Cada unidade é uma seqüência finita de símbolos de um vocabulário finito, composto por identificadores, cadeias, números, operadores e delimitadores. Todos esses símbolos são compostos por seqüências de caracteres.

Na descrição da sintaxe será usada a notação própria para a utilização do gerador de analisador sintático YACC [Johnson 75]. Uma regra sintática tem a forma:

NT : *corpo_da_regra*
;

NT representa um não-terminal; *corpo_da_regra* representa uma seqüência de zero ou mais nomes e literais. Um nome se refere a símbolos terminais (também chamados *átomos*) ou a símbolos não-terminais. Nesse texto, o átomo correspondente a uma palavra reservada da linguagem será uma seqüência de caracteres em **negrito**. Cada símbolo não-terminal NT deve aparecer do lado esquerdo do símbolo ":" ao menos em uma regra sintática. Um literal consiste de um ou mais caracteres entre aspas simples ('). Se várias regras gramaticais têm o mesmo símbolo NT à esquerda do sinal de ":", a barra vertical ("|") pode ser usada para evitar a repetição de NT. Um ponto e vírgula (";") termina uma regra. Comentários serão quaisquer seqüências de caracteres entre "/*" e "*/".

Capítulo 3

Símbolos

A representação de símbolos em termos de caracteres utilizará o conjunto ASCII. Brancos[1] e terminadores de linha[2] são ignorados a menos que sejam essenciais para separar dois símbolos consecutivos ou então, no caso de brancos, que ocorram em cadeias.

3.1 Identificadores

identificador : *letra*
| *identificador letra*
| *identificador dígito*
| *identificador* '_'
;

dígito : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
;

letra : 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O'
| 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o'
| 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
;

Um identificador é uma seqüência de caracteres ASCII que correspondem às letras maiúsculas e minúsculas, os dígitos de 0 a 9 mais o caractere '_'. Qualquer outro caractere delimita o identificador. Letras minúsculas e maiúsculas não são

[1] Caracteres ' ' (código 32) e <TAB> (código 9).
[2] Caracteres <CR> (código 13) e/ou <LF> (código 10).

consideradas distintas quando representam identificadores e palavras reservadas. O comprimento máximo de um identificador é dependente da implementação. O uso de caracteres que não fazem parte do conjunto ASCII também é dependente de implementação.

3.2 Números

número : *inteiro* | *real*
;

inteiro : *inteiro_decimal* | *inteiro_hexadecimal*
;

inteiro_decimal : *dígito* | *inteiro_decimal dígito*
;

inteiro_hexadecimal : *dígito dígitos_hexadecimais base_hexadecimal*
;

dígitos_hexadecimais : /* vazio */
| *dígitos_hexadecimais dígito_hexadecimal*
;

dígito_hexadecimal : *dígito* | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
;

base_hexadecimal : 'H' | 'h'
;

real : *parte_inteira* '.' *parte_fracionária fator_de_escala*
;

parte_inteira : *inteiro_decimal*
;

parte_fracionária : /* vazio */
| *parte_fracionária dígito*
;

fator_de_escala : /* vazio */
| *símbolo_de_escala sinal_do_expoente inteiro_decimal*
;

símbolo_de_escala : 'E' | 'e'
;

sinal_do_expoente : '+' | '−' | /* vazio */
;

Tipo	Intervalo	Intervalo em computadores de 16 bits
SHORTINT	[MINSHORTINT..MAXSHORTINT]	[−128..127]
SHORTCARD	[0..MAXSHORTCARD]	[0..255]
INTEGER	[MININT..MAXINT]	[−32768..32767]
CARDINAL	[0..MAXCARD]	[0..65535]
UNIV_INTEGER	[MINUNIVINT..MAXUNIVINT]	[−32768..65535]

Tabela 3.1: Constantes do módulo SYSTEM para inteiros predefinidos

Números são inteiros ou reais. Inteiros são seqüências de dígitos. Se um inteiro for delimitado à direita pelas letras *H* ou *h*, será um inteiro hexadecimal. Um inteiro *i*, a menos de explícita indicação em contrário, através do uso de funções de transferência de tipos (cf. 6.5.6), é considerado do tipo predefinido UNIV_INTEGER.[3] Outros tipos numéricos inteiros predefinidos são SHORTINT, SHORTCARD, INTEGER e CARDINAL. Constantes definidas no módulo SYSTEM dão os intervalos representáveis por cada um desses tipos. A tabela 3.1 traz o valor dessas constantes para computadores comuns de 16 bits.

Um número real contém um ponto decimal e é escrito na base decimal. Opcionalmente, contém um fator decimal de escala. As letras *E* ou *e* devem ser lidas como "dez elevado à potência de". Um número real é do tipo predefinido REAL.

3.3 Caracteres e Cadeias

Um caractere em MC é um símbolo gráfico do conjunto de caracteres representável por uma dada implementação, entre aspas simples ('), ou então, o resultado da função CHR do módulo SYSTEM. Os caracteres aspa simples ('), aspa dupla (") e barra invertida (\)[4] devem ser representados por \', \" e \\, respectivamente.

Uma cadeia é uma seqüência de caracteres entre aspas duplas ("). A função CHR não pode ser usada dentro de uma cadeia; para incluir seu resultado numa cadeia deve-se usar o operador de concatenação (operador &; cf. 9.2.4). O *tamanho* ou *comprimento* de uma cadeia é o número de caracteres que a compõem.

cadeia : "*seqüência_de_caracteres*"

[3] O tipo "inteiro-universal" (UNIV_INTEGER) não pode ser explicitamente usado por um programa em MC; é usado apenas para compatibilização dos tipos que são subconjuntos de inteiros (cf. 6.1.1).

[4] Códigos ASCII 39, 34 e 92, respectivamente.

```
        ;
seqüência_de_caracteres : /* vazio */
                        | seqüência_de_caracteres caractere
                        ;
```

Exemplos de cadeias: "MC", "\"Ser ou não ser.\"", "Barra invertida: " & CHR(92) & ".", "" & CHR(127).

3.4 Operadores e Símbolos

Operadores e símbolos são caracteres especiais e pares de caracteres (tabela 3.2) ou palavras reservadas (listadas na tabela 3.3). Não há diferença entre caracteres maiúsculos e minúsculos em uma palavra reservada.

3.5 Comentários

Comentários podem ser inseridos entre dois símbolos quaisquer do programa-fonte. São uma seqüência arbitrária de caracteres entre "/*" e "*/". Podem ser encaixados.

"	'	..	:=	<<	>>
++	--	=	#	<	<=
>	>=	\	;	(	)
.	[	]	:	,	|
↑	{	}	+	-	&
*	/	%			

Tabela 3.2: Operadores e Símbolos de MC

and	**end**	**in**	**procedure**	**until**
array	**exception**	**inparray**	**process**	**var**
begin	**exceptions**	**loop**	**raise**	**with**
by	**exit**	**module**	**record**	**when**
case	**export**	**not**	**repeat**	**while**
const	**for**	**notin**	**return**	**xor**
continue	**from**	**of**	**set**	
definition	**function**	**or**	**start**	
do	**if**	**others**	**then**	
else	**implementation**	**outparray**	**to**	
elsif	**import**	**pointer**	**type**	

Tabela 3.3: Palavras Reservadas de MC

Capítulo 4

Módulos e Unidades de Compilação

Módulos permitem a especificação de grupos de entidades logicamente relacionadas, podendo ser usados para esconder detalhes de implementação e para proteger seus dados contra acessos indevidos. Um módulo estabelece um escopo, isto é, restringe o intervalo de visibilidade de objetos (que, em última instância, são *identificadores*) e suas propriedades, exportando (tornando visíveis) apenas aqueles que servirão como interface entre módulos, o que facilita sua manutenção e integridade. MC apresenta módulos de *definição*, de *implementação*, *de programa* e *locais*. O módulo de definição é como um prefixo do módulo de implementação correspondente e especifica os nomes e propriedades dos objetos que podem ser usados por outros módulos;[1] o módulo de implementação contém objetos e comandos locais (por exemplo, corpos de subprogramas cujos cabeçalhos se encontram no módulo de definição correspondente) e que não devem ser conhecidos externamente. Um módulo de programa constitui um programa principal e nada pode ser exportado por ele. Um módulo local pode ser declarado em um bloco (cf. 8.1) e sua finalidade é esconder os detalhes de seus objetos declarados internamente, cuja transparência é estritamente controlada por suas listas de importação e exportação.

Um texto que é aceito pelo compilador como uma unidade é chamado *unidade de compilação*. Há três tipos de unidades de compilação em MC: módulos de programa, módulos de definição e módulos de implementação.

Módulos de definição e de implementação são as duas partes constituintes de um módulo e, portanto, devem ter o mesmo identificador que dá o nome a

[1] Diz-se que tais objetos são *exportados* pelo módulo.

ele. Esses módulos podem existir aos pares.[2] Ambos podem importar objetos e todos os que forem declarados no módulo de definição estão disponíveis no módulo de implementação correspondente sem uma importação explícita. MC provê um esquema de *compilação em separado* (mas não independente, no sentido que um módulo de implementação necessariamente usará seu módulo de definição correspondente e outros módulos de definição que são importados, se existirem, já compilados).

unidade_de_compilação : *módulo_de_definição*
| *módulo_de_programa*
| **implementation** *módulo_de_implementação*
;

módulo de_programa : *módulo_de_implementação*
;

Os conceitos de módulos de MC são idênticos aos de Modula-2 [Wirth 85].

4.1 Módulo de Definição

módulo de_definição : **definition module** *identificador* ';'
parte_de_importação parte_de_definições
end *identificador* '.'
;

Um módulo de definição representa a interface entre o módulo de implementação e outros módulos. Contém as declarações dos objetos que podem ser usados por outros módulos, chamados *módulos clientes*. É uma lista de exportação (estendida) do módulo de implementação e declara todos os objetos (identificadores) exportados. O identificador que segue o símbolo **module** é o nome do módulo e também deve seguir o símbolo **end** que o finaliza.

Todos os identificadores declarados são globais, embora sejam visíveis e possibilitem acesso apenas aos módulos que os importam, sendo invisíveis a outros. A declaração de *procedimentos, funções* e *processos* consiste apenas de seu cabeçalho. *Tipos* declarados nesse módulo tornam visíveis seus detalhes de implementação a todos os módulos que os importam (*exportação transparente*). A

[2] Um módulo de implementação pode não ter um módulo de definição. Considere, por exemplo, o caso de um módulo que dispara um relógio, utilizando as facilidades de baixo nível (definidas no módulo LOW_LEVEL). Analogamente, um módulo de definição pode não ter o de implementação correspondente. Como exemplo, considere um módulo que somente apresente declarações de constantes, tipos e variáveis.

exportação opaca consiste apenas na declaração do nome do tipo, restringindo-se a apontadores.

parte_de_definições : /* vazio */
| *parte_de_definições* **const** *declaração_de_constantes*
| *parte_de_definições* **type** *declaração_de_tipos*
| *parte_de_definições* **var** *declaração_de_variáveis*
| *parte_de_definições* **exception** *declaração_de_exceções*
| *parte_de_definições cabeçalho_de_processo*
| *parte_de_definições cabeçalho_de_subprograma*
;

4.2 Módulo de Implementação

módulo_de_implementação : **module** *identificador_com_atributos* ';'
parte_de_importação bloco
identificador '.'
;
identificador_com_atributos : *identificador especificação_de_atributos*
;
especificação_de_atributos : /* vazio */
| '[' *lista_de_expressões* ']'
;

Um módulo de implementação pode apresentar um módulo de definição associado, isto é, com o mesmo nome. Nesse caso, as declarações e importações feitas neste último são, automaticamente, consideradas como pertencentes ao primeiro. O identificador que segue o símbolo **module** é o nome do módulo e deve anteceder também o símbolo '.' que delimita seu final.

Um módulo de implementação pode importar identificadores de outros módulos. Contém objetos, subprogramas e comandos que manipulam as estruturas locais, importadas ou exportadas. O escopo de suas declarações é local, isto é, nenhum identificador é conhecido externamente ao módulo, a menos que esteja presente no módulo de definição correspondente.

Os atributos que podem ser associados a módulos são os que dizem respeito a opções de compilação. A forma de especificação dos mesmos é dependente de implementação e não exige a importação do identificador *SpecialAttributes* do módulo LOW LEVEL como no caso da especificação de atributos de variáveis, subprogramas e processos. Os atributos especificados em módulos se estendem a toda entidade sintaticamente encaixada nos mesmos.

4.3 Módulo de Programa

Um programa em MC é, essencialmente, um módulo de implementação no qual não há listas de exportação. Assim, não existe um módulo de definição associado a um módulo de programa.

4.4 Módulos Especiais

Módulos podem formar bibliotecas de rotinas e ser colocados à disposição de usuários em um sistema de programação. Em particular, alguns módulos especiais são definidos, constituindo parte integrante de MC:

- SYSTEM – exporta as entidades que são dependentes da configuração do sistema a ser utilizado (por exemplo, os limites dos tipos numéricos e as funções predefinidas). Seus identificadores são *penetrantes* (cf. 6), com algumas exceções, e é importado, automaticamente, por todos os módulos de MC.
- LOW_LEVEL – permite o acesso às facilidades de manipulação de baixo nível (por exemplo, o tratamento de interrupções e o uso de funções de transferência de tipos).
- MATH_LIB – biblioteca de rotinas aritméticas.

4.5 Módulos Locais

Um bloco (cf. 8.1) pode trazer a declaração de um módulo local. Diferentemente de outros módulos, não é uma entidade compilável separadamente; seu propósito é somente esconder detalhes de seus objetos declarados internamente. Cada módulo estabelece um escopo de visibilidade de identificadores (cf. 8.2).

declaração_de_módulo : **module** *identificador_com_atributos* ';'
parte_de_importação parte_de_exportação
bloco identificador
;

Os dois identificadores presentes em *declaração_de_módulo* devem ser o mesmo e dão um nome ao módulo. Identificadores podem ser importados pelo módulo local para sua utilização ou exportados para a utilização pelo bloco que contém esse módulo. Os módulos locais são ativados (cf. 8.3) quando o bloco que os contém é ativado, segundo sua ordem textual de declaração.

4.6 Importação e Exportação

A parte de importação de um módulo especifica todos os identificadores declarados em outros módulos e que podem ser usados por este. A parte de exportação especifica todos os identificadores declarados em um módulo e que podem ser usados por outros.

parte_de_importação : /* vazio */
| *parte_de_importação* **from** *identificador* **import** *lista_de_identificadores* ';'
| *parte_de_importação* **import** *lista_de_identificadores* ';'
;

parte_de_exportação : /* vazio */
| **export** *lista_de_identificadores* ';'
;

lista_de_identificadores : *lista_de_identificadores* ',' *identificador*
| *identificador*
;

Um módulo pode apresentar várias listas de identificadores a serem importados, que devem ser precedidos pelo símbolo **from** e o identificador que é o nome de um módulo. O uso de um identificador importado dessa maneira é feito de maneira não qualificada, isto é, tudo se passa como se o identificador tivesse sido declarado no módulo que o está importando, estando sujeito às regras de conflito de nomes em declarações e escopo descritas na seção 8.2. Se a cláusula *from* não é utilizada, importam-se os nomes de módulos e todos os identificadores que estes exportam. Mesmo que identificadores iguais ocorram em mais de um módulo, não haverá conflito de nomes pois seu uso deve ser qualificado, como é feito com identificadores de campos de registros, ou seja, um identificador importado deve ser precedido pelo nome do módulo no qual foi declarado. Assim, se um módulo `M` exporta os identificadores `a,b,c`, a especificação **import** `M` em um módulo `P` faz com que, no módulo `P`, esses identificadores devam ser referenciados de maneira qualificada por `M.a`, `M.b`, `M.c`, respectivamente; se é feita a especificação **from** `M` **import** `a,b` em `P`, os identificadores `a,b` de `M` devem ser referenciados por `a,b` em `P`. Nesse caso, `c` não é visível nesse módulo.

Se uma lista de importação ocorre em um módulo local fazendo referência ao nome do módulo no qual está imediatamente encaixado sintaticamente, a cláusula *from* pode ser omitida. Se um tipo registro é exportado, todos os seus identificadores de campos também o são.[3] O mesmo ocorre com os identificadores

[3]Note que os nomes de tipos não são exportados automaticamente nesse caso.

de constantes de um tipo enumerado.

Uma exportação especificada por um módulo local é, automaticamente, uma importação implícita pelo módulo que imediatamente o contém; uma importação por um módulo local é também uma exportação implícita pelo módulo que imediatamente o contém.

A importação não apresenta a propriedade transitiva. Para exemplificar esta observação, considere três módulos: a, b e c. Se b importa a e c importa b, c não poderá utilizar os identificadores exportados por a, utilizados por b, a menos que c importe a.

4.7 Unidade de Execução

Uma unidade de execução ou programa é obtida a partir de um módulo de programa, que pode ser ligado com outros módulos ou rotinas auxiliares, dependendo do sistema a ser utilizado.

A ativação de um programa obtido dessa maneira consiste na ativação do bloco do módulo de programa. À ativação do módulo deste antecede a ativação dos módulos por ele incluídos, segundo sua ordem textual (cf. 8.3).

Capítulo 5

Constantes

Uma declaração de constante associa um identificador a um valor constante, resultado de uma expressão constante, isto é, uma expressão que seja possível de ser avaliada por uma simples análise textual, sem que o programa tenha que ser realmente executado. Numa seção de declaração de constantes, a ordem de avaliação segue a ordem textual. Portanto, expressões que façam referência a outras constantes resultarão em erro de compilação caso estas não estejam com seu valor definido.

declaração_de_constantes : /* vazio */
| *declaração_de_constantes identificador* '=' *expressão_constante* ';'
;
expressão_constante : *expressão*
;

O tipo de uma constante é o tipo da expressão associada (cf. 6). Exemplos:

```
const m = 15;                 /* m: UNIV_INTEGER */
      sm = SHORTINT(15);      /* sm: SHORTINT */
      ch_A = 'A';             /* ch_A: CHAR */
      m2 = 2 * m;             /* m2: UNIV_INTEGER */
      cad = "A" & CHR(92);    /* cad: array [1..2] of CHAR */
```

Capítulo 6

Tipos

Um tipo caracteriza um conjunto de valores e um conjunto de operações que podem ser aplicados a estes valores. Uma declaração de tipo sinônimo associa um nome a um tipo. Cada declaração de tipo, opaco ou sinônimo, introduz um novo tipo.

declaração_de_tipos : /* vazio */
| *declaração_de_tipos declaração_de_tipo_opaco*
| *declaração_de_tipos declaração_de_tipo_sinônimo*
;

declaração_de_tipo_opaco : *identificador* ';'
;

declaração_de_tipo_sinônimo : *identificador* '=' *tipo* ';'
;

lista_de_tipos_sinônimos : /* vazio */
| *lista_de_tipos_sinônimos declaração_de_tipo_sinônimo*
;

Há quatro classes de tipos: *escalares*, cujos valores não apresentam componentes atômicos; *estruturados*, cujos valores consistem de vários componentes; *apontadores*, que possibilitam acesso a outros objetos e *identificadores de processos*, que possibilitam a comunicação entre processos.

Um *tipo opaco* ou *privado* é usado em módulos de definição; o efeito é a exportação de um tipo cuja estrutura é escondida de outros módulos, bem como os detalhes das operações sobre ele. A implementação de um tipo privado é feita utilizando-se apontadores. Um tipo *sinônimo* pode ser usado em qualquer tipo de módulo; introduz um novo tipo através da associação de um identificador a um tipo.

tipo : *nome_de_tipo*
| *intervalo*
| *enumeração*
| *tipo_estruturado*
| *tipo_apontador*
;
nome_de_tipo : *identificador_de_tipo*
;
identificador_de_tipo : *identificador*
| *identificador_de_tipo* '.' *identificador*
;

Um tipo pode ser usado em declarações de outros tipos ou, diretamante, para descrever tipos de objetos.

6.1 Tipos Escalares

Um tipo escalar ou simples determina um conjunto ordenado de valores. Há duas classes de tipos simples: *real* e *ordinal*. O tipo real implementa um subconjunto finito dos números reais; o tipo ordinal apresenta uma correspondência um-a-um entre os seus valores e um conjunto finito de números ordinais consecutivos.

6.1.1 Tipos Simples Predefinidos

Alguns tipos simples são predefinidos em MC e apresentam identificadores-padrão *penetrantes*,[1] isto é, não precisam ser importados de módulo algum, pois já fazem parte da linguagem:

- real: REAL
- ordinal: BOOLEAN, CHAR, SHORTINT, SHORTCARD, INTEGER, CARDINAL

Especificação dos tipos simples predefinidos:

REAL - determina um subconjunto finito dos números reais. Seu tamanho (número de bits) e subconjunto representado são dependentes de implementação.

[1] Em inglês: *pervasive identifiers*.

BOOLEAN – é um tipo enumerado que determina o conjunto de valores lógicos denotados pelos identificadores penetrantes de constantes TRUE e FALSE, onde FALSE < TRUE.

CHAR – determina o conjunto da caracteres ASCII (códigos entre 0 e 127, inclusive). Caracteres com códigos entre 128 e 255 são dependentes de implementação.

SHORTINT – inclui um subconjunto dos inteiros com sinal. Compreende os valores entre MINSHORTINT e MAXSHORTINT.

SHORTCARD – inclui um subconjunto dos inteiros sem sinal. Compreende os valores entre 0 e MAXSHORTCARD.

INTEGER – inclui um subconjunto dos inteiros com sinal. Compreende os valores entre MININT e MAXINT.

CARDINAL – inclui um subconjunto dos inteiros sem sinal. Compreende os valores entre 0 e MAXCARD.

MC apresenta um outro tipo ordinal, chamado tipo "inteiro-universal" (UNIV_INTEGER), que é usado para compatibilização dos tipos que são subconjuntos de inteiros. Compreende os valores entre MINUNIVINT e MAXUNIVINT. Esse tipo não é passível de utilização explícita por um programa em MC.

Observações:

$$\begin{aligned} \text{MININT} \leq \text{MINSHORTINT} < 0 \\ 0 < \text{MAXSHORTCARD} \leq \text{MAXINT} \leq \text{MAXCARD} \\ \text{MINUNIVINT} \leq \text{MININT} \\ \text{MAXCARD} \leq \text{MAXUNIVINT} \end{aligned}$$

Os tipos ordinais predefinidos SHORTINT, SHORTCARD, INTEGER, CARDINAL, mais o tipo UNIV_INTEGER constituem os *tipos inteiros.* Os tipos inteiros, juntamente com o tipo predefinido REAL, constituem os *tipos numéricos.*

O tipo de uma constante inteira, a menos de explícita indicação em contrário (por exemplo, através do uso de funções de transferência de tipos) é UNIV_INTEGER. No entanto, dependendo do contexto, uma constante pode ser considerada de um dos tipos inteiros predefinidos, dependendo do intervalo de representação que estes definem. Assim, uma constante será do primeiro tipo simples predefinido listado a seguir em cujo intervalo de definição ela se encontra

ou de qualquer tipo seguinte, respeitadas a mesma regra de intervalo, de acordo com o contexto.

1. SHORTCARD
2. SHORTINT
3. CARDINAL
4. INTEGER

Exemplo:

```
m * 2          /* m: CARDINAL; 2: CARDINAL */
a := 2         /* a: SHORTINT; 2: SHORTINT */
```

6.1.2 Tipos Enumerados

Uma enumeração é uma lista de identificadores que denotam os valores que constituem um tipo de dados. Esses identificadores são usados como constantes no programa. Os valores são ordenados e a relação de ordem é crescente, do primeiro ao último identificador. O valor ordinal do primeiro valor é zero.

enumeração : '(' *lista_de_identificadores* ')'
;

6.1.3 Tipos Subintervalos

Um tipo T pode ser definido como um subintervalo de um tipo ordinal T' pela especificação de seus valores mínimo e máximo. O tipo T' especificado por esses limites é chamado de *tipo base* de T.

intervalo : *especificação_de_subintervalo*
| *identificador_de_tipo especificação_de_subintervalo*
;
especificação_de_subintervalo : '[' *expressão_constante* '..' *expressão_constante* ']'
;

A primeira expressão constante especifica o limite inferior do subintervalo e a segunda, o limite superior. Se o limite inferior for maior que o superior, diz-se que o subintervalo é *vazio*. O tipo base de um tipo subintervalo T pode ser especificado

por um identificador qualificado.[2] Neste caso, as constantes devem ser de tipo resultante compatível para atribuição com o tipo base especificado (cf. 6.5.4). Se o identificador qualificado for omitido, o tipo base de T será determinado pelos tipos das expressões constantes que definem os seus limites; ambas devem ser do mesmo tipo resultante e este será o tipo base de T.

Em se tratando de um subintervalo de inteiros, o tipo base de T será o primeiro tipo simples predefinido listado a seguir com o qual as duas expressões constantes que definem o intervalo são compatíveis para atribuição:[3]

1. SHORTCARD
2. SHORTINT
3. CARDINAL
4. INTEGER

Caso contrário, o subintervalo de inteiros não será válido.

Todos os operadores aplicáveis ao tipo base de um subintervalo também o são ao tipo subintervalo.

6.2 Tipos Estruturados

Um tipo estruturado é caracterizado pelos tipos de seus componentes e seu método de estruturação.

tipo_estruturado : *tipo_vetor*
| *tipo_registro*
| *tipo_conjunto*
;

6.2.1 Tipos Vetores

Um vetor é uma estrutura que apresenta um número fixo de componentes (zero ou mais), todos do mesmo tipo, chamado *tipo do componente*. Os componentes são designados por índices e estão em correspondência um-a-um com os valores possíveis dos mesmos.

[2]Um identificador é um caso particular de um identificador qualificado.

[3]O tipo é determinado de maneira que ambas as expressões constantes pertençam ao intervalo fechado definido por ele.

tipo_vetor : *vetor_com_limites_definidos*
| *vetor_com_limites_abertos*
;

Uma declaração de tipo vetor especifica o *tipo do componente* bem como o *tipo do índice*. MC permite tanto a declaração de *vetores com limites definidos* quanto a de *vetores com limites abertos*. O acesso a cada um dos elementos do vetor é feito através de *designadores* (cf. 9.2.2).

Vetores com Limites Definidos

Na declaração desse tipo de vetor é especificado o número de seus componentes, através dos seus limites inferior e superior.

vetor_com_limites_definidos : **array** *lista_de_tipos_simples* **of** *tipo*
;
lista_de_tipos_simples : *lista_de_tipos_simples* ',' *tipo_simples*
| *tipo_simples*
;
tipo_simples : *tipo*
;

Cada tipo de índice do vetor com limites definidos deve ser um tipo simples: enumerado, subintervalo ou um dos tipos simples ordinais predefinidos ou sinônimos. Como cada um desses tipos especifica um intervalo de valores possíveis, os limites do vetor ficam definidos.

Exemplos:

```
type
    T1 = array CHAR of CHAR;
         /* tipo de índice: CHAR; limites: CHR(0),CHR(255) */
    T2 = array [1..20] of INTEGER;
         /* tipo de índice: subintervalo de inteiros; limites: 1,20;
            tipo base do tipo de índice: SHORTCARD */
```

Uma declaração da forma

array T_1,T_2,...,T_n **of** T

com n tipos de índices, T_1,T_2,...,T_n, é uma abreviação da declaração

array T_1 **of**
array T_2 **of** ...
array T_n **of** T

Vetores com Limites Abertos

Vetor com limites abertos é uma extensão do conceito de vetor *semelhante* ou *conforme*[4] do Pascal [Cooper 83] e permite a manipulação de vetores cujo tamanho (seus limites inferior e superior) é definido em tempo de execução. Seu uso é restrito a *variáveis dinâmicas* (cf. 7) e a *parâmetros de subprogramas e processos* (cf. 11.3). Um vetor com limites abertos não pode ser componente de vetores com limites definidos ou de registros.

A especificação de um tipo vetor com limites abertos é incompleta, não sendo definidos os valores de seus limites mas apenas o seu tipo.

vetor_com_limites_abertos : **array** '[' *declaração_de_limites_de_vetor* ']' **of** *tipo*
;
declaração_de_limites_de_vetor : *identificador* '..' *identificador* ':' *tipo_simples*
| *declaração_de_limites_de_vetor* ',' *identificador* '..' *identificador* ':' *tipo_simples*
;

Um par de variáveis com os nomes dados pelos identificadores do subintervalo na especificação do índice, de mesmo tipo (chamado *tipo do índice*), será usado para armazenar os limites do vetor quando este for criado (variável dinâmica) ou quando for passado como parâmetro para um subprograma ou processo. O tipo do índice deve ser um tipo simples: enumerado, subintervalo ou um dos tipos simples ordinais predefinidos ou sinônimos.

Vetores com Limites Abertos como Variáveis Dinâmicas

A criação do vetor, como variável dinâmica, dá uma instância aos seus limites.

Uma declaração de tipo

array [L_1..U_1: T_1, L_2..U_2: T_2,..., L_n..U_n: T_n] **of** T

é funcional mas não sintaticamente equivalente à declaração de um tipo registro (cf. 6.2.2) da forma:

[4] Em inglês: *conformant array*.

record
  L_1,U_1: T_1;
  L_2,U_2: T_2;
    ⋮
  L_n,U_n: T_n;
  "Vetor_Componente": **array** T_1 [L_1..U_1], T_2 [L_2..U_2],...,
                                T_n [L_n..U_n] **of** T
end

Os valores dos limites do vetor são obtidos quando da criação da variável dinâmica deste tipo, através de um procedimento de alocação adequado. No caso de se utilizar o procedimento predefinido NEW, do módulo SYSTEM, além do parâmetro que indica o tipo da variável a ser criada, deve-se passar uma lista de expressões que serão usadas para dar os limites do vetor aberto.

Vetores com Limites Abertos como Parâmetros Formais

Um parâmetro formal (cf. 11.3) de tipo vetor com limites abertos tem o seu tamanho estabelecido quando da associação do parâmetro efetivo correspondente na chamada do subprograma. O parâmetro efetivo deve ser *conforme* (cf. 6.5.5) com o parâmetro formal.

Se o mecanismo de passagem é por valor (cf. 11.3.1) a declaração de parâmetro formal

$$\text{p: } \textbf{array } [L_1..U_1\text{: } T_1,\ L_2..U_2\text{: } T_2, \ldots,\ L_n..U_n\text{: } T_n] \textbf{ of } T$$

é funcional mas não sintaticamente equivalente à declaração de um tipo registro da forma:

$$\begin{array}{l}
\textbf{type } T' = \textbf{record} \\
\qquad L_1,U_1\text{: } T_1; \\
\qquad L_2,U_2\text{: } T_2; \\
\qquad\quad \vdots \\
\qquad L_n,U_n\text{: } T_n; \\
\qquad \text{“Vetor_Componente”: } \textbf{array } T_1\ [L_1..U_1],\ T_2\ [L_2..U_2], \ldots, \\
\qquad\qquad\qquad\qquad\qquad\qquad T_n\ [L_n..U_n] \textbf{ of } T \\
\quad \textbf{end}
\end{array}$$

e à declaração do parâmetro formal **p** como

$$\text{p: } T'$$

Os valores dos limites efetivos L_i, U_i e dos elementos de “`Vetor_Componente`”$[i_1,\ldots,i_n]$ são copiados quando da associação entre os parâmetros efetivo e formal. O pseudo-identificador “`Vetor_Componente`” não deve ser usado para se fazer acesso aos elementos do vetor.

Exemplo:

```
var
    VetPequeno: array [1..2] of REAL;
    VetGrande: array [MINSHORTINT..MAXSHORTINT] of REAL;
    Soma: REAL;
                    ⋮
```

```
procedure SomaVet (var t: REAL; p: array [L..U: INTEGER] of REAL);
    var i: INTEGER;
        s: REAL := 0.0;
begin
    for i := p.L to p.U do s := s + p[i] end;
    t := s
end SomaVet;
```

Chamadas corretas desse procedimento seriam:

```
SomaVet (Soma, VetPequeno);
SomaVet (Soma, VetGrande);
```

Na chamada `SomaVet(Soma,VetPequeno)`, p.L = 1, p.U = 2. No corpo do subprograma, `p[i]` pode ser visto como uma abreviação de `p."Vetor_Componente"[i]`. Os limites do vetor se comportam como constantes cujos valores são definidos quando da associação entre os parâmetros efetivo e formal.

Se o mecanismo de passagem é por variável (cf. 11.3.1), a declaração do parâmetro formal

var pp: **array** [$L_1..U_1$: T_1, $L_2..U_2$: T_2,..., $L_n..U_n$: T_n] **of** T

é funcional mas não sintaticamente equivalente à declaração de um tipo registro da forma:

type T'' = **record**
L_1,U_1: T_1;
L_2,U_2: T_2;
⋮
L_n,U_n: T_n;
"Apontador_Componente": **pointer to array** T_1 [$L_1..U_1$],
T_2 [$L_2..U_2$],..., T_n [$L_n..U_n$] **of** T
end

e à declaração do parâmetro formal pp como

var pp: T''

Assim como no caso de parâmetro por valor, os valores dos limites efetivos L_i, U_i são copiados quando da associação entre os parâmetros efetivo e formal; no entanto, os elementos do vetor não são copiados mas é passado o endereço deste como parâmetro efetivo. O pseudo-identificador "`Apontador_Componente`" e o operador de derreferenciação ↑ não devem ser usados para se fazer acesso aos elementos do vetor.

Exemplo: Considerando as declarações do exemplo anterior,

```
procedure SomaVarVet (var t: REAL; var p: array [L..U: INTEGER] of REAL);
    var i: INTEGER;
        s: REAL := 0.0;
begin
    for i := p.L to p.U do s := s + p[i] end;
    t := s
end SomaVarVet;
```

Na chamada `SomaVarVet(Soma,VetGrande)`, p.L = MINSHORTINT, p.U = MAXSHORTINT. No corpo do subprograma, `p[i]` pode ser visto como uma abreviação de `p."Apontador_Componente"↑[i]`. Também como no caso de parâmetros por valor, os limites do vetor se comportam como constantes cujos valores são definidos quando da associação entre os parâmetros efetivo e formal.

A especificação de parâmetros formais

$$p_1,p_2,\ldots,p_m\text{: } \textbf{array } [L_1..U_1\text{: } T_1,\ L_2..U_2\text{: } T_2,\ldots,\ L_n..U_n\text{: } T_n] \textbf{ of } T$$

é permitida e equivalente a

$$\begin{aligned} p_1&\text{: } \textbf{array } [L_1..U_1\text{: } T_1,\ L_2..U_2\text{: } T_2,\ldots,\ L_n..U_n\text{: } T_n] \textbf{ of } T;\\ p_2&\text{: } \textbf{array } [L_1..U_1\text{: } T_1,\ L_2..U_2\text{: } T_2,\ldots,\ L_n..U_n\text{: } T_n] \textbf{ of } T;\\ &\vdots\\ p_m&\text{: } \textbf{array } [L_1..U_1\text{: } T_1,\ L_2..U_2\text{: } T_2,\ldots,\ L_n..U_n\text{: } T_n] \textbf{ of } T; \end{aligned}$$

mas os tipos dos parâmetros p_i não são equivalentes entre si, conforme apresentado em 6.5.2.

6.2.2 Tipos Registros

Um registro é uma estrutura que apresenta um número fixo de componentes (zero ou mais), possivelmente de tipos diferentes. A declaração do tipo registro

especifica para cada componente, chamado *campo*, seu tipo e o identificador que o denota. O escopo desses identificadores de campo é local à definição do registro e o acesso a eles é feito através de *seletores* (designadores – seção 9.2.2 ou comando de escopo de registro – seção 9.3.5).

```
tipo_registro : record end
              | record seqüência_de_campos end
              ;
seqüência_de_campos : lista_de_campos fim_de_lista_de_campos
          | seqüência_de_campos ';' lista_de_campos fim_de_lista_de_campos
          ;
lista_de_campos : lista_de_identificadores ':' tipo_simples
                | case identificador ':' tipo_simples of
                       lista_de_variantes outros_casos_para_registro end
                | case tipo_simples of
                       lista_de_variantes outros_casos_para_registro end
                ;
fim_de_lista_de_campos : /* vazio */
                       | ';'
                       ;
lista_de_variantes : lista_de_variantes '|' variante
                   | variante '|'
                   | variante
                   ;
variante : lista_de_rótulos_de_casos ':' seqüência_de_campos
         ;
lista_de_rótulos_de_casos : lista_de_rótulos_de_casos ',' rótulo_de_casos
                          | rótulo_de_casos
                          ;
rótulo_de_casos : expressão_constante
                | expressão_constante '..' expressão_constante
                ;
outros_casos_para_registro : /* vazio */
                           | others seqüência_de_campos
                           ;
```

Uma seqüência de campos pode apresentar várias seções variantes. Cada seção deve ser precedida por um campo de marca[5] ou por um identificador qualificado

[5]Em inglês: *tag field*.

que dá o tipo da marca. O valor de um campo de marca indica qual variante está ativa em um determinado instante; um registro sem este campo permite a referência a qualquer das variantes de uma seção sem que haja a indicação de qualquer erro. Cada variante é identificada pelos *rótulos de casos*, que são constantes de tipo compatível para atribuição com o tipo do campo de marca (cf. 6.5), que deve ser um tipo simples: enumerado, subintervalo ou um dos tipos ordinais predefinidos ou sinônimos.

Em um rótulo de casos, *expressão_constante* é uma abreviação de *expressão_constante..expressão_constante*, sendo avaliada uma única vez. Assim, cada rótulo de caso define um subintervalo do tipo do campo de marca. Numa seção variante em um registro, esses subintervalos de rótulos de casos devem ser disjuntos.

Valores iniciais de campos de marca são indefinidos. A disposição dos componentes de um registro segue a ordem textual de sua declaração.

6.2.3 Tipos Conjuntos

Um tipo conjunto definido como **set of** T compreende todos os conjuntos de valores de seu *tipo base* T. Este deve ser um tipo simples: enumerado, subintervalo ou um dos tipos ordinais predefinidos ou sinônimos.

O *conjunto vazio*, representado por $'\{\}'$, é uma expressão constante do tipo UNIV_SET e é compatível para atribuição e para operação (cf. 6.5) com conjuntos de qualquer tipo. O tipo UNIV_SET é um tipo "conjunto-universal" que é usado para a compatibilização de tipos conjuntos e não é passível de utilização explícita por um programa em MC.

tipo_conjunto : **set of** *tipo_simples*
;

6.3 Tipos Apontadores

Objetos de um tipo apontador P assumem como valores apontadores para variáveis de um outro tipo T.[6]

O valor de um apontador pode ser obtido por uma chamada de um subprograma de alocação de memória dinâmica (por exemplo, através do procedimento predefinido NEW, do módulo SYSTEM) ou pela utilização da função ADDR, do módulo LOW_LEVEL.

[6]Um apontador equivale ao endereço de um objeto.

tipo_apontador : **pointer to** *tipo*
;

Uma variável do tipo apontador também pode receber o valor da expressão constante NIL, de tipo UNIV_POINTER, compatível para atribuição e para operação com qualquer tipo apontador (cf. 6.5), que indica que variável alguma está sendo apontada. NIL é um identificador penetrante da linguagem. O tipo UNIV_POINTER é um tipo "apontador-universal" que é usado para compatibilização de tipos apontadores e não é passível de utilização explícita por um programa em MC.

6.4 Identificadores de Processos

Uma variável de tipo predefinido PROCESSID é de tipo identificador de processo. Seu valor pode ser obtido como resultado da execução de um comando de ativação de processo (cf. 9.3.6) ou como o valor da expressão constante NO_PID, de tipo PROCESSID, que indica que nenhum processo está sendo referenciado pela expressão. NO_PID é um identificador penetrante da linguagem.

Através de variáveis de tipo identificadores de processo é que pode ocorrer a comunicação entre processos em MC (cf. 12).

6.5 Compatibilidade entre Tipos

Compatibilidade entre tipos em MC é um conceito que é considerado em quatro situações em que os objetos da linguagem são confrontados:

1. Compatibilidade para operação
2. Compatibilidade para atribuição
3. Compatibilidade para passagem de parâmetros
4. Compatibilidade para devolução de resultado de funções

Em algumas situações, expressões de MC produzem resultados cujos tipos são indefinidos. Um tipo indefinido nunca é compatível com qualquer outro tipo.

Como visto na seção 6.1, um tipo simples ou escalar é aquele que não apresenta componentes atômicos; compreende os tipos simples predefinidos (BOOLEAN, CHAR, SHORTINT, SHORTCARD, INTEGER, CARDINAL, REAL), o tipo UNIV_INTEGER, os tipos enumerados, subintervalos e seus sinônimos.

O *tipo base* de um conjunto ou subintervalo designa o tipo de seus elementos. Numa declaração **type** $T_s = T_o$, o *tipo raiz* de T_s é o tipo raiz de T_o. Se T_o for um tipo subintervalo, o tipo raiz de T_s é tipo base de T_o. Se T_s não tiver sido declarado como sinônimo de outro tipo, T_s será seu próprio tipo raiz.

6.5.1 Tipos Anônimos

Para cada especificação de tipo sem um identificador, isto é, um nome de tipo, MC associa um nome interno como se fosse feita uma nova declaração, chamada *declaração de tipo anônimo*. Note-se que esta introduz um *novo* identificador de tipo anônimo.

Exemplo: A seqüência de declarações a seguir

```
type TipoInteiro = INTEGER;
     Cadeia = array [1..MAXCADEIA] of CHAR;
     Identificação = record n: INTEGER; k: SHORTCARD end;

var i,j: INTEGER;
    k,l: TipoInteiro;
    s1: Cadeia;
    s2: array [1..MAXCADEIA] of CHAR;
    n1: Identificação;
    n2: record n: INTEGER; k: SHORTCARD end;
```

é considerada como

```
type TipoInteiro = INTEGER;
type MC$TA0001 = array [1..MAXCADEIA] of CHAR;
     Cadeia = MC$TA0001;
type MC$TA0002 = record n: INTEGER; k: SHORTCARD end;
     Identificação = MC$TA0002;
var i,j: INTEGER;
var k,l: TipoInteiro;
var s1: Cadeia;
type MC$TA0003 = array [1..MAXCADEIA] of CHAR;
var s2: MC$TA0003;
var n1: Identificação;
type MC$TA0004 = record n: INTEGER; k: SHORTCARD end;
var n2: MC$TA0004;
```

6.5.2 Tipos Sinônimos e Tipos Equivalentes

Dois tipos, T_1 e T_2, são *sinônimos* se:

- T_1 e T_2 são o mesmo identificador de tipo.
- T_2 é declarado como **type** $T_2 = T_1$.
- Existe um tipo T' que é sinônimo de T_1 e também de T_2.

Dois tipos, T_1 e T_2, são *equivalentes* se:

- T_1 é sinônimo de T_2 e nenhum deles é sinônimo de um vetor com limites abertos.
- T_1 e T_2 são equivalentes a tipos subintervalos de tipo base equivalentes, com os mesmos limites.
- T_1 e T_2 são equivalentes a tipos conjuntos com tipos base equivalentes.
- T_1 e T_2 são equivalentes a tipos vetores com limites definidos, com tipos de índices equivalentes e os tipos de componentes equivalentes.
- T_1 e T_2 são equivalentes a tipos apontadores para tipos equivalentes ou sinônimos.

6.5.3 Compatibilidade para Operação

Um operador n-ário define os tipos $T_1,\ldots,T_n$ de seus operandos. Alguns operadores podem ser *sobrecarregados*,[7] isto é, um mesmo símbolo (operador) pode denotar operações distintas, com uma seqüência de tipos de seus operandos para cada uma delas. Cada operador n-ário, aplicado a seus operandos, produz um valor de um certo tipo como resultado. O apêndice A traz as tabelas de operadores, os tipos de seus operandos e o tipo do resultado.

Para que uma operação seja semanticamente aceita em MC, os tipos de seus operandos, $T'_1,\ldots,T'_n$, devem ser *compatíveis para operação* com os tipos $T_1,\ldots,T_n$ aceitos pelos operadores, conforme descrito no apêndice A.

T'_i é compatível para operação com T_i se:

- T_i e T'_i são equivalentes.

[7] Em inglês: *overloaded operators*.

- T_i e T'_i são equivalentes a tipos subintervalos com o mesmo tipo raiz.
- T_i é equivalente ao tipo ADDRESS, definido no módulo LOW_LEVEL, e T'_i é equivalente a um tipo apontador.

6.5.4 Compatibilidade para Atribuição

Uma expressão compõe-se de operadores e operandos. Como visto na seção 6.5.3, cada operador n-ário define os tipos $T_1,\ldots,T_n$ de seus operadores. Respeitadas as regras de compatibilidade para operação, a expressão pode ser avaliada segundo as regras de prioridade entre os operadores (cf. 9.2.4) e um resultado é obtido, de tipo resultante T_e. A atribuição de um valor de tipo T_e a uma variável de tipo T_v é possível se T_e é *compatível para atribuição* com T_v.

T_e é compatível para atribuição com T_v se:

- T_e é compatível para operação com T_v.
- T_e é equivalente ao tipo SHORTINT (SHORTCARD) e T_v é equivalente ao tipo INTEGER (CARDINAL).
- T_e é do tipo UNIV_POINTER e T_v é equivalente a um tipo apontador.
- T_e é do tipo UNIV_SET e T_v é equivalente a um tipo conjunto.

Observações:

1. Note-se que um vetor com limites abertos não é compatível com qualquer tipo; portanto, não pode ser atribuído como uma unidade.
2. Quando da atribuição, serão verificadas as restrições impostas pelos tipos. Por exemplo, a atribuição de um valor maior que a constante MAXSHORTINT a uma variável de tipo SHORTINT levantará a exceção RANGE_ERROR, da mesma maneira que a atribuição de um valor que exceda os limites de uma variável de tipo subintervalo.

6.5.5 Compatibilidade para Passagem de Parâmetros

A discussão a seguir diz respeito à parte fixa de parâmetros de um subprograma. Quanto à parte variável de parâmetros, cf. 11.3.3. Como apresentado na seção 11.3, os parâmetros de um subprograma ou processo podem ser considerados em duas classes: *parâmetros por valor* e *parâmetros por variável.*

Conformidade

Sejam T_v um tipo vetor (com limites definidos ou abertos), com um índice de tipo T_i, e T_l o tipo dos identificadores de limites de um parâmetro formal de tipo vetor com limites abertos. Um vetor de tipo T_v é *semelhante a* ou *conforme com* um parâmetro formal de tipo vetor com limites abertos se todas as condições a seguir forem verificadas:

1. T_i é compatível para atribuição com T_l.
2. Os limites de T_i, quando definidos, pertencem ao intervalo fechado definido por T_l.
3. O tipo de componente de T_v é compatível para atribuição com o tipo de componente do parâmetro formal ou o tipo do componente de T_v é conforme com o tipo do componente do parâmetro formal.

Parâmetro por Valor

Sejam T_f o tipo do parâmetro formal e T_e o tipo do parâmetro efetivo correspondente. O tipo T_e é *compatível para passagem de parâmetros por valor* com T_f se:

- T_e é compatível para atribuição com T_f.
- T_f é do tipo vetor com limites abertos, T_e é um tipo vetor (com limites definidos ou abertos) e T_e é conforme com T_f.

Parâmetro por Variável

Sejam T_f o tipo do parâmetro formal e T_e o tipo do parâmetro efetivo correspondente. O tipo T_e é *compatível para passagem de parâmetros por variável* com T_f se:

- T_e e T_f são equivalentes.
- T_e e T_f são equivalentes a tipos subintervalos com o mesmo tipo raiz.
- T_f é do tipo vetor com limites abertos, T_e é um tipo vetor (com limites definidos ou abertos) e T_e é conforme com T_f.

6.5.6 Compatibilidade para Devolução de Resultado de Funções

Sejam T_f o tipo especificado no cabeçalho de uma função e T_r o tipo do comando "**return** *expressão*" da parte de expressões da função (*expressão* não pode ser vazio; pode haver mais de um comando *return* dessa forma).

T_r é *compatível para devolução de resultado de funções* com T_f se T_r é compatível para atribuição com T_f.

6.5.7 Funções de Transferência e Conversão

MC provê mecanismos para a explícita compatibilização de tipos, através do uso de funções de transferência de tipos e funções de conversão entre tipos, que podem ou não gerar código e levantar exceções em casos de erros.

As funções de transferência podem ser utilizadas se for importado o identificador *TypeTransferFunctions*, do módulo LOW_LEVEL, pois é uma facilidade de baixo nível. Os identificadores de funções podem ser quaisquer nomes de tipos, predefinidos ou definidos em um programa.

As funções de conversão devolvem a representação de seu parâmetro no tipo especificado por cada uma delas. A descrição das funções de conversão predefinidas no módulo SYSTEM se encontra no apêndice B.

Exemplos:

SHORTCARD(10) – informa que a constante 10 é do tipo SHORTCARD.

INTEGER(x) /* x: CARDINAL */ – resulta num valor do tipo INTEGER a partir de um tipo CARDINAL. Nesse caso, nenhuma conversão é realizada.

Capítulo 7

Variáveis

Uma declaração de variável introduz um ou mais identificadores de variável, associando-os a um tipo. Variáveis cujos identificadores aparecem na mesma lista são todas do mesmo tipo.[1]

declaração_de_variáveis : /* vazio */
    | *declaração_de_variáveis lista_de_identificadores_com_atributos* ':' *tipo* ';'
    | *declaração_de_variáveis lista_de_identificadores_com_atributos* ':' *tipo*
        ':=' *expressão_para_valor_inicial* ';'
    ;
lista_de_identificadores_com_atributos : *identificador_com_atributos*
    | *lista_de_identificadores_com_atributos* ';' *identificador_com_atributos*
    ;

Na declaração de uma variável podem ser especificados atributos, que são uma facilidade de baixo nível (por exemplo, especificação de um endereço para a variável ou um registrador de máquina que deva estar associado a ela). Para usar essa facilidade, deve ser importado o identificador *SpecialAttributes* do módulo LOW_LEVEL. A forma de especificação dos atributos é dependente de implementação.

Exemplos:

```
var
    Palavra_de_Estado [0FEh /* endereço */] : BYTE;
    Registrador_AX [AX /* registrador de máquina */] : CARDINAL;
```

[1]Embora sejam do mesmo tipo, não necessariamente são de tipos equivalentes. Por exemplo, considere o caso de vetores com limites abertos.

Uma variável é criada quando da ativação do bloco onde foi declarada e destruída quando da desativação do mesmo. Assim, diz-se que ela é *automática.*[2] Variáveis automáticas podem receber um valor inicial quando o bloco onde foram declaradas é ativado.

```
expressão_para_valor_inicial : expressão
                             | expressão_estruturada
                             ;
expressão_estruturada : '(' componente_de_expressão_estruturada ')'
                      ;
componente_de_expressão_estruturada : expressão
                | componente_de_expressão_estruturada ',' expressão
                | componente_de_expressão_estruturada ',' expressão_estruturada
                ;
```

Se a variável é de tipo escalar, conjunto ou apontador, um valor inicial lhe pode ser atribuído. O tipo da expressão para valor inicial, T_e, deve ser compatível para atribuição com o tipo da variável declarada, T_v. Se a variável é estruturada, de tipos registro ou vetor com limites definidos, valores iniciais podem ser atribuídos a todos os seus componentes, através de uma *expressão estruturada*, que é uma lista de expressões ou expressões estruturadas entre os símbolos '(' e ')'. A correspondência entre cada expressão componente da expressão estruturada e cada componente da variável estruturada é um-a-um, segundo sua ordem textual. O número de componentes de ambas deve ser o mesmo e o tipo resultante de cada expressão componente deve ser de tipo compatível para atribuição com o tipo do componente correspondente. Se a estrutura contém subestruturas, a mesma regra se aplica recursivamente aos seus componentes.

As expressões para valor inicial são calculadas em tempo de execução[3] e sua ordem de avaliação segue a ordem textual da declaração das variáveis. Um erro na avaliação da expressão ou na atribuição à variável pode levantar uma exceção.

Uma declaração da forma:

$$\textbf{var}\ \mathrm{id}_1,\mathrm{id}_2,\ldots,\mathrm{id}_n\colon\ \mathrm{T} := \mathrm{E}$$

é funcional mas não sintaticamente equivalente à declaração:

[2] Da mesma forma como é considerada em C. Cf. [KernRit 78].

[3] O compilador pode decidir esta questão. No caso de expressões constantes e variáveis globais, seria razoável fazer suas avaliações em tempo de compilação.

```
var   id1: T := E;
      id2: T := E;
      :
      idn: T := E;
```

Neste caso, no entanto, a expressão E é avaliada *apenas* uma vez.

Variáveis que não recebem um valor inicial estão *indefinidas*. O acesso a uma variável indefinida para utilização de seu valor levanta a exceção NO_VALUE_ERROR.

Variáveis dinâmicas podem ser criadas através do procedimento predefinido NEW, do módulo SYSTEM, que aloca[4] uma seqüência de posições na memória dinâmica, cujo tamanho é dependente da variável a ser criada, e devolve no seu argumento, de tipo apontador, o endereço da primeira posição reservada. Vetores com limites abertos (cf. 6.2.1) são criados através do mesmo procedimento, sendo passados como argumentos, além do apontador, os limites de cada vetor com limites abertos da variável dinâmica a ser criada. Cada par de limites passado como argumento deve ser de tipo compatível para atribuição com o tipo do par de limites correspondente do vetor com limites abertos. A variável criada dessa maneira terá definidos os componentes que dão os limites do vetor.

As variáveis dinâmicas não seguem a disciplina de variáveis automáticas. Uma vez criadas em um bloco, continuam a existir quando da desativação do mesmo. Deixarão de existir quando houver a desalocação da área anteriormente reservada para elas, através do procedimento predefinido DISPOSE, do módulo SYSTEM.

O procedimento predefinido NEW utiliza a função padrão de alocação de memória, ALLOCATE_MEMORY, definida no módulo LOW_LEVEL; de maneira semelhante, o procedimento predefinido DISPOSE utiliza o procedimento padrão para liberação de memória, DEALLOCATE_MEMORY, também definida no módulo LOW_LEVEL (cf. apêndice D). As funções ALLOCATE_MEMORY e DEALLOCATE_MEMORY podem ser redefinidas no programa; se isto ocorrer, o compilador as utilizará, ao invés das especificadas no módulo LOW_LEVEL.

---

[4] *Alocar* (em inglês: *allocate*): atribuir determinado recurso ou espaço de memória para o uso na execução de um programa ou rotina específicos ou para o armazenamento de dados ou arquivos. *Desalocar* (em inglês: *deallocate*): liberar um recurso que já esteja alocado ou atribuído a uma tarefa específica. *A. H. Fragomeni. Dicionário Enciclopédico de Informática. Prefácio de Antônio Houaiss. Editora Campus/Livraria Nobel S/A.*

Capítulo 8

Blocos e Escopo de Identificadores

Blocos são a base para a implementação de algoritmos em MC. As regras de escopo determinam onde um identificador que é introduzido em um determinado lugar pode ser usado, baseando-se na estrutura estática (textual) do programa. As regras de *ativação* determinam que entidade (por exemplo, uma variável) é denotada por um particular identificador, baseando-se na estrutura dinâmica (isto é, dependendo da execução) do programa.

8.1 Blocos

Um bloco consiste de uma parte de declarações e uma parte de expressões.

bloco : *parte_de_declarações parte_de_expressões*
;

parte_de_declarações : /* vazio */
| *parte_de_declarações* **const** *declaração_de_constantes*
| *parte_de_declarações* **type** *lista_de_tipos_sinônimos*
| *parte_de_declarações* **var** *declaração_de_variáveis*
| *parte_de_declarações* **exception** *declaração_de_exceções*
| *parte_de_declarações declaração_de_processo*
| *parte_de_declarações declaração_de_subprograma*
| *parte_de_declarações declaração_de_módulo*
| *parte_de_declarações cabeçalho_de_processo*
| *parte_de_declarações cabeçalho_de_subprograma*
;

parte_de_expressões : **begin** *seqüência_de_expressões*
tratamento_de_exceções **end**
;

8.2 Declarações e Regras de Escopo

Todo identificador que ocorre em um programa deve ser introduzido por uma declaração, a menos que seja um identificador-padrão. Este é considerado predeclarado e é válido em todas as partes do programa, sendo chamado, por isso, de *penetrante*. As declarações também especificam as propriedades permanentes de um objeto (por exemplo, se é uma constante, um tipo, uma variável, um módulo, etc.). Assim, um identificador é usado para referenciar o objeto associado.

Define-se *escopo* de uma declaração a região do programa na qual a declaração tem efeito. As regras de escopo são dadas a seguir:

1. O escopo de uma declaração se estende ao bloco de módulo, processo ou subprograma ao qual a declaração pertence. Diz-se que o objeto declarado é local ao bloco. Um identificador não pode ser redeclarado em um mesmo escopo.

2. Se um identificador x definido por uma declaração D_1 é usado em uma outra declaração D_2, então D_1 deve preceder D_2 textualmente, exceto no caso da regra 3.

3. Uma declaração de tipo **pointer to** T pode preceder a declaração do tipo T se ambas ocorrem no mesmo bloco.

4. Se um identificador definido em um módulo M_1 é exportado, seu escopo se expande sobre o bloco que contém M_1. Se M_1 é uma unidade de compilação, o escopo é estendido a todas as unidades que importam M_1.

5. Identificadores de campos de uma declaração de registro (cf. 6.2.2) são válidos apenas em designadores de campos (cf. 9.2.2) e em comandos de escopo de registro (cf. 9.3.5) que se referem à variável daquele tipo de registro. Um comando de escopo de registro abre um novo escopo; múltiplos comandos de escopo de registro encaixados abrem escopos encaixados sintaticamente.

6. Os identificadores de constantes de um tipo enumerado têm escopo de declaração local. A importação de um tipo enumerado introduz todos os identificadores de constantes no escopo do módulo que o importa.

7. O escopo de um parâmetro formal é o bloco do processo ou subprograma onde a declaração de parâmetro se encontra.

8. Todo identificador (exceto penetrante) que denota um objeto não local a um módulo deve ser importado. Seu uso pode ser qualificado para evitar conflitos com os identificadores locais de um bloco ou com outros identificadores importados (cf. 4.6).

9. Se um identificador penetrante é redeclarado em algum bloco, seu novo significado se mantém nas declarações internas a este.

8.3 Ativação

A ativação de um módulo de programa, um processo ou um subprograma (procedimento ou função) é a ativação de seu bloco. Contém as seguintes entidades, que existem até que a ativação termine:

- Um algoritmo que é especificado pela parte de expressões do bloco; inicia-se quando um bloco é ativado e seu término causa a desativação do bloco.

- Uma variável para cada identificador de variável que é local ao bloco, incluindo parâmetros formais; quando o algoritmo é iniciado, a variável está indefinida a menos que seja um parâmetro ou que, na sua declaração, haja uma expressão que lhe dê um valor inicial (cf. 7). Nesse caso, antes de iniciado o algoritmo, essa expressão é avaliada e atribuída.

- Um processo, procedimento ou função para cada identificador de processo, procedimento ou função que são locais ao bloco.

Um módulo local é ativado quando o bloco que o contém é ativado e seus objetos locais existem enquanto bloco estiver ativo. A parte de expressões do bloco de um módulo local é executada quando da sua ativação; a ordem de execução segue a ordem textual de declaração dos módulos.

As partes de expressões de módulos de programa e de módulos de implementação são executadas quando da ativação do programa, seguindo a ordem de inclusão dos mesmos.

Capítulo 9

Expressões

MC é uma linguagem de expressões. A avaliação de uma expressão resulta num valor (mesmo que seja indefinido) com, possivelmente, efeitos laterais, ou levanta uma exceção. O resultado e o tipo de uma seqüência de expressões correspondem ao resultado e tipo da última expressão da seqüência.

As expressões podem ser divididas em dois grupos: *expressões próprias* e *comandos* e podem ser antecedidas por um rótulo.

```
seqüência_de_expressões : seqüência_de_expressões ';' expressão_com_rótulo
                        | expressão_com_rótulo
                        ;
expressão_com_rótulo : rótulo expressão
                     | expressão
                     ;
expressão : expressão_própria
          | comando
          ;
```

9.1 Rótulos

A declaração de um rótulo associa um identificador a uma expressão e seu escopo é o bloco onde foi declarado, não podendo ser redefinido no mesmo escopo. Rótulos podem ser referenciados pelos comandos de desvio (cf. 9.3.4).

```
rótulo : '<<' identificador '>>'
       ;
```

9.2 Expressões Próprias

Uma expressão própria é uma construção que define as regras para a obtenção de valores de variáveis e geração de novos valores pela aplicação de operadores. Consiste de operadores, operandos e chamadas de funções e procedimentos. Parênteses podem ser usados para expressar associações específicas entre operadores e operandos.

Expressões são avaliadas da esquerda para a direita, segundo sua ordem textual e regras de prioridade entre os operadores.

expressão_própria : *expressão_disjuntiva*
;
expressão_disjuntiva : *expressão_conjuntiva*
| *expressão_disjuntiva operador_lógico_disjuntivo expressão_conjuntiva*
;
expressão_conjuntiva : *expressão_conjuntiva* **and** *relação*
| *relação*
;
relação : *expressão_simples*
| *expressão_simples operador_relacional expressão_simples*
;
expressão_simples : *operador_aditivo termo outros_termos*
| *termo outros_termos*
;
outros_termos : /* vazio */
| *outros_termos operador_aditivo termo*
;
termo : *fatores*
;
fatores : *fatores operador_multiplicativo fator*
| *fator*
;
fator : **not** *primário*
| *primário*
;
operador_lógico_disjuntivo : **or** | **xor**
;
operador_relacional : '=' | '#' | '<' | '<=' | '>' | '>=' | **in** | **notin**
;
operador_aditivo : '+' | '−' | '&'

;

operador_multiplicativo : '*' | '/' | '%'

;

9.2.1 Expressões Primárias

São expressões primárias:

- *Constantes*: Uma constante numérica inteira é de um dos tipos simples predefinidos SHORTINT, SHORTCARD, INTEGER, CARDINAL, caso seja o resultado da aplicação da função de transferência correspondente (cf. apêndice B) a alguma constante numérica inteira; caso contrário, seu tipo é UNIV_INTEGER. Uma constante numérica real é do tipo REAL. Uma constante ASCII é um caractere entre aspas simples ou o resultado da função predefinida CHR e seu tipo é CHAR. Uma cadeia de caracteres é de tipo anônimo (**array** [1..L] **of** CHAR, onde L é o tamanho da cadeia). Os identificadores penetrantes TRUE e FALSE são do tipo BOOLEAN. A constante NIL é do tipo UNIV_POINTER e a constante NO_PID, do tipo PROCESSID.

- *Conjuntos*: Um conjunto é denotado por uma lista de elementos entre os símbolos '{' e '}', podendo ser antecedido por um identificador qualificado que especifica o seu *tipo base*. Este deve ser um tipo enumerado, subintervalo ou um dos tipos predefinidos ou sinônimos, excetuando-se o conjunto vazio, representado por '{}', que é do tipo UNIV_SET. Cada elemento do conjunto deve ser de tipo compatível para atribuição com o tipo base, se este estiver presente. Caso contrário, este tipo é determinado pelos elementos que compõem o conjunto, da seguinte maneira:

  1. As expressões são de tipos resultantes equivalentes: o tipo base será qualquer um dos tipos resultantes.
  2. As expressões são de tipos resultantes equivalentes a SHORTCARD e CARDINAL: o tipo base será CARDINAL.
  3. As expressões são de tipos resultantes equivalentes a SHORTINT e INTEGER: o tipo base será INTEGER.
  4. Excluídos os casos anteriores, as expressões são de tipos resultantes inteiros; o tipo base será UNIV_INTEGER.

- *Chamada de Função ou Procedimento*: A chamada de um subprograma, do tipo função ou procedimento, é denotada pelo identificador qualificado

do subprograma seguido por uma lista, possivelmente vazia, de expressões entre parênteses, chamadas *parâmetros efetivos* (veja na seção 11 a especificação da forma e ordem de avaliação dos parâmetros). Os tipos dos parâmetros efetivos devem ser compatíveis para passagem de parâmetros com os tipos dos parâmetros formais correspondentes, de acordo com seus respectivos mecanismos de passagem (cf. 6.5.5). Uma função retorna um valor cujo tipo é o especificado na sua declaração; um procedimento retorna um resultado de tipo e valor indefinidos.

- *Expressão de Incremento ou Decremento*: Um designador (expressão) pode ser incrementado ou decrementado,[1] antes ou após sua avaliação (cf. 9.2.3).

- *Expressão entre Parênteses*: O tipo e valor de uma expressão entre parênteses são os mesmos da expressão sem parênteses, que são usados para alterar a ordem de avaliação.

```
primário : número
         | cte_ASCII
         | cadeia
         | conjunto
         | chamada_de_função_ou_procedimento
         | expressão_de_incremento_ou_decremento
         | '(' expressão ')'
         ;
cte_ASCII : ''' caractere '''
          ;
conjunto : designador_ou_ident_qualificado '{' lista_de_elementos '}'
         | '{' lista_de_elementos '}'
         ;
lista_de_elementos : lista_de_elementos ',' elemento
                   | elemento
                   ;
elemento : expressão '..' expressão
         | expressão
         ;
chamada_de_função_ou_procedimento :
         | designador_ou_ident_qualificado '(' lista_de_parâmetros_efetivos ')'
         ;
```

[1] *Decrementar* (em inglês: *decrement*): diminuir (de um) o valor de um número. Antônimo: *incrementar* (em inglês: *increment*). A. H. Fragomeni, op. cit.

lista_de_parâmetros_efetivos : *lista_de_parâmetros_efetivos* ',' *expressão*
| *expressão*
;

9.2.2 Designadores

De uma forma geral, operandos são denotados por *designadores*. Um designador consiste de um identificador que se refere à constante ou variável a ser designada. Esse identificador pode ser qualificado por identificadores de módulos (sendo chamado *identificador qualificado*) e seguido por seletores, se o objeto designado é um elemento de uma estrutura. Se a estrutura é um vetor V, então o designador V[e] denota o componente de V cujo índice é o valor corrente da expressão e. O tipo da expressão e deve ser compatível para atribuição com o tipo de índice de V. Um designador da forma $V[e_1, e_2, \ldots, e_n]$ é uma abreviação de $V[e_1][e_2]\ldots[e_n]$. Se o resultado da expressão e não está no intervalo definido na declaração do tipo de índice de V, é levantada a exceção INDEX_ERROR. Se a estrutura é um registro R, então o designador R.c denota o campo c de R. O designador P↑ denota a variável que é referenciada pelo apontador P. Se o apontador P contém o valor NIL, P↑ levanta a exceção ACCESS_ERROR. Se o objeto designado é uma variável, o designador se refere ao seu valor corrente.

designador_ou_ident_qualificado : *identificador*
| *designador_ou_ident_qualificado* '.' *identificador*
| *designador_ou_ident_qualificado* '[' *lista_de_expressões* ']'
| *designador_ou_ident_qualificado* '↑'
;
lista_de_expressões : *lista_de_expressões* ',' *expressão*
| *expressão*
;

9.2.3 Expressões de Incremento e Decremento

Uma expressão de incremento apresenta um operador de incremento prefixo ou posfixo e um designador, seu operando. Uma expressão de decremento apresenta um operador de decremento prefixo ou posfixo e um designador, seu operando. Os operadores de incremento e decremento, no entanto, são opcionais.

expressão_de_incremento_ou_decremento : *designador_ou_ident_qualificado*
| '++' *designador_ou_ident_qualificado*
| '--' *designador_ou_ident_qualificado*

| *designador_ou_ident_qualificado* '++'
| *designador_ou_ident_qualificado* '--'
;

O operador de incremento ('++') se prefixo ao designador, resulta neste valor incrementado de 1, atualizando-o como efeito lateral. Se posfixo ao designador, resulta no seu valor corrente e, posteriormente (cf. as funções equivalentes a seguir), incrementam-no de 1, atualizando-o. Efeitos semelhantes ocorrem para o operador de decremento (nesse caso, é feito o decremento de 1). Esses operadores só podem ser aplicados a parâmetros e variáveis não estruturados.

As expressões $++x$ e $x++$ são equivalentes funcionalmente às seguintes funções, onde T denota um tipo inteiro:

```
function Pre_Incremento (var x: T): T;
        /* ++x */
begin
    x := x + 1;
    return x;
    /* ou return x := x + 1 */
end Pre_Incremento;

function Pos_Incremento (var x: T): T;
        /* x++ */
    var r: T;
begin
    r := x; x := x + 1;
    return r;
    /* ou return (x := x + 1) - 1 */
end Pos_Incremento;
```

Equivalências análogas podem ser feitas para as expressões $--x$ e $x--$.

9.2.4 Operadores

A sintaxe de expressões especifica a precedência de acordo com quatro classes de operadores. Os operadores de incremento e decremento têm maior precedência. Seguem os operadores binários multiplicativos, aditivos, relacionais e lógicos, com prioridades decrescentes. Seqüências de operadores binários de mesma precedência são executados da esquerda para a direita. Os operadores também

Operador	Operação
+	identidade
−	inversão de sinal

Tabela 9.1: Operadores Aritméticos Unários

Operador	Operação
+	adição
−	subtração
*	multiplicação
/	divisão
%	resto

Tabela 9.2: Operadores Aritméticos Binários

podem ser classificados de acordo com os tipos de seus operandos e do resultado. Neste sentido, diz-se que os operadores podem ser sobrecarregados (cf. 6.5.3).

Numa subexpressão com operadores binários, a subexpressão à esquerda é avaliada primeiramente.

Operadores Aritméticos

Esses operadores se aplicam a operandos de tipo numérico e sinônimos (exceto o operador %, que não se aplica ao tipo REAL). Seus operandos devem ser de tipo compatível para operação com os definidos para esses operadores no apêndice A. A tabela 9.1 traz a descrição dos operadores unários e a tabela 9.2, dos operadores binários.

A divisão e o resto são definidos pela relação a seguir, para x e y de algum tipo inteiro:

$$x = (x/y) * y + x\%y$$

Para $y > 0$, x % y **in** $[0..y-1]$ é verdadeiro, independente do sinal de x. Para $y < 0$, x % y **in** $[y+1..0]$ é verdadeiro, independente do sinal de x.

A avaliação de um termo na forma x/y ou $x\%y$ levanta a exceção DIVIDE_ERROR se y é zero. Todos os operadores podem levantar as exceções OVERFLOW e UNDERFLOW.

Operador	Operação	Significado
not p	negação	Se p então FALSE senão TRUE
p **and** q	conjunção lógica	Se p então q senão FALSE
p **or** q	disjunção lógica inclusiva	Se p então TRUE senão q
p **xor** q	disjunção lógica exclusiva	Se $p = q$ então FALSE senão TRUE

Tabela 9.3: Operadores Lógicos

Operador	Operação
+	união
−	diferença
*	intersecção
/	diferença simétrica

Tabela 9.4: Operadores para Conjuntos

Operadores Lógicos

Os operadores lógicos se aplicam a operandos de tipo compatível para operação com o tipo BOOLEAN, conforme mostra o apêndice A, e resultam num valor de tipo BOOLEAN. Seu significado é mostrado na tabela 9.3, onde p e q são expressões de tipo BOOLEAN.

Operadores para Conjuntos

Os operadores para conjuntos se aplicam a expressões de tipos conjuntos com tipos base equivalentes, conforme apresentado na seção 6.5.3. A tabela 9.4 descreve esses operadores e o apêndice A traz o seu significado e discute os tipos dos operandos e o tipo do resultado.

Operadores Relacionais

Os operadores relacionais são apresentados na tabela 9.5 e se aplicam aos tipos simples predefinidos SHORTINT, SHORTCARD, INTEGER, CARDINAL, BOOLEAN, CHAR, REAL; ao tipo UNIV_INTEGER; ao tipo PROCESSID; a tipos enumerados, subintervalos, conjuntos, apontadores e sinônimos, conforme mostra o apêndice A. O resultado de operações relacionais é do tipo BOOLEAN.

Os operadores <= e >=, quando aplicados a conjuntos, denotam inclusão (imprópria). A expressão $x <= y$, onde x e y são de tipos conjuntos compatíveis para operação, resulta em TRUE se x é um subconjunto de y.

Operador	Relação	Tipos dos Operandos
=	igual	simples, apontador, conjunto, identificador de processo
#	diferente	simples, apontador, conjunto, identificador de processo
<	menor	simples
<=	menor ou igual (inclusão de conjuntos)	simples, conjunto
>	maior	simples
>=	maior ou igual (inclusão de conjuntos)	simples, conjunto
in	pertencente a	ordinal e conjunto
notin	não pertencente a	ordinal e conjunto

Tabela 9.5: Operadores Relacionais

Operadores para Caracteres

O operador & concatena cadeias de caracteres. Seus operandos devem ser compatíveis para operação com os tipos definidos para esse operador, conforme o apêndice A. O resultado é uma cadeia de caracteres, de tipo anônimo **array** [1..L_1+L_2] **of** CHAR, onde L_1 e L_2 são os tamanhos dos operandos (isto é, o número de caracteres de cada cadeia).

9.3 Comandos

Comandos denotam ações como atribuição, chamada e retorno de subprogramas, execução condicional e repetitiva, etc. Resultam num valor, às vezes indefinido.

comando : *comando_de_atribuição*
| *comando_condicional*
| *comando_repetitivo*
| *comando_de_desvio*
| *comando_de_escopo_de_registro*
| *comando_de_ativação_de_processo*
| *comando_de_levantamento_de_exceção*
| *comando_vazio*
;

Comandos em seqüência são executados sucessivamente, a menos que uma exceção seja levantada ou que seja executado um comando de desvio.

9.3.1 Comando de Atribuição

Uma atribuição substitui o valor corrente de uma variável, simples ou estruturada, por um novo valor, resultado da avaliação de uma expressão ou expressão estruturada (cf. 7).

comando_de_atribuição : *designador_ou_ident_qualificado* ':=' *expressão*
| *designador_ou_ident_qualificado* ':=' *expressão_estruturada*
;

O designador à esquerda do operador de atribuição (':=') denota uma variável, simples ou estruturada. A discussão sobre a expressão ou expressão estruturada à direita deste operador deve ser vista na seção 7. Quando da atribuição, a não obediência às restrições impostas pela variável ou por um componente de uma variável, causará o levantamento da exceção correspondente à restrição violada (cf.apêndice C).

Neste comando, o designador à esquerda do operador de atribuição é avaliado antes da expressão à sua direita; efeitos laterais se propagam nessa ordem.

O resultado do comando de atribuição é o valor da expressão ou expressão estruturada e o tipo é o mesmo do designador da variável a ser atribuída.

9.3.2 Comandos Condicionais

Um comando condicional seleciona para execução uma de suas expressões componentes.

comando_condicional : **if** *expressão*
then *seqüência_de_expressões comando_if_encadeado*
else *seqüência_de_expressões* **end**
| **if** *expressão* **then** *seqüência_de_expressões comando_if_encadeado* **end**
| **case** *expressão* **of** *lista_de_casos outros_casos_de_seleção* **end**
;
comando_if_encadeado : /* vazio */
| *comando_if_encadeado* **elsif** *expressão* **then** *seqüência_de_expressões*
;
lista_de_casos : *lista_de_casos* '|' *caso fim_de_casos*
| *caso fim_de_casos*

```
                ;
fim_de_casos : /* vazio */
             | '|'
             ;
caso : lista_de_rótulos_de_casos ':' seqüência_de_expressões
     ;
outros_casos_de_seleção : /* vazio */
                        | others seqüência_de_expressões
                        ;
```

Comando if

As expressões que seguem os símbolos **if** e **elsif** são de tipos compatíveis para operação com o tipo BOOLEAN. São avaliadas segundo sua ordem textual até que uma delas resulte no valor TRUE. A seqüência de expressões associada à primeira expressão que resultar no valor TRUE será executada completando, assim, o comando *if*. Se uma cláusula **else** está presente, sua seqüência de expressões associada é executada se e somente se todas as expressões lógicas resultam no valor FALSE.

O resultado e o tipo deste comando serão os mesmos da seqüência de expressões executada se todas as seqüências de expressões presentes no comando forem compatíveis para operação entre si. Se não houver tal compatibilidade ou se a cláusula *else* não estiver presente, o resultado e o tipo do comando *if* serão indefinidos.

Comando case

Este comando especifica a seleção e execução de uma seqüência de expressões de acordo com o valor de uma expressão, chamada *expressão-índice*. Primeiro, a expressão-índice, que segue o símbolo **case**, é avaliada. Então, a seqüência de expressões que apresenta, na lista de rótulos de casos que a antecede, um valor igual ao valor da expressão-índice, é executada e o comando termina. O tipo da expressão-índice deve ser compatível para operação com um tipo simples, exceto o tipo REAL e todos os rótulos de casos devem ser compatíveis para operação com este tipo. Um rótulo de caso é uma constante e não pode aparecer em mais de uma lista de rótulos de um comando *case*.

Se o valor da expressão-índice não ocorre como um rótulo de caso, então a seqüência de expressões que segue o símbolo **others** é executada, se esta cláusula estiver presente; caso contrário, a exceção SELECT_ERROR é levantada.

O resultado e o tipo deste comando serão os mesmos da seqüência de expressões executada se todas as seqüências de expressões presentes no comando forem compatíveis para operação entre si. Se não houver tal compatibilidade o resultado e o tipo do comando *case* serão indefinidos.

9.3.3 Comandos Repetitivos

Um comando repetitivo especifica que uma seqüência de expressões deve ser executada repetidamente, zero ou mais vezes. Sua execução termina quando a especificação de iteração da malha é atingida ou quando um comando de desvio que causa o término da malha (cf. 9.3.4) é executado.

comando_repetitivo : **while** *expressão* **do** *seqüência_de_expressões* **end**
| **repeat** *seqüência_de_expressões* **until** *expressão*
| **loop** *seqüência_de_expressões* **end**
| **for** *identificador* ':=' *expressão* **to** *expressão*
expressão_de_incremento **do** *seqüência_de_expressões* **end**
;
expressão_de_incremento : /* vazio */
| **by** *expressão*
;

Comando while

Este comando especifica a execução repetida da seqüência de expressões associada (entre os símbolos **do** e **end**), dependendo do valor da expressão que segue o símbolo **while**, que deve ser compatível para operação com o tipo BOOLEAN. Esta expressão é avaliada antes de cada execução subseqüente da seqüência de expressões, que deixa de ser realizada tão logo a expressão resulte no valor FALSE. Note-se que a malha pode ser executada zero ou mais vezes. O resultado e o tipo do comando *while* são indefinidos.

Comando repeat

Este comando especifica a execução repetida da seqüência de expressões associada (entre os símbolos **repeat** e **until**) dependendo do valor da expressão que segue o símbolo **until**, que deve ser compatível para operação com o tipo BOOLEAN. Esta expressão é avaliada após cada execução da seqüência de expressões e a repetição termina tão logo resulte no valor TRUE. Assim, a malha é executada no mínimo uma vez. O resultado e o tipo do comando *repeat* são indefinidos.

Comando loop

Este comando especifica a execução repetida da seqüência de expressões associada (entre os símbolos **loop** e **end**). A repetição termina quando da execução dos comandos de desvio *return* ou *exit* (cf. 9.3.4). O resultado e o tipo do comando *loop* são indefinidos.

Comando for

Este comando indica que a seqüência de expressões associada (entre os símbolos **do** e **end**) é executada repetidamente enquanto uma progressão de valores é atribuída a uma variável, denominada *variável de controle*. Esta não pode ser estruturada e seu valor não pode ser alterado pela seqüência de expressões. A expressão que segue o símbolo **for**, denominada *expressão de valor inicial*; a que segue o símbolo **to**, denominada *expressão de valor final* e a que segue o símbolo **by**, denominada *expressão de incremento*, são avaliadas apenas uma vez, antes de se começar a repetição. A expressão de valor inicial deve ser de tipo compatível para atribuição com o tipo da variável de controle; as expressões de valor final e de incremento devem ser compatíveis para operação com o tipo dessa variável.

O comando

for v := A **to** B **do** E **end**

expressa que a seqüência de expressões E será executada com v recebendo, sucessivamente, os valores da seqüência A,succ(A),succ^2(A),...,succ^n(A), onde *succ(x)* é uma função predefinida no módulo SYSTEM que devolve o sucessor de x no conjunto de valores definido pelo seu tipo, *succ^i(x)* denota a aplicação sucessiva da função succ i vezes e *succ^n(A)* é o último termo que não excede B. Neste caso, v deve ser um tipo enumerado, inteiro ou um dos tipos predefinidos CHAR ou BOOLEAN.

O comando

for v := A **to** B **by** C **do** E **end**

expressa que a seqüência de expressões E será executada com v recebendo, sucessivamente, os valores da seqüência crescente ou decrescente A,A+C,A+2C,...,A+nC, onde A+nC é o último termo que não excede B. Neste caso, v deve ser um tipo numérico.

Após a execução do comando *for*, a variável de controle permanece com o seu valor corrente. O resultado e o tipo do comando são indefinidos.

9.3.4 Comandos de Desvio

Um comando de desvio causa o término de um comando repetitivo ou o desvio para a avaliação de sua expressão de controle ou, ainda, o término da execução de um subprograma ou processo.

comando_de_desvio : **continue**
| **continue** *rótulo*
| **return** *expressão*
| **exit**
| **exit** *rótulo*
;

Comando continue

Este comando causa o desvio para a avaliação da expressão de controle do comando repetitivo mais interno ou, caso venha acompanhado de um rótulo, do comando repetitivo que é antecedido por tal rótulo. Se se tratar do comando *loop*, no qual não há expressão de controle, o desvio se dará para a primeira expressão da seqüência. O resultado e o tipo do comando *continue* são indefinidos.

Comando return

Este comando indica o término da execução de um subprograma (procedimento ou função) ou processo e a expressão que segue o símbolo **return**, possivelmente vazia, especifica o valor retornado como resultado.

Se o subprograma for uma função, sua expressão não pode ser vazia e seu tipo deve ser compatível para devolução de resultado de função (cf. 6.5.6) com o tipo da função na qual o comando *return* ocorre. O tipo e o resultado do comando, neste caso, serão os mesmos da expressão.

Em procedimentos e processos, a expressão que segue o símbolo **return** deve ser vazia, sendo o tipo e resultado do comando indefinidos. O significado deste comando em processos é discutido na seção 12.

Comando exit

Este comando causa o término do comando repetitivo mais interno, a menos que venha seguido de um rótulo que está associado a um comando repetitivo; neste caso, o comando é interrompido. O resultado e o tipo do comando *exit* são indefinidos.

9.3.5 Comando de Escopo de Registro

Este comando especifica um designador de variável estruturada de tipo registro e uma seqüência de expressões. Nesta seqüência, a qualificação dos identificadores de campos pode ser omitida.

comando_de_escopo_de_registro :

with *expressão* **do** *seqüência_de_expressões* **end**

;

A expressão deve denotar uma variável estruturada de tipo registro e é avaliada apenas uma vez, antes da execução da seqüência de expressões. O comando *with* abre um novo escopo. Seu resultado e tipo são os da seqüência de expressões associada.

Não há limite para o número de comandos de escopo de registro que podem ser encaixados sintaticamente. As regras de escopo (cf. 8.2) valem para cada novo escopo que é aberto.

9.3.6 Comando de Ativação de Processo

O comando *start* inicia a execução de um processo (cf. 12). Este é denotado por um identificador qualificado seguido por uma lista, possivelmente vazia, de expressões, chamadas *parâmetros efetivos.*

comando_de_ativação_de_processo :

start *especificação_de_atributos designador_ou_ident_qualificado*
'(' *lista_de_parâmetros_efetivos* ')'

;

Quando um processo é ativado, podem ser especificados alguns atributos como, por exemplo, a prioridade de sua execução, espaço de memória a ser reservado, etc. Para usar essa facilidade de baixo nível, deve ser importado o identificador *SpecialAttributes* do módulo LOW_LEVEL. A forma de especificação dos atributos é dependente da implementação.

Exemplo:

p := **start** [ACTIVATION_PRIORITY := 5, WSMAX := 1024] Monitor ( );

A falha na ativação de um processo levanta a exceção PROCESS_ACTIVATION_ERROR. Se o comando é bem sucedido, a ativação devolve uma identificação para o processo criado. O resultado do comando é esta identificação de processo, de tipo predefinido PROCESSID. Através de tal identificação é que pode ocorrer a comunicação entre processos em MC (cf. 12).

9.3.7 Comando de Levantamento de Exceção

Uma exceção (cf. 10) pode ser levantada por alguma condição especial ou explicitamente pelo comando de levantamento de exceção.

comando_de_levantamento_de_exceção : **raise**
| **raise** *designador_ou_ident_qualificado*
;

O identificador qualificado que, opcionalmente, segue o símbolo **raise** deve ser um identificador de exceção. A seção 10 desse manual faz considerações sobre tratadores de exceção e efeitos do comando **raise**. O resultado e tipo deste comando são indefinidos.

9.3.8 Comando Vazio

O comando vazio consiste de nenhum símbolo e denota ação alguma. É incluído para proporcionar um relaxamento nas regras de pontuação em seqüências de expressões. Este comando não apresenta tipo nem resultado.

comando_vazio : /* vazio */
;

Capítulo 10

Exceções

MC provê facilidades para lidar com erros ou outras situações excepcionais que ocorrem durante a execução de um programa. Por exemplo, ao ocorrer um certo erro em tempo de execução,[1] o programa poderia não parar mas realizar alguma ação, como informar o erro e recuperar-se dele.

Uma *exceção* é um evento que causa a suspensão da execução normal de um programa. Chamar a atenção para um evento excepcional é *levantar* ou *sinalizar* uma exceção. A execução de algumas ações em resposta a uma exceção que foi levantada é chamada *tratamento de exceção*.

Em MC, as exceções são denotadas por identificadores de exceção. Estes podem ser definidos em um programa ou podem ser predefinidos.

As unidades cujas execuções podem ser terminadas prematuramente por uma exceção são os blocos de subprogramas, processos e módulos. Exceções podem ser levantadas pelo sistema, no caso das predefinidas, ou levantadas explicitamente, através do comando *raise*, no caso das definidas em programas ou das predefinidas.

O tratamento de uma exceção é especificado em uma seção de tratadores,[2] declarada em um bloco. Uma exceção pode ser tratada por um tratador declarado em um bloco; pode ser propagada, somente no caso de subprogramas, se o identificador da exceção estiver presente no cabeçalho do subprograma a fim de que o tratamento seja provido pela unidade invocadora ou, ainda, pode interromper a execução da unidade.

[1] Em inglês: *run-time error.*

[2] Em inglês: *handlers.*

10.1 Declaração de Exceção

Uma declaração de exceção introduz zero ou mais identificadores de exceção, que podem aparecer em comandos de levantamento de exceção e tratadores, dentro do escopo da declaração.

declaração_de_exceções : /* vazio */
| *lista_de_identificadores* ';'
;

As exceções predefinidas da linguagem, cujos identificadores são penetrantes, estão listadas no apêndice C.

10.2 Tratadores de Exceção

O tratamento de uma ou mais exceções é especificado por um tratador de exceção. Um tratador pode aparecer no final de um bloco (no corpo de um módulo, subprograma ou processo).

tratamento_de_exceções : /* vazio */
| **exceptions** *seqüência_de_tratadores_de_exceções*
tratador_para_outras_exceções
;
seqüência_de_tratadores_de_exceções : *tratador_de_exceções*
| *seqüência_de_tratadores_de_exceções* ';' *tratador_de_exceções*
;
tratador_de_exceções : **when** *lista_de_identificadores_de_exceção*
do *seqüência_de_expressões* **end**
;
tratador_para_outras_exceções : /* vazio */
| **others** *seqüência_de_expressões* **end**
;

Um tratador diz respeito às exceções cujos identificadores seguem o símbolo **when** e é invocado quando a exceção correspondente é levantada no bloco onde ocorre sua declaração.

Quando uma exceção é levantada em um bloco, durante a elaboração de suas declarações locais ou durante a execução de sua seqüência de expressões, a execução do tratador correspondente substitui a do restante do bloco; as ações seguintes ao ponto onde a exceção foi levantada são puladas e é feito o desvio

Para suprimir a detecção de exceções em um subprograma, processo ou programa, devem-se utilizar atributos de supressão na sua declaração. A especificação destes é um atributo de compilação.

Capítulo 11

Subprogramas

Um subprograma é uma unidade de programa executável que é invocada por uma chamada de subprograma. Sua definição pode ser dada em duas partes: uma declaração, que define sua convenção de chamada, e o seu corpo, que define sua execução. Nesse corpo pode haver a declaração de objetos de escopo local.

Há duas classes de subprogramas: *procedimentos* e *funções*. Uma função devolve um valor e pode apresentar efeitos laterais. Toda expressão de comandos *return* que ocorrem no seu corpo deve ser de tipo compatível para devolução de resultado de função com o tipo da função (cf. 6.5.6). Um procedimento retorna um valor indefinido; seu tipo também é indefinido.

Todos os subprogramas podem ser invocados recursivamente. MC não permite a declaração de subprogramas encaixados sintaticamente. Podem ser declarados de maneira a aceitarem um número variável de parâmetros.

11.1 Subprogramas em Módulos de Definição

Como apresentado na seção 4.1, um módulo de definição contém as declarações dos objetos que podem ser utilizados pelos módulos clientes. A declaração de subprogramas em um módulo de definição consiste apenas de seu cabeçalho, com a especificação do nome do subprograma e seus atributos (por exemplo, se é uma rotina de interrupção, em qual endereço deve estar localizado, etc.), especificação dos tipos de seus parâmetros e respectivos mecanismos de passagem, o tipo do resultado (em se tratando de um subprograma do tipo função) e as exceções por ele propagadas. A forma de especificação dos atributos é dependente de implementação e, para se utilizar essa facilidade de baixo nível, deve ser importado o identificador *SpecialAttributes* do módulo LOW_LEVEL.

cabeçalho_de_subprograma : *cabeçalho_de_procedimento*
| *cabeçalho_de_função*
;
cabeçalho_de_procedimento :
procedure *identificador_com_atributos* '(' *lista_de_tipos_formais* ')'
declaração_de_exceções_propagadas ';'
;
cabeçalho_de_função :
function *identificador_com_atributos* '(' *lista_de_tipos_formais* ')'
':' *designador_ou_ident_qualificado*
declaração_de_exceções_propagadas ';'
;
lista_de_tipos_formais : /* vazio */
| *tipos_formais_fixos*
| *tipos_formais_variáveis*
| *tipos_formais_fixos* ';' *tipos_formais_variáveis*
;

Uma função retorna um valor do tipo especificado pelo identificador qualificado presente no seu cabeçalho.

11.2 Declaração de Subprogramas

A declaração de um subprograma é feita através de um cabeçalho onde estão presentes o nome do subprograma e seus atributos, sua lista de *parâmetros formais*[1] e respectivos mecanismos de passagem, tipo de resultado devolvido (em se tratando de função), lista de exceções propagadas e, finalmente, seu bloco, que contém as declarações locais e expressões que manipulam os parâmetros formais e variáveis conhecidas no ponto de declaração do subprograma.

declaração_de_subprograma : *declaração_de_procedimento*
| *declaração_de_função*
;
declaração_de_procedimento :
procedure *identificador_com_atributos* '(' *lista_de_parâmetros_formais* ')'
declaração_de_exceções_propagadas ';' *bloco identificador* ';'
;
declaração_de_função :

[1] Em inglês: *formal parameters*.

function *identificador_com_atributos* '(' *lista_de_parâmetros_formais* ')'
':' *designador_ou_ident_qualificado*
declaração_de_exceções_propagadas ';' *bloco identificador* ';'
;

lista_de_parâmetros_formais : /* vazio */
| *parâmetros_formais_fixos*
| *parâmetros_formais_variáveis*
| *parâmetros_formais_fixos* ';' *parâmetros_formais_variáveis*
;

O identificador que segue o símbolo **procedure** ou **function** dá o nome do subprograma e deve ser repetido após a declaração de seu bloco. Uma função retorna um valor cujo tipo é dado por um identificador qualificado.

Se um subprograma deve ser conhecido externamente a um módulo então o cabeçalho presente no seu módulo de definição deve ser idêntico ao descrito no módulo de implementação associado, a menos dos identificadores dos parâmetros formais.

Subprogramas não podem trazer, no seu bloco, a declaração ou o cabeçalho de processos ou de outros subprogramas.

Um subprograma pode conter tratadores de exceções para tratar de condições excepcionais durante a sua execução (cf. 10). As exceções listadas no seu cabeçalho são aquelas que o subprograma pode propagar.

declaração_de_exceções_propagadas : /* vazio */
| **exception** '(' *lista_de_identificadores_de_exceção* ')'
;

lista_de_identificadores_de_exceção : /* vazio */
| *lista_de_identificadores*
;

Um procedimento pode apresentar, na sua parte de expressões, um ou mais comandos *return*, que causam o término de uma chamada do procedimento (cf. 9.3.4). Uma função devolve um valor e deve apresentar, na sua parte de expressões, ao menos um comando da forma **return** *expressão*, com *expressão* não vazia; seu tipo resultante deve ser compatível para devolução de resultado de funções com o tipo especificado no cabeçalho da função (cf. 6.5.6).

MC supõe a presença de um comando *return* implícito antecedendo o símbolo **end** que finaliza o bloco de um procedimento.

Subprogramas podem chamar qualquer outro subprograma cujo cabeçalho é conhecido no ponto de chamada. O cabeçalho de um subprograma pode ser especificado antes da declaração do mesmo, da mesma maneira que em módulos

de definição, a fim de se informar as suas características. Também nesse caso, o cabeçalho deve ser repetido de forma idêntica quando for declarado o corpo do subprograma, a menos dos identificadores de parâmetros formais. Uma declaração dessa maneira pode ser utilizada para permitir recursão indireta.

11.3 Parâmetros Formais

Parâmetros formais são identificadores que denotam os *parâmetros efetivos*[2] especificados na chamada do procedimento (cf. 9.2.1,11.4). A correspondência entre parâmetros formais e efetivos é estabelecida quando o subprograma é chamado. Parâmetros formais são considerados locais ao subprograma.

11.3.1 Mecanismo de Passagem

mecanismo_de_passagem : /* vazio */
| **var**
;

Há duas classes de parâmetros em MC: *parâmetros por valor* e *parâmetros por variável*. Estes últimos devem vir antecedidos pelo símbolo **var**. Essas classes são aplicáveis apenas à parte fixa dos parâmetros formais; a parte variável é discutida em 11.3.3.

O parâmetro por valor é, com efeito, uma variável local cujo valor inicial é dado pelo parâmetro efetivo correspondente. Este é avaliado na chamada do subprograma e seu valor é atribuído ao parâmetro por valor na ativação do seu bloco. Atribuições a um parâmetro por valor não têm efeito no parâmetro efetivo, mesmo que este denote uma variável.

O parâmetro por variável é um nome local para o parâmetro efetivo correspondente, que deve ser uma variável ou um componente de uma variável, não podendo simplesmente representar um valor, como uma constante ou uma chamada de função. Assim, nenhuma variável é alocada para o parâmetro formal e o identificador associado denota a variável que é passada como parâmetro efetivo. Qualquer atribuição ao parâmetro por variável é equivalente a uma atribuição ao parâmetro efetivo correspondente. A associação entre um parâmetro por variável e o seu parâmetro efetivo é estabelecida antes da chamada do subprograma. Em conseqüência, se uma variável indexada é o parâmetro efetivo, alterar o índice não afeta o componente que foi realmente passado como parâmetro.

[2]Em inglês: *actual parameters*.

O parâmetro por valor é implementado através do esquema de passagem por valor; o parâmetro por variável é implementado através do esquema de passagem por referência.

11.3.2 Parâmetros Formais Fixos

tipos_formais_fixos : *mecanismo_de_passagem definição_de_tipo_formal*
| *tipos_formais_fixos* ';' *mecanismo_de_passagem definição_de_tipo_formal*
;

definição_de_tipo_formal : *tipo*
| *esquema_de_vetor_com_limites_abertos*
;

esquema_de_vetor_com_limites_abertos :
array '[' *lista_de_tipos_de_índices_de_vetor* ']' **of** *definição_de_tipo_formal*
;

lista_de_tipos_de_índices_de_vetor : *especificação_de_tipo_de_índice*
| *lista_de_tipos_de_índices_de_vetor* ',' *especificação_de_tipo_de_índice*
;

especificação_de_tipo_de_índice : ':' *tipo_simples*
;

parâmetros_formais_fixos :
mecanismo_de_passagem declaração_de_parâmetro_formal
| *parâmetros_formais_fixos* ';' *mecanismo_de_passagem*
declaração_de_parâmetro_formal
;

declaração_de_parâmetro_formal : *lista_de_identificadores* ':' *tipo*
;

Um parâmetro formal fixo é denotado por identificador; tem um tipo formal e um mecanismo de passagem associados.

Uma especificação de parâmetros formais da forma:

$$p_1, \ldots, p_n: T$$

é equivalente a

$$p_1: T; \ldots; p_n: T$$

Se a especificação anterior vier precedida pelo símbolo **var**, a equivalência se verifica com a repetição do mecanismo antes de cada parâmetro formal p_i.

Um tipo formal pode ser a especificação de um vetor com limites abertos, da mesma forma que o esquema de vetores semelhantes do padrão ISO para Pascal, Nível 1 (ISO 7185–1983), discutido também por [Cooper 83].[3] Esse esquema permite a passagem de vetores de tamanho variável como parâmetros, cujos tamanhos dependem dos parâmetros efetivos na chamada do subprograma. A seção 6.2.1 apresenta a maneira como os limites do vetor são estabelecidos.

11.3.3 Parâmetros Formais Variáveis

A parte de parâmetros variáveis de um subprograma consiste, basicamente, na declaração de um vetor com limites abertos, considerado da mesma maneira que no caso de parâmetro por valor em termos de acesso (cf. 6.2.1) e na especificação de funções de conversão que devem ser aplicadas automaticamente aos parâmetros efetivos na chamada do subprograma ou ao seu término, na dependência do vetor ser de entrada ou de saída. O vetor aberto de parâmetros pode ser de entrada (**inparray**) ou de saída (**outparray**). Note-se que é possível especificar apenas uma das classes de parâmetros variáveis.

tipos_formais_variáveis :
  definição_de_vetor_de_parâmetros_variáveis
  '(' *lista_de_identificadores_qualificados* ')' **of** *designador_ou_ident_qualificado*
 | *definição_de_vetor_de_parâmetros_variáveis* **of** *designador_ou_ident_qualificado*
 ;
parâmetros_formais_variáveis :
  identificador ':' *declaração_de_vetor_de_parâmetros_variáveis*
  '(' *lista_de_identificadores_qualificados* ')' **of** *designador_ou_ident_qualificado*
 | *identificador* ':' *declaração_de_vetor_de_parâmetros_variáveis* **of**
  designador_ou_ident_qualificado
 ;
definição_de_vetor_de_parâmetros_variáveis :
      inparray '[' *lista_de_tipos_de_índices_de_vetor* ']'
    | **outparray** '[' *lista_de_tipos_deíndices_de_vetor* ']'
    ;
declaração_de_vetor_de_parâmetros_variáveis :
      inparray '[' *declaração_de_limites_de_vetor* ']'
    | **outparray** '[' *declaração_de_limites_de_vetor* ']'
    ;

[3] Em se tratando do esquema de vetor com limites abertos, a lista de parâmetros formais só poderá ter um identificador para cada vetor desse tipo.

lista_de_identificadores_qualificados : *designador_ou_ident_qualificado*
| *lista_de_identificadores_qualificados* ',' *designador_ou_ident_qualificado*
;

Na declaração do vetor aberto de parâmetros, seus identificadores de limites se comportam como constantes e recebem valores na chamada do subprograma, sendo permitido somente um par de limites (ou seja, um vetor de apenas uma dimensão). O valor do limite inferior será o menor valor dentre os possíveis de seu tipo; o limite superior será o valor do inferior mais o número de expressões (menos um) que serão consideradas como parâmetros efetivos.[4] Podem ser especificadas, entre os símbolos '(' e ')', funções de conversão a serem aplicadas aos parâmetros efetivos na entrada do subprograma ou na saída do mesmo, conforme a classe de parâmetro formal variável. O identificador qualificado que segue o símbolo **of** deve ser um sinônimo de um tipo qualquer, a menos de vetor com limites abertos. Seja esse tipo T_{pv} na discussão a seguir.

Na chamada do subprograma é criado um vetor com limites abertos com tipo de componente T_{pv}; seu tamanho é dependente do número de expressões especificadas como parâmetros efetivos, descontadas as que foram consideradas como parâmetros efetivos fixos.

As funções de conversão listadas no cabeçalho do subprograma devem ter um único argumento. No caso de parâmetros de entrada, as funções não podem ter argumentos de tipos equivalentes entre si (cf. 6.5.2) e seus tipos de resultados devem ser equivalentes ao tipo T_{pv}. No caso de parâmetros de saída, as funções devem ter seus argumentos equivalentes ao tipo T_{pv} mas seus resultados não podem ser de tipos equivalentes entre si.

Se o parâmetro variável é de entrada, a cada parâmetro efetivo da parte variável cujo tipo resultante, T_e, não é equivalente ao tipo T_{pv}, é aplicada a função especificada no seu cabeçalho cujo argumento é de tipo equivalente ao tipo T_e; caso contrário, não há aplicação de função. Os resultados obtidos dessa forma são usados, então, para dar os valores iniciais ao vetor aberto da parte variável de parâmetros.

Se o parâmetro variável é de saída, cada parâmetro efetivo da parte variável deve ser uma variável ou o componente de uma variável, de tipo T_v. Na chamada do subprograma, o vetor aberto não tem os conteúdos de seus elementos definidos. À saída, a cada elemento do vetor, de tipo T_{pv}, é aplicada a função de conversão especificada no seu cabeçalho cujo resultado é de tipo equivalente ao

[4] O termo "mais" deve ser aqui entendido como relação de ordem entre os valores de um tipo ordinal. Assim, caso declarado um procedimento S (p: **inparray** [1..u: BOOLEAN] **of** INTEGER), a chamada S (4,12) resultará em p.1 = FALSE e p.u = TRUE no corpo de S.

tipo T_v, se T_v e T_{pv} não forem equivalentes; caso contrário, não há aplicação de função. Os resultados obtidos dessa forma são atribuídos aos parâmetros efetivos correspondentes.

Exemplo:

```
type String = array [1..MAXSTR] of CHAR;

function ShortCard_to_Str (SHORTCARD): String;
function ShortInt_to_Str (SHORTINT): String;
function Cardinal_to_Str (CARDINAL): String;
function Integer_to_Str (INTEGER): String;
function Str_to_ShortCard (String): SHORTCARD;
function Str_to_ShortInt (String): SHORTINT;
function Str_to_Cardinal (String): CARDINAL;
function Str_to_Integer (String): INTEGER;

                    ⋮

procedure Leitura_de_Inteiros (vl: outparray [l..n: SHORTCARD]
   (Str_to_ShortCard,Str_to_ShortInt,Str_to_Cardinal,Str_to_Integer) of String);
   var j: SHORTCARD;
begin
     for j := vl.l to vl.n do
         SkipSeparators;
         ReadStr (vl[j])
     end
end Leitura_de_Inteiros;

procedure Escrita_de_Inteiros (ve: inparray [l..n: SHORTCARD]
   (ShortCard_to_Str,ShortInt_to_Str,Cardinal_to_Str,Integer_to_Str) of String);
   var j: SHORTCARD;
begin
     for j := ve.l to ve.n do
         WriteStr (ve[j]);
         WriteStr (' ')
     end
end Escrita_de_Inteiros;
         ⋮
```

```
var
    i,j: INTEGER;
    l,m: SHORTCARD;
    n: SHORTINT;
```

Chamadas corretas dos procedimentos *Leitura_de_Inteiros* e *Escrita_de_Inteiros* seriam:[5]

```
Leitura_de_Inteiros (i,j,l);
        /* funções aplicadas: Str_to_Integer,
            Str_to_Integer, Str_to_ShortCard;
            limites: 0,2 */
Leitura_de_Inteiros (m,n);
        /* funções aplicadas:
            Str_to_ShortCard, Str_to_ShortInt;
            limites: 0,1 */
Escrita_de_Inteiros (i+2,i,7);
        /* funções aplicadas: Integer_to_Str,
            Integer_to_Str, ShortCard_to_Str;
            limites: 0,2 */
Escrita_de_Inteiros (m,m*2,n*j);
        /* funções aplicadas: ShortInt_to_Str,
            ShortInt_to_Str, Integer_to_Str;
            limites: 0,2 */
```

O mecanismo de passagem utilizado para implementar esse esquema é a passagem por cópia (passagem por *valor*, no caso de parâmetros de entrada e passagem por *resultado*, no caso de saída. Cf. [GhezJaz87]).

11.4 Chamadas de Subprogramas

Uma chamada de subprograma invoca a execução do corpo desse subprograma. A chamada especifica a associação dos parâmetros efetivos com os parâmetros formais do subprograma. A associação é feita de maneira *posicional*, isto é, o primeiro parâmetro efetivo é associado com o primeiro parâmetro formal e

[5]Nesse exemplo, supõe-se que *ReadStr(v)* devolve, em *v*, a próxima cadeia de caracteres, separada de outra por algum conjunto predefinido de caracteres delimitadores, de um dispositivo de entrada enquanto que *WriteStr(v)* escreve a cadeia de caracteres dada por *v* em um dispositivo de saída.

assim por diante. Um parâmetro efetivo deve ser compatível para passagem de parâmetros com o parâmetro formal correspondente, como discutido em 6.5.5, a menos que se trate da parte variável de parâmetros, onde as funções de conversão explicitadas no subprograma deverão ser aplicadas, conforme o caso.

A avaliação dos parâmetros efetivos é feita da esquerda para a direita; efeitos laterais são propagados nesta ordem. Em se tratando de parâmetros por valor, o parâmetro efetivo pode ser uma expressão qualquer; no caso de parâmetros por variável, o parâmetro efetivo pode ser a especificação de qualquer variável ou componente de variável.

Capítulo 12

Processos

MC oferece facilidades de multiprogramação através da utilização de processos. Um processo é uma entidade que apresenta parâmetros, declarações locais e um corpo que pode ser executado em paralelo com outros processos. Da mesma maneira que subprogramas, se um processo deve ser conhecido por outros módulos, apenas o seu cabeçalho deve ser estar presente no módulo de definição, onde são especificados o nome do processo e seus atributos (por exemplo, tamanho da memória alocada pelo processo ou sua prioridade), dependentes de implementação, e os tipos de seus parâmetros formais e respectivos mecanismos de passagem.

12.1 Declaração

Um processo pode ativar qualquer outro processo cujo cabeçalho é conhecido no ponto de ativação. Neste caso, da mesma forma que subprogramas, apenas o cabeçalho de um processo deve ser especificado antes de sua declaração ou, então, ser importado de um módulo de definição. O cabeçalho deve ser repetido de maneira idêntica quando da declaração do processo, a menos dos identificadores dos parâmetros formais.

cabeçalho_de_processo :
    process *identificador_com_atributos* '(' *lista_de_tipos_formais* ')' ';'
    ;

Uma declaração de processo especifica seu nome e atributos, os parâmetros formais e respectivos mecanismos de passagem e o bloco que traz suas declarações

locais e as expressões que manipulam seus parâmetros e/ou variáveis conhecidas no ponto de declaração do processo.

Processos admitem somente parâmetros por valor na parte fixa de parâmetros e vetor de parâmetros de entrada na parte de parâmetros variáveis. O bloco de um processo não pode conter a declaração de outros processos.

O identificador que segue o símbolo **process** dá o nome ao processo e deve ser repetido após o seu bloco, antecedendo o símbolo ';'.

A utilização de atributos é um recurso de baixo nível que é permitida através da importação do nome de tipo especial *SpecialAttributes* do módulo LOW_LEVEL. A forma de especificação dos atributos é dependente de implementação.

declaração_de_processo : **process** *identificador_com_atributos*
'(' *lista_de_parâmetros_formais* ')' ';' *bloco identificador* ';'
;

12.2 Ativação

A ativação de um processo é feita pelo comando *start* (cf. 9.3.6), com a indicação de seus parâmetros efetivos. As regras de passagem de parâmetros são as mesmas discutidas para os subprogramas (cf. 11,6.5). Como resultado dessa ativação, é devolvido um valor, de tipo predefinido PROCESSID, que dá a identificação do processo. Através dessa identificação é que se pode implementar um mecanismo de comunicação entre processos. A ativação também pode especificar uma lista de atributos de processo, dependente da implementação (veja seção supra-citada).

A ativação de um processo implica na alocação de recursos necessários (por exemplo, área de memória), conforme as possibilidades da implementação a ser utilizada. O término de um processo (quando o final de seu bloco é atingido ou quando é executado algum comando *return*) libera esses recursos, de maneira que também depende de implementação, e torna inválida a identificação do processo que lhe foi associada quando de sua ativação. A referência a um processo que não está mais ativo levanta a exceção NO_PROCESS_ID.

A alocação de memória para um processo é feita através do subprograma ALLOCATE_PROCESS_MEMORY e a desalocação pelo subprograma DEALLOCATE_PROCESS_MEMORY, definidos no módulo LOW_LEVEL. Se forem redefinidos no programa, o compilador utilizará as novas versões ao invés das que estão definidas no referido módulo. O número de parâmetros de cada um desses subprogramas é dependente de implementação.

Se um processo é interrompido por alguma exceção que não pode ser tratada no seu bloco, este termina sem indicação do erro.

Note-se que não há forma diretamente definida na linguagem de determinar se um processo está ativo ou não. Esta e outras facilidades, como, por exemplo, um esquema de sinalização entre processos, podem ser criadas através dos recursos providos pelo módulo LOW_LEVEL.

Apêndice A

Operadores de MC

Um operador n-ário define os tipos $T_1, \ldots, T_n$ de seus operandos. Aplicados a eles, produz um valor de tipo T_r como resultado. Alguns operadores podem ser sobrecarregados,[1] isto é, um mesmo símbolo que representa uma operação n-ária aplicável a operandos de uma certa seqüência $T_1, \ldots, T_n$ de tipos, pode representar uma outra operação aplicável a operandos de uma outra seqüência $T'_1, \ldots, T'_m$ de tipos.

Este apêndice descreve os operadores de MC e sua seqüência de tipos associada, bem como o tipo do resultado obtido a partir da aplicação de cada um deles à sua seqüência de operandos.

Os tipos ordinais predefinidos SHORTINT, SHORTCARD, INTEGER, CARDINAL, mais o tipo UNIV_INTEGER constituem os *tipos inteiros*. Um tipo subintervalo com tipo raiz de tipo inteiro também é de tipo inteiro. Os tipos inteiros mais o tipo predefinido REAL constituem os *tipos numéricos*.

Na descrição a seguir, sejam T_1,T_2, os tipos dos operandos do operador binário especificado e T_1 o tipo do operando de um operador unário.

A.1 Operadores Aritméticos

A.1.1 Operadores Aritméticos Unários

Operador	T_1	Tipo de Resultado
+	numérico	mesmo do operando
−	numérico	mesmo do operando

[1] Em inglês: *overloaded operators*.

A.1.2 Operadores Aritméticos Binários: $+, -, *, /, \%$

T_1†	T_2†				
	SHORTINT	SHORTCARD	INTEGER	CARDINAL	REAL
SHORTINT	SHORTINT	—	INTEGER	—	—
SHORTCARD	—	SHORTCARD	—	CARDINAL	—
INTEGER	INTEGER	—	INTEGER	—	—
CARDINAL	—	CARDINAL	—	CARDINAL	—
REAL	—	—	—	—	REAL‡

† Para um operando de tipo subintervalo de inteiros, deve ser considerado o seu *tipo raiz*.
‡ Não aplicável ao operador %.

A.2 Operadores Lógicos

Um operador lógico resulta num valor de tipo BOOLEAN.

Operador	T_1	T_2
not†	BOOLEAN	—
and	BOOLEAN	BOOLEAN
or	BOOLEAN	BOOLEAN
xor	BOOLEAN	BOOLEAN

† Operador unário.

A.3 Operadores Relacionais

Um operador relacional resulta num valor de tipo BOOLEAN.

Operador	T_1	T_2
=,#,<, <=, >, >=	BOOLEAN	BOOLEAN
	CHAR	CHAR
	inteiro	inteiro
	REAL	REAL
	enumerado T_e ou subintervalo de tipo raiz T_e	enumerado T_e ou subintervalo de tipo raiz T_e
=,#	**pointer to T**	**pointer to T**
	PROCESSID	PROCESSID
=,#,<=, >=	conjunto com tipo base T_b	conjunto com tipo base T_b
in,notin	T_b	conjunto com tipo base T_b

A.4 Operadores para Conjuntos: $+, -, *, /$

T_1	T_2	Tipo do Resultado
conjunto com tipo base T_b	conjunto com tipo base T_b	conjunto com tipo base T_b

Operador	Significado‡
+	x **in** $(s1 + s2) \Leftrightarrow (x$ **in** $s1)$ **or** $(x$ **in** $s2)$
−	x **in** $(s1 - s2) \Leftrightarrow (x$ **in** $s1)$ **and** $(x$ **notin** $s2)$
*	x **in** $(s1 * s2) \Leftrightarrow (x$ **in** $s1)$ **and** $(x$ **in** $s2)$
/	x **in** $(s1/s2) \Leftrightarrow (x$ **in** $s1)$ **#** $(x$ **in** $s2)$

‡ $s1, s2$: conjuntos com tipo base T_b; x: T_b.

A.5 Operadores para Caracteres e Cadeias: &

T_1	T_2	Tipo do Resultado†
CHAR	CHAR	**array** [1..2] **of** CHAR
CHAR	**array** [1..L] **of** CHAR	**array** [1..L+1] **of** CHAR
array [1..L] **of** CHAR	CHAR	**array** [1..L+1] **of** CHAR
array [1..L_1] **of** CHAR	**array** [1..L_2] **of** CHAR	**array** [1..L_1+L_2] **of** CHAR

† L, L_1, L_2 são os tamanhos dos vetores.

Apêndice B

Funções Predefinidas de MC

MC apresenta funções cujos identificadores são penetrantes, definidas no módulo SYSTEM. Essas funções podem ser consideradas em três grupos: funções de transferência de tipos, funções de conversão e funções aritméticas.

B.1 Funções de Transferência de Tipos

Uma função de transferência de tipos transfere o valor de seu parâmetro, de um certo tipo, para o tipo especificado pelo identificador da função. Uma função desse tipo é uma facilidade de baixo nível e, para ser utilizada, deve ser importado o nome de tipo especial *TypeTransferFunctions*, do módulo LOW_LEVEL.

Essas funções não envolvem cálculo algum nem geração de código, tendo o propósito de desabilitar o sistema de verificação de tipos da linguagem, indicando qual a interpretação que deve ser dada a uma expressão.

Essa facilidade é estritamente dependente de implementação; por isso, uma particular implementação poderá não aceitar transferência entre tipos de tamanhos diferentes.

Os identificadores das funções de transferência de tipos podem ser quaisquer nomes de tipos, predefinidos ou definidos em um programa.

B.2 Funções de Conversão

Uma função de conversão devolve a representação do seu parâmetro, de um certo tipo, na representação correspondente de tipo especificado pela função. Pode levantar exceções, dependendo da conversão a ser realizada e do valor do parâmetro.

A descrição das funções de conversão se encontra na tabela B.1. Os argumentos são parâmetros por valor.

B.3 Funções Aritméticas

As funções aritméticas realizam operações aritméticas simples, de uso geral. Podem levantar exceções, dependendo da operação a ser realizada e dos valores dos seus argumentos.

A descrição das funções aritméticas se encontra na tabela B.2. Os argumentos podem ser parâmetros por valor ou por variável, conforme a descrição apresentada.

Nome da Função	Tipos dos Argumentos	Tipo do Resultado	Função
CHR(x)	inteiro	CHAR	Caractere com o número ordinal x
TOSHORTINT(x)	inteiro	SHORTINT	x representado como um valor de tipo SHORTINT
TOSHORTCARD(x)	inteiro	SHORTCARD	x representado como um valor de tipo SHORTCARD
TOINTEGER(x)	inteiro	INTEGER	x representado como um valor de tipo INTEGER
TOCARDINAL(x)	inteiro	CARDINAL	x representado como um valor de tipo CARDINAL
TOREAL(x)	inteiro	REAL	x representado como um valor de tipo REAL
FLOOR(x)	REAL	INTEGER	$\lfloor x \rfloor$†
CEIL(x)	REAL	INTEGER	$\lceil x \rceil$ ‡
TRUNC(x)	REAL	INTEGER	$\lfloor x \rfloor$ se $x \geq 0$, $\lceil x \rceil$ se $x < 0$
ORD(x)	inteiro, enumerado, CHAR, BOOLEAN	CARDINAL	Número ordinal de x no conjunto de valores definido pelo tipo de x
VAL(T,x)	T: BOOLEAN, tipo enumerado; X: CARDINAL	T	Valor com o número ordinal x, de tipo T

† $\lfloor x \rfloor = i \Rightarrow x - 1 < i \leq x, i \in Z, x \in \Re$

‡ $\lceil x \rceil = i \Rightarrow x \leq i < x + 1, i \in Z, x \in \Re$

Tabela B.1: Funções de Conversão entre Tipos

Nome da Função	Tipos dos Argumentos	Tipo do Resultado	Função
ABS(x)	inteiro	tipo de x	$\mid x \mid$, valor absoluto
ASH(x,n)	x,n: inteiros	tipo de x	$x \times 2^n$, deslocamento aritmético
DEC(v,n)	v,n: inteiros	tipo de v	$v := v - n$; devolve novo valor de v
EVEN(x)	inteiro	BOOLEAN	$x \% 2 = 0$
EXCL(v,x)	v: **set of** T; x: T	tipo de v	$v := v - \{x\}$; devolve novo valor de v
INC(v,n)	v,n: inteiros	tipo de v	$v := v + n$; devolve novo valor de v
INCL(v,x)	v: **set of** T; x: T	tipo de v	$v := v + \{x\}$; devolve novo valor de v
ODD(x)	inteiro	BOOLEAN	$x \% 2 \# 0$
PRED(x)	enumerado, CHAR, BOOLEAN, inteiro	tipo de x	predecessor de x no conjunto de valores definido pelo tipo de x
SUCC(x)	enumerado, CHAR, BOOLEAN, inteiro	tipo de x	sucessor de x no conjunto de valores definido pelo tipo de x

Tabela B.2: Funções Aritméticas Predefinidas

Apêndice C

Exceções Predefinidas em MC

As exceções predefinidas na linguagem MC, cujos identificadores são penetrantes, são listadas a seguir, com as condições de erro responsáveis pelo levantamento da exceção. A verificação das condições de erro pode deixar de ser realizada através de atributos especificados na cabeçalho de módulos de implementação e de programa, processos ou subprogramas.

ACCESS_ERROR
: Valor NIL usado como o endereço de um objeto.

DISCRIMINANT_ERROR
: Acesso a um componente de uma parte variante de um registro não prescrito pelo valor corrente do campo de marca correspondente.

DIVIDE_ERROR
: Divisão de um número por zero.

INDEX_ERROR
: O valor de um índice ultrapassa os limites estabelecidos na declaração do vetor.

NO_PROCESS_ID
: O processo especificado por um identificador de processo não está ativo.

NO_VALUE_ERROR
: Acesso a uma variável que ainda não teve um valor atribuído.

OVERFLOW
: Operação aritmética tenta resultar num valor muito grande para ser manipulado pela implementação.

PROCESS_ACTIVATION_ERROR
: Falha na ativação de um processo.

RANGE_ERROR
: O intervalo definido por um certo tipo é excedido quando da avaliação de uma expressão ou na atribuição de um valor a uma variável desse tipo.

SELECT_ERROR
: Nenhuma das alternativas do comando de desvio condicional *case* é escolhida e o comando não apresenta uma cláusula *others*.

STACK_OVERFLOW
: Espaço reservado para área de pilha é exaurido.

STORAGE_OVERFLOW
: Espaço reservado para memória dinâmica é exaurido.

UNDERFLOW
: Operação aritmética envolvendo o tipo REAL tenta resultar num valor que é muito pequeno para ser manipulado pela implementação.

Apêndice D

Módulos Especiais de MC

Os módulos listados a seguir fazem parte da definição da linguagem MC e são apresentados aqui na forma de módulos de definição. O conteúdo dos módulos apresentados é minimal. Note-se que nem sempre será possível fazer a descrição completa usando somente os elementos da linguagem.

D.1 Módulo SYSTEM

O módulo SYSTEM exporta as entidades que são dependentes da configuração do sistema a ser utilizado. A descrição a seguir supõe uma implementação em microcomputadores comuns de 16 bits.

Todos os identificadores exportados pelo módulo SYSTEM são *penetrantes*, a menos de indicação em contrário.

```
definition module SYSTEM;

const
        /* Limites para o tipo SHORTINT */
        MINSHORTINT = -128;
        MAXSHORTINT = 127;
        /* Limite superior para o tipo SHORTCARD */
        MAXSHORTCARD = 255;
        /* Limites para o tipo INTEGER */
        MININT = -32768;
        MAXINT = 32767;
```

```
        /* Limite superior para o tipo CARDINAL */
        MAXCARD = 65535;

        /* Função CHR */
function CHR (SHORTCARD): CHAR;
        /* Devolve o caractere cuja representação é o inteiro, de tipo
        SHORTCARD, passado como argumento. */

type
        /* Tipos simples predefinidos */
        SHORTINT = [MINSHORTINT..MAXSHORTINT];
        SHORTCARD = [0..MAXSHORTCARD];
        INTEGER = [MININT..MAXINT];
        CARDINAL = [0..MAXCARD];
        REAL = subintervalo_dos_reais;
        BOOLEAN = (FALSE,TRUE);
        CHAR = [CHR(0)..CHR(255)];

        /* Tipo identificador de processo */
type
        PROCESSID;

        /* Tipos e constantes predefinidos mas que não podem ser usados
        por um programa em MC */
/*
const
        /* Limites para o tipo UNIV_INTEGER */
        MINUNIVINT = MININT;
        MAXUNIVINT = MAXCARD;

type
        UNIV_INTEGER = [MINUNIVINT..MAXUNIVINT]; /* inteiro universal */
        UNIV_POINTER; /* apontador universal */
        UNIV_SET; /* conjunto universal */
*/

        /* Exceções predefinidas */
exception
        ACCESS_ERROR,
```

```
    ACTIVATION_PROCESS_ERROR,
    DISCRIMINANT_ERROR,
    DIVIDE_ERROR,
    INDEX_ERROR,
    NO_PROCESS_ID,
    NO_VALUE_ERROR,
    OVERFLOW,
    RANGE_ERROR,
    SELECT_ERROR,
    STACK_OVERFLOW,
    STORAGE_OVERFLOW,
    UNDERFLOW;

    /* Funções que devolvem os tamanhos (múltiplos de unidade
    básica de armazenamento; cf. módulo LOW_LEVEL) de tipos e
    variáveis */

function SIZE (T): CARDINAL;
    /* Devolve o tamanho de memória reservada para a variável pas-
    sada como argumento. Tipo T: qualquer tipo. */

function TYPESIZE (T): CARDINAL;
    /* Devolve o tamanho de memória reservada para o tipo passado
    como argumento. Tipo T: qualquer tipo. */

    /* Procedimento para alocação de memória dinâmica */
procedure NEW (var T; inparray [n] of INTEGER)
        exception (STORAGE_OVERFLOW)
    /* Tipo T: pointer to T'. Devolve um apontador de tipo T para
    a variável dinâmica de tipo T'. Utiliza a função de alocação de
    memória ALLOCATE_MEMORY, que é definida no módulo LOW_LEVEL,
    podendo ser redefinida no programa, sendo esta a versão conside-
    rada pelo compilador. A versão desse procedimento com a parte
    de parâmetros variáveis é usada para alocação de vetores com li-
    mites abertos. Os parâmetros efetivos darão os limites do vetor
    na sua criação. Se o nome do vetor for v, v[i] e v[i+1] darão os
    limites inferior e superior, nessa ordem. i = 1,3,5,...,n - 1. */

    /* Procedimento para liberar espaço na memória dinâmica */
procedure DISPOSE (T);
```

```
        /* Tipo T: pointer to T'. Libera a área de memória apon-
        tada pelo argumento de tipo T. Utiliza o procedimento para de-
        salocação de memória DEALLOCATE_MEMORY, que é definido no
        módulo LOW_LEVEL, podendo ser redefinida no programa, sendo
        esta a versão considerada pelo compilador. */


        /* Funções Predefinidas de Conversão */

function TOSHORTINT (T): SHORTINT
            exception (RANGE_ERROR);
        /* Tipo T: inteiro. Devolve a representação do argumento como
        SHORTINT. */

function TOSHORTCARD (T): SHORTCARD
            exception (RANGE_ERROR);
        /* Tipo T: inteiro. Devolve a representação do argumento como
        SHORTCARD. */

function TOINTEGER (T): INTEGER
            exception (RANGE_ERROR);
        /* Tipo T: inteiro. Devolve a representação do argumento como
        INTEGER. */

function TOCARDINAL (T): CARDINAL
            exception (RANGE_ERROR);
        /* Tipo T: inteiro. Devolve a representação do argumento como
        CARDINAL. */

function TOREAL (T): REAL
            exception (RANGE_ERROR);
        /* Tipo T: inteiro. Devolve a representação do argumento como
        REAL. */

function FLOOR (REAL): INTEGER
            exception (RANGE_ERROR);
        /* Devolve o maior inteiro que é menor ou igual ao argumento. */

function CEIL (REAL): INTEGER
            exception (RANGE_ERROR);
```

```
    /* Devolve o menor inteiro que é maior ou igual ao argumento. */

function TRUNC (REAL): INTEGER
        exception (RANGE_ERROR);
    /* Devolve a parte inteira do argumento. */

function ORD (T): CARDINAL exception (RANGE_ERROR);
    /* Devolve o número ordinal correspondente ao argumento no con-
    junto de valores definido pelo seu tipo. Tipo T: inteiro, enumerado,
    CHAR e BOOLEAN. */

function VAL (T; CARDINAL): T exception (RANGE_ERROR);
    /* Devolve o valor com o número ordinal correspondente ao se-
    gundo argumento, de tipo dado pelo primeiro. Tipo T: enumerado
    ou BOOLEAN. */


    /* Funções Predefinidas */

function ABS (T): T;
    /* Devolve o valor absoluto do argumento. Tipo T: inteiro ou
    REAL. */

function ASH (T; T'): T
        exception (OVERFLOW,UNDERFLOW);
    /* Tipos T,T': inteiros. Devolve o primeiro argumento deslocado
    aritmeticamente o número de bits dado pelo segundo argumento.
    Deslocamento à direita: segundo argumento negativo; deslocа-
    mento à esquerda: segundo argumento positivo. */

function DEC (var T; T'): T
        exception (RANGE_ERROR);
    /* Tipos T,T': inteiros. Devolve o primeiro argumento diminuído
    do segundo, atualizando-o. */

function EVEN (T): BOOLEAN;
    /* Tipo T: inteiro. Devolve a indicação lógica do seu argumento
    ser par. */

function EXCL (var T; T'): T;
```

```
        /* Tipo T: set of T'; tipo T': enumerado, subintervalo ou um
        dos tipos ordinais predefinidos ou sinônimos. Retira o elemento
        dado pelo segundo argumento do conjunto definido pelo primeiro,
        atualizando-o e devolvendo-o como resultado. */

function INC (var T; T): T
            exception (RANGE_ERROR);
        /* Tipos T,T': inteiros. Devolve o primeiro argumento somado ao
        segundo, atualizando-o. */

function INCL (var T; T'): T;
        /* Tipo T: set of T'; tipo T': enumerado, subintervalo ou um
        dos tipos ordinais predefinidos ou sinônimos. Inclui o elemento
        dado pelo segundo argumento no conjunto definido pelo primeiro,
        atualizando-o e devolvendo-o como resultado. */

function ODD (T): BOOLEAN;
        /* Tipo T: inteiro. Devolve a indicação lógica do seu argumento
        ser ímpar. */

function PRED (T): T exception (RANGE_ERROR);
        /* Devolve o predecessor do argumento no conjunto de valores
        definido pelo seu tipo. Tipo T: enumerado, inteiro, CHAR ou BOO-
        LEAN. */

function SUCC (T): T exception (RANGE_ERROR);
        /* Devolve o sucessor do argumento no conjunto de valores defi-
        nido pelo seu tipo. Tipo T: enumerado, inteiro, CHAR ou BOOLEAN.
        */


end SYSTEM.
```

D.2 Módulo LOW_LEVEL

O módulo LOW_LEVEL exporta as entidades que possibilitam o acesso às facilidades de baixo nível providas por uma dada implementação.

O exemplo a seguir supõe uma implementação em um microprocessador de 16 bits, compatível com Intel 8086.

```
definition module LOW_LEVEL;

    /* Tipos especiais de autorização para utilização de recursos de
    baixo nível */

type
    TypeTransferFunctions; /* funções de transferência de tipos */
    SpecialAttributes; /* atributos de identificadores de variáveis,
                          subprogramas e processos */


    /* Tipos para manipulação de endereços */

type
    BYTE; /* unidade endereçável */
    WORD; /* 2 bytes; também unidade endereçável */
    ADDRESS; /* endereço */


    /* Funções para manipulação de endereços */

function ADDR (T): ADDRESS;
    /* Tipo T: qualquer tipo. Devolve o endereço da primeira posição
    de memória reservada para o seu argumento. */

function SEGMENT (ADDRESS): CARDINAL;
    /* Devolve o valor do registrador de segmento do endereço. */

function OFFSET (ADDRESS): CARDINAL;
    /* Devolve o valor do deslocamento do endereço em relação ao
    registrador de base do segmento. */

function PTR (CARDINAL; CARDINAL): ADDRESS;
    /* Devolve o endereço montado a partir do primeiro argumento
    (valor do registrador de segmento) e do segundo (deslocamento).
    */

procedure INCRPTR (var T);
```

```
        /* Tipo T: pointer to T'. Incrementa o apontador passado como
        argumento, isto é, soma o tamanho do tipo T' ao argumento,
        atualizando-o. */

procedure DECRPTR (var T);
        /* Tipo T: pointer to T'. Decrementa o apontador passado como
        argumento, isto é, diminui o tamanho do tipo T' do argumento,
        atualizando-o. */


        /* Função para alocação de memória dinâmica */
function ALLOCATE_MEMORY (CARDINAL): ADDRESS
            exception (STORAGE_OVERFLOW);
        /* Devolve o endereço da primeira posição da área de memória
        alocada, cujo tamanho é passado como argumento. */

        /* Procedimento para desalocação de memória dinâmica */
procedure DEALLOCATE_MEMORY (ADDRESS; CARDINAL;);
        /* Libera a área de memória a partir do endereço dado pelo pri-
        meiro argumento e tamanho dado pelo segundo. */


        /* Funções para manipulação dos bits de tipos inteiros */

function LSH (T; T'): T;
        /* Tipos T,T': inteiros. Devolve o primeiro argumento deslocado
        logicamente à esquerda o número de bits dado pelo segundo. */

function RSH (T; T'): T;
        /* Tipos T,T': inteiros. Devolve o primeiro argumento deslocado
        logicamente à direita o número de bits dado pelo segundo. */


        /* Atributos Especiais */

type
        /* registradores de máquina */
        REGISTERS = (AX,BX,CX,DX,SP,BP,SI,DI,CS,DS,SS,ES,FLAGS,ANYREG);

        /* Esquema de prioridades */
```

```
const
    MIN_PRIORITY = 0; /* mínima prioridade */
    MAX_PRIORITY = 15; /* máxima prioridade */
    BASE_PRIORITY = 4; /* prioridade base */

type
    PRIORITY = [MIN_PRIORITY..MAX_PRIORITY];

    /* Memória alocada por um processo, em palavras */

const
    MIN_WS = 1024; /* mínimo */
    MAX_WS = 8192; /* máximo */
    DEF_WS = 2048; /* tamanho inicial de um processo */
    EXT_WS = 65535; /* máxima extensão possível */

type
    WORKING_SET = [MIN_WS..EXT_WS];

    /* Subprogramas para gerenciamento de memória para processos */

function ALLOCATE_PROCESS_MEMORY (var ADDRESS, CARDINAL): ADDRESS
        exception (ACTIVATION_PROCESS_ERROR);
    /* Reserva uma área de memória a partir do primeiro argumento,
    de tamanho dado pelo segundo. O endereço devolvido é o da
    primeira posição reservada; o primeiro argumento é incrementado
    com o tamanho alocado. */

procedure DEALLOCATE_PROCESS_MEMORY (var ADDRESS; CARDINAL);
    /* Libera a área de memória que precede o endereço dado pelo
    primeiro argumento e tamanho dado pelo segundo. O primeiro
    argumento é decrementado do segundo. */


    /* Alinhamento de um objeto */

type
    ALIGNED = (UNALIGNED,BYTE_ALIGNED, WORD_ALIGNED);
```

```
        /* UNALIGNED: alinhamento decidido pelo compilador;
        BYTE_ALIGNED: alinhamento por byte;
        WORD_ALIGNED: alinhamento por palavra.*/

        /* Atributos de alocação de variáveis, subprogramas, processos e
        módulos */

type
        ALLOCATION = (STATIC,AUTOMATIC);
        /* STATIC: variável alocada apenas uma vez; AUTOMATIC: variável
        alocada cada vez que se entra no subprograma ou processo no qual
        foi declarada. */

procedure AT (CARDINAL; CARDINAL);
        /* Associa o objeto em cuja lista de atributos este ocorre ao en-
        dereço de memória passado como argumento (valores de segmento
        e deslocamento, respectivamente). */


        /* Atributos para verificação de erros. CHECK: habilita as veri-
        ficações; SUPRESS_CHECK: desabilita-as. São aplicáveis a módulos
        de implementação, módulos de programa, subprogramas e pro-
        cessos e são estendidas a toda entidade sintaticamente encaixada
        naquela onde os atributos são especificados. */

type
        CHECK = (ALL_CHECK, ACTIVATION_PROCESS_CHECK,
                BOUNDS_CHECK, DISCRIMINANT_CHECK,
                NO_VALUE_CHECK,
                OVERFLOW_CHECK, POINTER_CHECK,
                RANGE_CHECK, UNDERFLOW_CHECK);
        SUPRESS_CHECK = (SUPRCHK_ALL, SUPRCHK_ACTIVATION_PROCESS,
                        SUPRCHK_BOUNDS, SUPRCHK_DISCRIMINANT,
                        SUPRCHK_NO_VALUE, SUPRCHK_OVERFLOW,
                        SUPRCHK_POINTER, SUPRCHK_RANGE,
                        SUPRCHK_UNDERFLOW);
```

/* ALL_CHECK, SUPRCHK_ALL: todas as formas de verificação; ACTIVATION_PROCESS_CHECK, SUPRCHK_ACTIVATION_PROCESS: falha na ativação de um processo; BOUNDS_CHECK, SUPRCHK_BOUNDS: verificação de estouro dos limites de um vetor; DISCRIMINANT_CHECK, SUPRCHK_DISCRIMINANT: verificação de acesso inválido a um campo da parte variante de um registro com o valor corrente do campo de marca não correspondente ao valor prescrito para o campo; NO_VALUE_CHECK, SUPRCHK_NO_VALUE: verificação de acesso a variável sem valor inicial; OVERFLOW_CHECK, SUPRCHK_OVERFLOW: verificação de estouro na representação de inteiros e reais (valores muito grandes para serem manipulados pela implementação); POINTER_CHECK, SUPRCHK_POINTER: verificação de acesso a uma variável dinâmica apontada por uma expressão cujo resultado é NIL; RANGE_CHECK, SUPRCHK_RANGE: verificação de estouro no intervalo de valores definido por um tipo numa atribuição; UNDERFLOW_CHECK, SUPRCHK_UNDERFLOW: verificação de estouro na representação de reais (valores muito pequenos para serem manipulados pela implementação). */

/* Tipos de subprogramas. */

type

SUBPROGRAM_TYPE = (INITIALIZE,INTERRUPT);
/* INITIALIZE: o subprograma deve ser chamado antes da ativação do módulo de programa; INTERRUPT: subprograma que trata de uma interrupção. */

/* Atributos que regulam o acesso a variáveis. Aplicáveis a variáveis e parâmetros formais. */

type

VAR_ACCESS = (READONLY,WRITEONLY);
/* READONLY: a variável pode apenas ser lida; WRITEONLY: a variável pode apenas ser escrita. */

/* Tamanhos de objetos */

type

SIZE_VAR = (BITS(n), BYTES(n), WORDS(n), LONGWORDS(n), QUADWORDS(n));

```
        /* BITS(n): bits; WORDS(n): bytes; WORDS(n): palavras (2 bytes);
        LONGWORDS(n): palavras longas (4 bytes); QUADWORDS(n): pala-
        vras quádruplas (8 bytes). n indica o número de unidades de
        armazenamento e é uma extensão da notação de tipos proposta
        por MC. */


        /* Comunicação entre Processos */

type
        /* esquema de sinalização entre processos: troca de mensagens */
        Message;

procedure Send (PROCESSID; Message) exception (NO_PROCESS_ID);
    /* Envia a mensagem passada no segundo argumento para o pro-
    cesso cuja identificação é dada pelo primeiro argumento. */

procedure Receive (var PROCESSID; var Message);
    /* Recebe e devolve, no segundo argumento, uma mensagem pen-
    dente, enviada pelo processo cuja identificação é devolvida no pri-
    meiro argumento. */

function Number_of_Messages (PROCESSID): CARDINAL
            exception (NO_PROCESS_ID);
    /* Devolve o número de mensagens pendentes para o processo cuja
    identificação é passada como argumento. */


end LOW_LEVEL.
```

D.3 Módulo MATH_LIB

O módulo MATH_LIB exporta as funções aritméticas.

```
definition module MATH_LIB;

        /* Exceções indicativas de erro */
```

```
exception
        PARAMETER_ERROR;


        /* Funções aritméticas */

function Sqrt (REAL): REAL exception (PARAMETER_ERROR);
        /* Devolve a raiz quadrada do argumento. */

function Exp (REAL): REAL exception (PARAMETER_ERROR);
        /* Devolve a constante neperiana (e) elevada ao argumento. */

function Ln (REAL): REAL exception (PARAMETER_ERROR);
        /* Devolve o logaritmo neperiano do argumento. */


        /* Funções Trigonométricas */

function Sin (REAL): REAL exception (PARAMETER_ERROR);
        /* Devolve o seno do argumento (arco em radianos). */

function Cos (REAL): REAL exception (PARAMETER_ERROR);
        /* Devolve o cosseno do argumento (arco em radianos). */

function ArcTg (REAL): REAL exception (PARAMETER_ERROR);
        /* Devolve o arco, em radianos, cuja tangente é dada pelo argu-
        mento. */

end MATH_LIB.
```

Apêndice E

Linguagem MC

E.1 Gramática

/* Parte léxica */

identificador : *letra*
| *identificador letra*
| *identificador dígito*
| *identificador* '_'
;

dígito : '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
;

letra : 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' | 'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' | 'O'
| 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' | 'V' | 'W' | 'X' | 'Y' | 'Z'
| 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' | 'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' | 'o'
| 'p' | 'q' | 'r' | 's' | 't' | 'u' | 'v' | 'w' | 'x' | 'y' | 'z'
;

número : *inteiro* | *real*
;

inteiro : *inteiro_decimal* | *inteiro_hexadecimal*
;

inteiro_decimal : *dígito* | *inteiro_decimal dígito*
;

inteiro_hexadecimal : *dígito dígitos_hexadecimais base_hexadecimal*
;

dígitos_hexadecimais : /* vazio */
| *dígitos_hexadecimais dígito_hexadecimal*

```
                    ;
dígito_hexadecimal : dígito | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
                   | 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
                   ;
base_hexadecimal : 'H' | 'h'
                 ;
real : parte_inteira '.' parte_fracionária fator_de_escala
     ;
parte_inteira : inteiro_decimal
              ;
parte_fracionária : /* vazio */
                  | parte_fracionária dígito
                  ;
fator_de_escala : /* vazio */
                | símbolo_de_escala sinal_do_expoente inteiro_decimal
                ;
símbolo_de_escala : 'E' | 'e'
                  ;
sinal_do_expoente : '+' | '-' | /* vazio */
                  ;
cadeia : "seqüência_de_caracteres"
       ;
seqüência_de_caracteres : /* vazio */
                        | seqüência_de_caracteres caractere
                        ;


/* Unidades de Compilação */

unidade_de_compilação : módulo_de_definição
                      | módulo_de_programa
                      | implementation módulo_de_implementação
                      ;
módulo_de_definição : definition module identificador ';'
                                  parte_de_importação parte_de_definições
                      end identificador '.'
                    ;
módulo_de_programa : módulo_de_implementação
                   ;
```

```
módulo_de_implementação : module identificador_com_atributos ';'
                                    parte_de_importação bloco
                            identificador '.'
                          ;

/* Lista de Importação/Exportação */

parte_de_importação : /* vazio */
                    | parte_de_importação from identificador import
                                                lista_de_identificadores ';'
                    | parte_de_importação import lista_de_identificadores ';'
                    ;
parte_de_exportação : /* vazio */
                    | export lista_de_identificadores ';'
                    ;

/* Parte de Definições */

parte_de_definições : /* vazio */
                    | parte_de_definições const declaração_de_constantes
                    | parte_de_definições type declaração_de_tipos
                    | parte_de_definições var declaração_de_variáveis
                    | parte_de_definições exception declaração_de_exceções
                    | parte_de_definições cabeçalho_de_processo
                    | parte_de_definições cabeçalho_de_subprograma
                    ;

/* Parte de Declarações */


/* Constantes */

declaração_de_constantes : /* vazio */
          | declaração_de_constantes identificador '=' expressão_constante ';'
          ;
expressão_constante : expressão
```

```
                    ;

/* Tipos */

declaração_de_tipos : /* vazio */
                    | declaração_de_tipos declaração_de_tipo_opaco
                    | declaração_de_tipos declaração_de_tipo_sinônimo
                    ;
declaração_de_tipo_opaco : identificador ';'
                         ;
declaração_de_tipo_sinônimo : identificador '=' tipo ';'
                            ;
lista_de_tipos_sinônimos : /* vazio */
                         | lista_de_tipos_sinônimos declaração_de_tipo_sinônimo
                         ;
tipo : nome_de_tipo
     | intervalo
     | enumeração
     | tipo_estruturado
     | tipo_apontador
     ;
nome_de_tipo : identificador_de_tipo
             ;
identificador_de_tipo : identificador
             | identificador_de_tipo '.' identificador
             ;
intervalo : identificador_de_tipo especificação_de_subintervalo
          | especificação_de_subintervalo
          ;
especificação_de_subintervalo : '[' expressão_constante '..' expressão_constante ']'
                              ;
enumeração : '(' lista_de_identificadores ')'
           ;
tipo_estruturado : tipo_vetor
                 | tipo_registro
                 | tipo_conjunto
                 ;
```

```
/* Vetores */

tipo_vetor : vetor_com_limites_definidos
           | vetor_com_limites_abertos
           ;
vetor_com_limites_definidos : array lista_de_tipos_simples of tipo
                            ;
lista_de_tipos_simples : lista_de_tipos_simples ',' tipo_simples
                       | tipo_simples
                       ;
tipo_simples : tipo
             ;
vetor_com_limites_abertos : array '[' declaração_de_limites_de_vetor ']' of tipo
                          ;
declaração_de_limites_de_vetor : identificador '..' identificador ':' tipo_simples
    | declaração_de_limites_de_vetor ',' identificador '..' identificador ':' tipo_simples
    ;


/* Registros */

tipo_registro : record end
              | record seqüência_de_campos end
              ;
seqüência_de_campos : lista_de_campos fim_de_lista_de_campos
          | seqüência_de_campos ';' lista_de_campos fim_de_lista_de_campos
          ;
lista_de_campos : lista_de_identificadores ':' tipo_simples
                | case identificador ':' tipo_simples of
                      lista_de_variantes outros_casos_para_registro end
                | case tipo_simples of
                      lista_de_variantes outros_casos_para_registro end
                ;
fim_de_lista_de_campos : /* vazio */
                       | ';'
                       ;
lista_de_variantes : lista_de_variantes '|' variante
                   | variante '|'
                   | variante
```

```
                ;
variante : lista_de_rótulos_de_casos ':' seqüência_de_campos
         ;
lista_de_rótulos_de_casos : lista_de_rótulos_de_casos ',' rótulo_de_casos
                          | rótulo_de_casos
                          ;
rótulo_de_casos : expressão_constante
                | expressão_constante '..' expressão_constante
                ;
outros_casos_para_registro : /* vazio */
                           | others seqüência_de_campos
                           ;


/* Conjuntos */

tipo_conjunto : set of tipo_simples
              ;


/* Apontadores */

tipo_apontador : pointer to tipo
               ;


/* Variáveis */

declaração_de_variáveis : /* vazio */
    | declaração_de_variáveis lista_de_identificadores_com_atributos ':' tipo ';'
    | declaração_de_variáveis lista_de_identificadores_com_atributos ':' tipo
                        ':=' expressão_para_valor_inicial ';'
    ;
lista_de_identificadores_com_atributos : identificador_com_atributos
    | lista_de_identificadores_com_atributos ';' identificador_com_atributos
    ;
expressão_para_valor_inicial : expressão
                             | expressão_estruturada
                             ;
expressão_estruturada : '(' componente_de_expressão_estruturada ')'
```

```
                    ;
componente_de_expressão_estruturada : expressão
                | componente_de_expressão_estruturada ',' expressão
                | componente_de_expressão_estruturada ',' expressão_estruturada
                ;

/* Blocos e Módulos Locais */

bloco : parte_de_declarações parte_de_expressões
      ;
parte_de_declarações : /* vazio */
                     | parte_de_declarações const declaração_de_constantes
                     | parte_de_declarações type lista_de_tipos_sinônimos
                     | parte_de_declarações var declaração_de_variáveis
                     | parte_de_declarações exception declaração_de_exceções
                     | parte_de_declarações declaração_de_processo
                     | parte_de_declarações declaração_de_subprograma
                     | parte_de_declarações declaração_de_módulo
                     | parte_de_declarações cabeçalho_de_processo
                     | parte_de_declarações cabeçalho_de_subprograma
                     ;
declaração_de_módulo : module identificador_com_atributos ';'
                                parte_de_importação parte_de_exportação
                                bloco identificador
                     ;

/* Processos */

cabeçalho_de_processo :
      process identificador_com_atributos '(' lista_de_tipos_formais ')' ';'
      ;
declaração_de_processo : process identificador_com_atributos
      '(' lista_de_parâmetros_formais ')' ';' bloco identificador ';'
      ;

/* Subprogramas */
```

```
cabeçalho_de_subprograma : cabeçalho_de_procedimento
                         | cabeçalho_de_função
                         ;
cabecalho_de_procedimento :
        procedure identificador_com_atributos '(' lista_de_tipos_formais ')'
                  declaração_de_exceções_propagadas ';'
        ;
cabecalho_de_função :
        function identificador_com_atributos '(' lista_de_tipos_formais ')'
                 ':' designador_ou_ident_qualificado
                 declaração_de_exceções_propagadas ';'
        ;
declaração_de_subprograma : declaração_de_procedimento
                          | declaração_de_função
                          ;
declaração_de_procedimento :
    procedure identificador_com_atributos '(' lista_de_parâmetros_formais ')'
              declaração_de_exceções_propagadas ';' bloco identificador ';'
    ;
declaração_de_função :
    function identificador_com_atributos '(' lista_de_parâmetros_formais ')'
             ':' designador_ou_ident_qualificado
             declaração_de_exceções_propagadas ';' bloco identificador ';'
    ;

/* Exceções Propagadas */

declaração_de_exceções_propagadas : /* vazio */
                  | exception '(' lista_de_identificadores_de_exceção ')'
                  ;

/* Parâmetros e Tipos dos Parâmetros */

lista_de_tipos_formais : /* vazio */
                       | tipos_formais_fixos
                       | tipos_formais_variáveis
                       | tipos_formais_fixos ';' tipos_formais_variáveis
                       ;
```

```
tipos_formais_fixos : mecanismo_de_passagem definição_de_tipo_formal
        | tipos_formais_fixos ';' mecanismo_de_passagem definição_de_tipo_formal
        ;
lista_de_parâmetros_formais : /* vazio */
            | parâmetros_formais_fixos
            | parâmetros_formais_variáveis
            | parâmetros_formais_fixos ';' parâmetros_formais_variáveis
            ;
parâmetros_formais_fixos :
        mecanismo_de_passagem declaração_de_parâmetro_formal
      | parâmetros_formais_fixos ';' mecanismo_de_passagem
                                     declaração_de_parâmetro_formal
      ;
declaração_de_parâmetro_formal : lista_de_identificadores ':' tipo
                                ;
mecanismo_de_passagem : /* vazio */
                        | var
                        ;
tipos_formais_variáveis :
    definição_de_vetor_de_parâmetros_variáveis
   '(' lista_de_identificadores_qualificados ')' of designador_ou_ident_qualificado
  | definição_de_vetor_de_parâmetros_variáveis of designador_ou_ident_qualificado
  ;
parâmetros_formais_variáveis :
    identificador ':' declaração_de_vetor_de_parâmetros_variáveis
   '(' lista_de_identificadores_qualificados ')' of designador_ou_ident_qualificado
  | identificador ':' declaração_de_vetor_de_parâmetros_variáveis of
    designador_ou_ident_qualificado
  ;
definição_de_tipo_formal : tipo
                         | esquema_de_vetor_com_limites_abertos
                         ;
esquema_de_vetor_com_limites_abertos :
    array '[' lista_de_tipos_de_índices_de_vetor ']' of definição_de_tipo_formal
    ;
lista_de_tipos_de_índices_de_vetor : especificação_de_tipo_de_índice
    | lista_de_tipos_de_índices_de_vetor ',' especificação_de_tipo_de_índice
    ;
especificação_de_tipo_de_índice : ':' tipo_simples
```

```
                                    ;
definição_de_vetor_de_parâmetros_variáveis :
        inparray '[' lista_de_tipos_de_índices_de_vetor ']'
      | outparray '[' lista_de_tipos_deíndices_de_vetor ']'
      ;
declaração_de_vetor_de_parâmetros_variáveis :
        inparray '[' declaração_de_limites_de_vetor ']'
      | outparray '[' declaração_de_limites_de_vetor ']'
      ;

/* Parte de Expressões */

parte_de_expressões : begin seqüência_de_expressões
                            tratamento_de_exceções end
                    ;
seqüência_de_expressões : seqüência_de_expressões ';' expressão_com_rótulo
                        | expressão_com_rótulo
                        ;
expressão_com_rótulo : rótulo expressão
                     | expressão
                     ;
rótulo : '<<' identificador '>>'
       ;

/* Expressões e Comandos */

expressão : expressão_própria
          | comando
          ;
comando : comando_de_atribuição
        | comando_condicional
        | comando_repetitivo
        | comando_de_desvio
        | comando_de_escopo_de_registro
        | comando_de_ativação_de_processo
        | comando_de_levantamento_de_exceção
        | comando_vazio
        ;
```

```
comando_de_atribuição : designador_ou_ident_qualificado ':=' expressão
            | designador_ou_ident_qualificado ':=' expressão_estruturada
            ;
designador_ou_ident_qualificado : identificador
              | designador_ou_ident_qualificado '.' identificador
              | designador_ou_ident_qualificado '[' lista_de_expressões ']'
              | designador_ou_ident_qualificado '↑'
              ;
lista_de_expressões : lista_de_expressões ',' expressão
                    | expressão
                    ;
comando_condicional : if expressão
        then seqüência_de_expressões comando_if_encadeado
        else seqüência_de_expressões end
      | if expressão then seqüência_de_expressões comando_if_encadeado end
      | case expressão of lista_de_casos outros_casos_de_seleção end
      ;
comando_if_encadeado : /* vazio */
      | comando_if_encadeado elsif expressão then seqüência_de_expressões
      ;
lista_de_casos : lista_de_casos '|' caso fim_de_casos
               | caso fim_de_casos
               ;
fim_de_casos : /* vazio */
             | '|'
             ;
caso : lista_de_rótulos_de_casos ':' seqüência_de_expressões
     ;
outros_casos_de_seleção : /* vazio */
                        | others seqüência_de_expressões
                        ;
comando_repetitivo : while expressão do seqüência_de_expressões end
             | repeat seqüência_de_expressões until expressão
             | loop seqüência_de_expressões end
             | for identificador ':=' expressão to expressão
                  expressão_de_incremento do seqüência_de_expressões end
             ;
expressão_de_incremento : /* vazio */
                        | by expressão
```

```
                    ;
comando_de_desvio : continue
                  | continue rótulo
                  | return expressão
                  | exit
                  | exit rótulo
                  ;
comando_de_escopo_de_registro :
            with expressão do seqüência_de_expressões end
            ;
comando_de_ativação_de_processo :
             start especificação_de_atributo designador_ou_ident_qualificado
                  '(' lista_de_parâmetros_efetivos ')'
             ;
comando_de_levantamento_de_exceção : raise
                                   | raise designador_ou_ident_qualificado
                                   ;
comando_vazio : /* vazio */
              ;
expressão_própria : expressão_disjuntiva
                  ;
expressão_disjuntiva : expressão_conjuntiva
    | expressão_disjuntiva operador_lógico_disjuntivo expressão_conjuntiva
    ;
expressão_conjuntiva : expressão_conjuntiva and relação
                     | relação
                     ;
relação : expressão_simples
        | expressão_simples operador_relacional expressão_simples
        ;
expressão_simples : operador_aditivo termo outros_termos
                  | termo outros_termos
                  ;
outros_termos : /* vazio */
              | outros_termos operador_aditivo termo
              ;
termo : fatores
      ;
fatores : fatores operador_multiplicativo fator
```

```
        | fator
        ;
fator : not primário
      | primário
      ;
primário : número
         | cte_ASCII
         | cadeia
         | conjunto
         | chamada_de_função_ou_procedimento
         | expressão_de_incremento_ou_decremento
         | '(' expressão ')'
         ;
cte_ASCII : ''' caractere '''
          ;
expressão_de_incremento_ou_decremento : designador_ou_ident_qualificado
              | '++' designador_ou_ident_qualificado
              | '--' designador_ou_ident_qualificado
              | designador_ou_ident_qualificado '++'
              | designador_ou_ident_qualificado '--'
              ;
chamada_de_função_ou_procedimento :
              | designador_ou_ident_qualificado '(' lista_de_parâmetros_efetivos ')'
              ;
lista_de_parâmetros_efetivos : lista_de_parâmetros_efetivos ',' expressão
                             | expressão
                             ;
conjunto : designador_ou_ident_qualificado '{' lista_de_elementos '}'
         | '{' lista_de_elementos '}'
         ;
lista_de_elementos : lista_de_elementos ',' elemento
                   | elemento
                   ;
elemento : expressão '..' expressão
         | expressão
         ;
operador_lógico_disjuntivo : or | xor
                           ;
operador_relacional : '=' | '#' | '<' | '<=' | '>' | '>=' | in | notin
```

```
                    ;
operador_aditivo : '+' | '-' | '&'
                 ;
operador_multiplicativo : '*' | '/' | '%'
                        ;

/* Exceções */

declaração_de_exceções : /* vazio */
                       | lista_de_identificadores ';'
                       ;
tratamento_de_exceções : /* vazio */
                       | exceptions seqüência_de_tratadores_de_exceções
                                    tratador_para_outras_exceções
                       ;
seqüência_de_tratadores_de_exceções : tratador_de_exceções
        | seqüência_de_tratadores_de_exceções ';' tratador_de_exceções
        ;
tratador_de_exceções : when lista_de_identificadores_de_exceção
                            do seqüência_de_expressões end
                     ;
tratador_para_outras_exceções : /* vazio */
                              | others seqüência_de_expressões end
                              ;

/* Listas de Identificadores */
identificador_com_atributos : identificador especificação_de_atributos
                            ;
especificação_de_atributos : /* vazio */
                           | '[' lista_de_expressões ']'
                           ;
lista_de_identificadores_qualificados : designador_ou_ident_qualificado
      | lista_de_identificadores_qualificados ',' designador_ou_ident_qualificado
      ;
lista_de_identificadores_de_exceção : /* vazio */
                                    | lista_de_identificadores
                                    ;
```

lista_de_identificadores : *lista_de_identificadores* ',' *identificador*
| *identificador*
;

E.2 Palavras Reservadas

and	**end**	**in**	**procedure**	**until**
array	**exception**	**inparray**	**process**	**var**
begin	**exceptions**	**loop**	**raise**	**with**
by	**exit**	**module**	**record**	**when**
case	**export**	**not**	**repeat**	**while**
const	**for**	**notin**	**return**	**xor**
continue	**from**	**of**	**set**	
definition	**function**	**or**	**start**	
do	**if**	**others**	**then**	
else	**implementation**	**outparray**	**to**	
elsif	**import**	**pointer**	**type**	

E.3 Operadores e Símbolos

"	'	..	:=	<<	>>
++	--	=	#	<	<=
>	>=	\	;	(	)
.	[	]	:	,	|
↑	{	}	+	-	&
*	/	%			

E.4 Identificadores Penetrantes

ABS	ACCESS_ERROR	ASH
BOOLEAN	CARDINAL	CEIL
CHAR	CHR	DEC
DISCRMINANT_ERROR	DISPOSE	EVEN
EXCL	FLOOR	INC
INCL	INDEX_ERROR	INTEGER
MAXCARD	MAXINT	MAXSHORTCARD
MAXSHORTINT	MININT	MINSHORTINT
NEW	NIL	NO_PROCESS_ID
NO_VALUE_ERROR	ODD	ORD
OVERFLOW	PRED	PROCESS_ACTIVATION_ERROR
PROCESSID	RANGE_ERROR	REAL
SELECT_ERROR	SIZE	SHORTCARD
SHORTINT	STACK_OVERFLOW	STORAGE_OVERFLOW
SUCC	TOINTEGER	TOREAL
TOSHORTCARD	TOSHORTINT	TOCARDINAL
TRUNC	TYPESIZE	UNDERFLOW
VAL		

Bibliografia

[Carvalho 89] C. S. R. Carvalho. *Projeto de uma Linguagem de Programação.* Tese de Mestrado. DCC - IMECC - UNICAMP (em preparação).

[Cooper 83] D. Cooper. *Standard Pascal - User Reference Manual.* W. W. Norton & Company, 1983.

[DecVaxPascal 85] Digital Equipment Corporation. *Programming in VAX Pascal.* Order Number: AA-L369B-TE. March 1985.

[DraftAda 83] United States Department of Defense. *Ada Programming Language.* Draft ANSI/MIL-STD 1815A. January 1983.

[GhezJaz87] C. Ghezzi, M. Jazayeri. *Programming Language Concepts. 2/E.* John Wiley & Sons, 1987.

[JenWirth 85] K. Jensen, N. Wirth. *Pascal User Manual and Report.* Third Edition, revised by A. B. Mickel, J. F. Miner. Springer-Verlag, 1985.

[Johnson 75] S. C. Johnson. *Yacc: Yet Another Compiler Compiler.* Computer Science Techical Report, Nº 32. Bell Laboratories, Murray Hill, 1975.

[KernRit 78] B. W. Kernighan, D. M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[PedroJr 83] J. Pedro Jr. *Uma implementação de Módula-2: Análise e Representação Intermediária.* Tese de Mestrado. DCC–IMECC–UNICAMP. Novembro 1983.

[Smedema 84] C. H. Smedema. *Chill User Manual.* Chill Bulletin, vol. 4, nº 1. AT&T and Philips Telecommunications. March 1984.

[Wirth 85] N. Wirth. *Programming in Modula-2. Third, Corrected Edition.* Springer-Verlag, 1985.