A Linguagem de Programação CHILL

Márcio Machado Pereira

Centro de Pesquisa e Desenvolvimento da TELEBRÁS
Caixa Postal 1579. 13085 Campinas SP

Resumo

Este artigo apresenta uma descrição geral da linguagem de programação CHILL. A apresentação destaca os conceitos, as construções e os mecanismos que tornam CHILL uma das linguagens de programação de alto nível mais aprimoradas da atualidade. É apresentado um pequeno histórico que identifica a motivação que determinou a criação da linguagem e descreve o processo de evolução pelo qual ela vem passando. São enfatizados as construções e os mecanismos providos para a construção de programas concorrentes, para a decomposição de programas em partes que possam ser desenvolvidas de forma independente e segura, para o controle de eventos dependentes de tempo etc.

Abstract

This paper presents an over-all description of the CHILL programming language. It emphasizes the concepts, contructions and mechanisms provided by CHILL that make it presently one of the most refined high level programming languages. The paper presents a brief historical review that identifies the motivations that led to a new language definition and describes its ongoing evolution process. Recent ideas on software engineering are reflected in the presentation of subjects such as safe and independent piecewise program development, mechanisms for concurrent process execution, supervision of time-dependent events, exception handling etc.

1 Introdução

A linguagem de programação CHILL[1], definida e recomendada pelo CCITT[2], foi concebida, inicialmente, com o objetivo de servir à programação de centrais telefônicas CPA[3]. Hoje, entretanto, ela é empregada em várias outras aplicações—na área de telecomunicações principalmente. Comutação de mensagens, comutação de pacotes e modelagem em geral são exemplos típicos.

CHILL pertence a uma geração de linguagens de programação que sucedeu a PL/I, Algol 68 e Pascal [9]. A primeira versão de sua definição—Recomendação Z.200 [1]—foi publicada pelo CCITT em 1980. Desde então, a manutenção e evolução dessa recomendação está a cargo de um grupo de trabalho do CCITT. O trabalho desse grupo é organizado em períodos de estudo de quatro anos. No final desses períodos é publicada uma versão atualizada da Recomendação Z.200. Esse *modus operandi* tem possibilitado que, num dado período de estudos, decisões adotadas em períodos anteriores possam ser reavaliadas à luz da realimentação fornecida por

[1] CCITT High Level Language
[2] Comitê Consultivo Internacional para Telegrafia e Telefonia
[3] Controle por Programa Armazenado

usuários e implementadores da linguagem Isso tem sido extremamente positivo para a linguagem, na medida em que o rigor de sua definição, sua legibilidade e poder de expressão vêm sendo aprimorados.

A definição da linguagem CHILL é flexível no sentido de admitir subconjuntos, ao contrário de outras linguagens como, por exemplo, Ada™, onde grande ênfase está concentrada na transportabilidade de programas [2]. CHILL preocupa-se com a sedimentação de uma *cultura*, uma vez que a transportabilidade de programas —especialmente daqueles orientados a equipamentos dedicados—é, em geral, difícil de ser conseguida.

Os requisitos mais significativos, considerados no projeto da linguagem CHILL, são os seguintes:

- aumentar a confiabilidade dos programas através do emprego intensivo de verificações em tempo de compilação;
- permitir a produção de código objeto altamente eficiente;
- ser flexível e poderosa a fim de cobrir um amplo espectro de aplicações e diferentes tipos de *hardware*;
- encorajar o desenvolvimento de programas modulares e estruturados;
- dar suporte a aplicações de tempo real proporcionando mecanismos para comunicação e sincronização de processos e supervisão de tempo;
- ser fácil de aprender e usar.

A linguagem CHILL é, atualmente, uma das linguagens de alto nível mais aprimoradas. Ela emprega muitas das mais recentes idéias disponíveis na área de engenharia de *software* e dispõe de construções e mecanismos orientados à programação concorrente em arquiteturas *mono* ou *multi*-processadoras [4]. Seus mecanismos de compilação em partes são flexíveis, rigorosos e amplamente adequados à atividade de desenvolvimento de *software* em larga escala [5,8]. Os mecanismos de supervisão de tempo da linguagem CHILL representam uma das características que a distingue das outras linguagens e que a tornam extremamente poderosa e adequada para aplicações que precisam controlar processos em tempo real. CHILL é uma linguagem que trata com rigor a relação entre a definição e o uso de objetos em geral[4]. Isso permite a realização de verificações em tempo de compilação. Algumas regras, no entanto, só podem ser verificadas em tempo de execução, sendo necessária a geração de código objeto destinado à verificação dessas regras. Os comandos e construções existentes em CHILL induzem à produção de programas estruturados. Programas podem ser escritos em CHILL de maneira totalmente independente de máquinas alvo. Os mecanismos de controle de visibilidade de objetos permitem ao programador escrever programas amplamente modularizados. Tipos abstratos de dados podem ser facilmente implementados, encapsulando-se, em *módulos*, objetos e funções que operem sobre eles [6,7]. A linguagem permite, sob algumas condições, o tratamento de algoritmos como dados—conceito largamente empregado em programação por objetos. O tratamento de exceções—pré-definidas ou definidas pelo programador—representa um dos inúmeros mecanismos, presentes na linguagem, que lhe conferem grande poder de expressão.

As seções a seguir introduzem os diversos mecanismos da linguagem CHILL, enfatizando as características ou aspectos não comumente encontrados em outras linguagens de programação.

[4] *strongly typed language*

2 Dados

Os objetos de dados da linguagem são variáveis e valores. Toda variável tem um tipo a ela associado (diz-se que a variável é daquele tipo). Todo valor tem uma classe a ele associada (diz-se que o valor é daquela classe). O tipo associado a uma variável define um conjunto de propriedade tais como: o conjunto de valores que essa variável pode conter, os métodos de acesso à mesma, as operações permitidas sobre seus valores etc. A classe associada a um valor determina as operações permitidas sobre os valores e os tipos das variáveis que podem conter esse valor. Tipos e classes são descritos na seção 2.1. Variáveis e valores são descritos na seção 2.2. Expressões e operadores são descritos na seção 2.3.

2.1 Tipos e Classes

A linguagem CHILL provê as seguintes categorias de tipos:

tipos discretos: inteiro, caractere, booleano, enumeração e seus intervalos;

tipo conjunto: conjunto de elementos de um tipo discreto;

tipos referência: referência estática, referência livre e referência dinâmica;

tipo procedimento: permite a manipulação de procedimentos como dados;

tipos compostos: cadeia de caracteres, cadeia de *bits*, matriz e estrutura;

tipo instância: identificação para processos;

tipos de sincronização: evento e *buffer*, usados para comunicação e sincronização de processos;

tipos de entrada e saída: associação, acesso e texto, usados para operações de entrada e saída em arquivos;

tipos de temporização: duração e tempo absoluto, usados para supervisão de tempo.

A linguagem CHILL provê um conjunto de tipos padrão. Novos tipos podem ser introduzidos através de construções destinadas à definição de tipos (seção 2.1.1).

Um tipo pode ser estático (i.e., para o qual todas as propriedades podem ser determinadas estaticamente) ou dinâmico (i.e., para o qual algumas propriedades só são conhecidas em tempo de execução). Tipos dinâmicos são sempre tipos parametrizados, cujos parâmetros denotam valores não constantes.

Tipos discretos

Os tipos discretos definem conjuntos e subconjuntos de valores bem-ordenados. São tipos discretos os tipos inteiro, booleano, caractere, enumeração e seus intervalos. Os nomes *int*, *bool* e *char* são pré-definidos como nomes de tipo inteiro, booleano e caractere, respectivamente. Um tipo enumeracão define um conjunto de valores ordenados denotados por nomes. Por exemplo:

```
newmode
   Frutas = set (abacaxi, laranja, banana, tangerina);
```

Um intervalo de um tipo discreto define um conjunto de valores compreendido entre dois limites especificados. Por exemplo:

```
newmode
   Dia_do_Mes = int (1:31);
```

Tipo Conjunto

Um tipo conjunto define valores que são conjuntos de valores de seu tipo membro. Por exemplo:

```
newmode
   Logico = powerset bool;
```

Os valores definidos pelo tipo de nome *Logico* são:

```
[false], [true], [false, true], [ ]
```

Tipos Referência

Um tipo referência define referências a variáveis referenciáveis. Existem 3 tipos de referências: estáticas, livres e dinâmicas.

Um tipo referência estática define valores de referência a variáveis de um tipo estático especificado. No exemplo abaixo, o valor *10* é armazenado na variável *i*:

```
newmode
   Ref_Int = ref int;
dcl
   i int,
   p Ref_Int := -> i;

   p -> := 10;
```

Um tipo referência livre define valores de referência a variáveis de qualquer tipo estático. O nome *ptr* é pré-definido como nome de tipo referência livre.

Um tipo referência dinâmica define valores de referência a variáveis de tipo dinâmico. No exemplo abaixo, *r* é uma referência a uma fatia de *s*. Esta fatia é especificada pelo limite inferior *i* e limite superior *j*.

```
dcl
   s chars (128), i int:= 10, j int:= 20,
   r row chars (128) := -> s (i:j);
```

Tipo Procedimento

Um tipo procedimento define valores que denotam procedimentos. Um tipo procedimento permite a manipulação dinâmica de procedimentos. Por exemplo, um procedimento pode ser atribuído a uma variável do tipo procedimento, pode ser um parâmetro real em uma chamada de outro procedimento etc. A definição de um tipo procedimento especifica seu nome, o tipo dos parâmetros (se houver), o tipo do resultado (se houver) e as exceções (se houver). Por exemplo:

```
newmode
   Op = proc (int, int) returns (int);
```

Variáveis do tipo *Op* podem conter valores que denotam procedimentos com 2 parâmetros do tipo *int* passados por valor e que devolvem um valor do tipo *int*.

Tipo Cadeia

Um tipo cadeia define sequências de valores booleanos (cadeia de *bits*) ou de caracteres (cadeia de caracteres). Um tipo cadeia pode ter tamanho fixo ou variável. Por exemplo:

```
dcl
   string chars (128) varying := "Exemplo de cadeia de tamanho variavel";
dcl
   byte bools (8) := B'0010.1000';
```

As notações binária (B'), octal (O') e hexadecimal (H') podem ser usadas para valores literais de cadeias de *bits*. O caractere sublinha (_) pode ser usado, sendo introduzido somente para aumentar a legibilidade.

Tipo Matriz

Um tipo matriz define valores compostos que consistem de uma lista de valores definidos pelo seu tipo elemento. Cada elemento desta lista pode ser tratado como um objeto separado com todas as propriedades do tipo elemento. Por exemplo:

```
newmode
   Linha  = set (a,b,c,d,e,f,g,h),
   Coluna = int (1:8),
   Pecas  = set (rei, dama, torre, bispo, cavalo, peao);
dcl
   tabuleiro array (Linha, Coluna) Pecas;

   tabuleiro (d,1) := dama;
```

Tipo Estrutura

Um tipo estrutura define valores compostos que consistem de uma lista de valores, cada um dos quais é selecionável pelo nome de um componente. Estruturas podem ser fixas, variantes

e parametrizadas. Uma estrutura variante pode ter um ou mais campos alternativos. Uma estrutura parametrizada é definida a partir de um tipo estrutura variante, a partir do qual alternativas são escolhidas através da especificação de expressões literais. Por exemplo:

```
newmode
   Pessoa = struct (nome chars (40),
                    idade int (0:120),
                    estado_civil set (solteiro, casado),
                    case estado_civil of
                       (casado):
                          nome_do_conjuge chars (40)
                       else
                    esac);
dcl
   filho   Pessoa,
   esposa Pessoa (casado);
```

A variável *esposa* contém a alternativa pertencente ao valor *casado* do campo seletor *estado_civil*. É incorreto mudar a alternativa variante atribuindo *solteiro* ao campo seletor.

2.1.1 Definições de tipo

Uma definição de tipo define um nome que denota o tipo especificado. Por exemplo:

```
newmode
   Cruzado = int;
```

O nome *Cruzado* denota um tipo similar ao tipo *int*. O tipo *Cruzado* herda propriedades do tipo inteiro, como os valores definidos pelo tipo, suas operações, tamanho etc.

Existem duas maneiras de definir um tipo, chamadas definições de tipo-sinônimo e definições de novo-tipo. Um tipo-sinônimo é *sinonímio* ao seu tipo definidor enquanto que um novo-tipo não o é. Por exemplo:

```
synmode
   Dolar, Cruzado = int;
```

Neste caso, variáveis do tipo *Dolar*, do tipo *Cruzado* e do tipo pré-definido *int* são equivalentes, isto é, podem operar entre si. Seja a definição:

```
newmode
   Dolar, Cruzado = int;
```

Neste caso, embora os tipos *Dolar*, *Cruzado* e *int* sejam similares, eles não são equivalentes, isto é, variáveis do tipo *Dolar* não podem operar, por exemplo, com variáveis do tipo *Cruzado* (não podem ser somadas, atribuídas, passadas como parâmetros etc.). Esta distinção faz com que um novo-tipo seja isolado de outros tipos similares.

As regras de compatibilidade da linguagem determinam como uma variável de um certo tipo ou valor de uma certa classe podem ser usados. Variáveis do tipo *Dolar*, acima, só podem operar com valores da classe *valor-de-Dolar* (valores contidos em variáveis de tipo *Dolar*, constantes que têm o tipo *Dolar* associado etc.) e da classe *derivada-de-int* (literais inteiros, resultados de algumas rotinas pré-definidas etc.).

2.2 Variáveis e Valores

Variáveis são "lugares" onde valores podem ser armazenados ou de onde valores podem ser obtidos. Uma variável é criada através de um comando de declaração de variável. Um comando de declaração pode também definir um novo nome de acesso a uma variável já criada (O nome de acesso é chamado de variável identidade.). No exemplo abaixo, são criadas duas variáveis: *i*, do tipo inteiro, e a, do tipo matriz. O nome *b* é um novo nome de acesso ao elemento da matriz denotado por a*(25+i)*:

```
dcl
   a array (1 : 100) int,
   i int := 10,
   b int loc := a (25 + i);
```

O tipo da variável identidade pode, também, ser dinâmico. No exemplo abaixo, o nome *parte* é um acesso a uma fatia de *vetor* definida pelos limites *i* e *j*:

```
newmode
   Tipo.Matriz = array (1 : 100) int;
dcl
   i int (1 : 100) := 5,
   j int (1 : 100) := 30,
   vetor Tipo.Matriz,
   parte Tipo.Matriz loc dynamic := vetor (i : j);
```

Uma variável é de *leitura apenas* se o tipo associado ao nome que denota a variável tem a propriedade de *leitura apenas*. No exemplo abaixo, o conteúdo da variável criada pode ser modificado dentro do módulo *m* através do nome *y*. Externamente a *m*, somente o nome *x* é visível (veja a seção 4.3) e o conteúdo da variável não pode ser modificado porque o tipo de *x* tem a propriedade de *leitura apenas*:

```
m:
   module
      dcl
         y int,
         x read int loc := y;

      grant x;
         :
   end m;
```

Existem duas formas de iniciação de variáveis: iniciação ligada ao domínio e iniciação ligada ao tempo de vida. Na iniciação ligada ao tempo de vida, o valor atribuído à variável tem que ser um valor constante. Na inciação ligada ao domínio, o valor não precisa ser constante, uma vez que ele é avaliado toda vez que se entra no domínio na qual a declaração está inserida (isto pode ocorrer mais de uma vez durante o tempo de vida da variável). No exemplo abaixo, *j* tem a sua iniciação ligada ao domínio e *i* tem sua iniciação ligada ao tempo de vida:

```
dcl
   i int init := 10,
   j int      := 2 * i + 10;
```

Valores

Valores são objetos sob os quais se define um conjunto específico de operações. Comandos de definição de sinônimos estabelecem novos nomes que denotam valores constantes. A linguagem permite também o uso de constantes de tipos compostos. No exemplo abaixo, *maior_salario* denota um valor da classe *valor-de-Cruzado*, *total_empregados* denota um valor da classe *derivada-de-int* e *c_zero* denota um valor da classe *valor-de-Complexo* :

```
newmode
   Cruzado   = int,
   Complexo = struct (real, imag int);

syn
   maior_salario     Cruzado   = 10.000,
   total_empregados            = 100,
   c_zero            Complexo = [.real:0, .imag:0 ];
```

Tuplas

Uma tupla é uma lista de valores. Uma tupla pode denotar um valor de conjunto, um valor de matriz ou um valor de estrutura. Uma tupla de conjunto consiste de uma lista de expressões e/ou faixas de valores do seu tipo membro. Por exemplo:

```
newmode
   Salada_de_Frutas = powerset Frutas,
   Frutas           = set (abacaxi, laranja, tangerina, pera, figo, pessego);
dcl
   frutas_citricas Salada_de_Frutas := [abacaxi:tangerina];
```

Uma tupla de matriz consiste de uma lista de valores, possivelmente com rótulos, do tipo do elemento da matriz. Em uma tupla da matriz sem rótulos um valor é especificado para cada elemento da matriz. Em uma tupla de matriz com rótulos, os valores correspondem aos elementos cujos índices são especificados no rótulo que precede o valor. O rótulo *else* denota todos os índices não especificados explicitamente. O rótulo * denota todos os valores do tipo índice do tipo matriz. Por exemplo:

```
dcl
   vetor1 array (1:5) char := |'a', 'b', 'c', 'd', 'e'|,
   vetor2 array (1:10) int  := |(1,2): 0, (4:7): 1, (else): 2|,
   vetor3 array (1:15) bool := |(*): false|;
```

Uma tupla de estrutura consiste de uma lista de valores correspondentes aos campos da estrutura. Estes valores podem ser rotulados ou não. Por exemplo:

```
dcl
   s struct (c chars(40) varying, i int, b bool)
            := [ "exemplo", 18, true ];
dcl
   v struct (i struct (x, y int),
             j array (1 : 10) bool )
             := [.i: [ 10, 20 ],
                 j: [ (*): true ]];
```

2.3 Expressões e Operadores

A linguagem CHILL provê o uso de expressões condicionais, isto é, expressões selecionáveis pela avaliação de uma expressão booleana (*if...then...else...fi*) ou por uma lista de seletores (*case...of...else...esac*). Por exemplo:

```
dcl
   i, j, k int;
   ⋮
i := if j < 0 then k - j else k + j fi;
```

A Tabela 1 contém todos os operadores da linguagem, agrupados em categorias de acordo com o tipo dos valores sob os quais eles podem ser aplicados. Na avaliação de expressões, operadores de maior precedência são aplicados primeiro. Operadores de mesma precedência são aplicados na ordem textual, da esquerda para a direita.

Dados dois valores booleanos *b1* e *b2*, o resultado de "*b1 orif b2*" é o mesmo que "*if b1 then true else b2 fi*". Por exemplo:

```
if p = null orif p ->.tipo = valor
   then
   ⋮
fi;
```

Se o resultado da avaliação da expressão "*p = null*" for verdadeiro, as ações da parte *then* são executadas sem que a expressão "*p ->.tipo*" seja avaliada.

Dados dois valores booleanos *b1* e *b2*, o resultado de "*b1 andif b2*" é o mesmo que "*if not b1 then false else b2 fi*".

Dada uma cadeia *c1*, o resultado de "*(n)c1*", que usa o operador de repetição de cadeias, é o mesmo que "*c1 // ... // ... // c1*", *n* vezes. Por exemplo:

Operador	Precedência	Função	Domínio
+	3	soma	
−	3	subtração	
*	4	multiplicação	
/	4	divisão	Valores Inteiros
mod	4	módulo	
rem	4	resto	
−	5	negação	
=	2	igualdade	
/=	2	desigualdade	
<	2	menor que	Valores discretos e cadeias
<=	2	menor ou igual	
>	2	maior que	
>=	2	maior ou igual	
xor	0	"ou" exclusivo	
or	0	"ou" lógico	
orif	0		Valores booleanos e cadeias
and	1	"e" lógico	de *bits*
andif	1		
not	5	negação	
in	0	pertinência	
xor	0	diferença simétrica	
or	0	união	
and	1	interseção	Conjuntos
<	2	subconjunto	
<=	2	subconjunto próprio	
>	2	superconjunto	
>=	2	superconjunto próprio	
−	3	diferença	
( )	5	repetição	Cadeias
//	3	concatenação	

Tabela 1: Operadores

```
dcl
    str chars (12) := (4)"xyz";
```

(4)"*xyz*" é equivalente a "*xyzxyzxyzxyz*".

3 Ações

A linguagem CHILL provê ações de controle de fluxo semelhantes às existentes em outras linguagens de programação modernas. Toda ação pode ser precedida de um rótulo (um nome seguido do caractere :). Toda ação composta possui um símbolo terminador (*if*...*fi*, *case*...*esac*, *do*...*od*, *begin*...*end* etc.). Um tratador de exceções pode ser anexado a toda ação que possa causar uma exceção (veja a seção 7).

Ação de atribuição

A ação de atribuição permite que se atribua um valor a uma ou mais variáveis. Por exemplo:

```
c, j := (i * j) mod 10;
```

Um operador diádico fechado (isto é, um operador para o qual o tipo do resultado é equivalente ao tipo dos dois operandos) pode ser usado, indicando que o valor contido na variável do lado esquerdo da atribuição deve ser "combinado" com o valor resultante da expressão do lado direito e o resultado deve ser armazenado na variável. No exemplo abaixo, o valor contido em *i* é multiplicado por *2* e o resultado é armazenado em *i*:

```
i *:= 2;
```

Ação *if*

A ação *if* tem uma parte *elsif* opcional, de uso adequado quando várias condições devem ser testadas seqüencialmente. Por exemplo:

```
teste:
   if a > b and a > c
      then /* ações 1 */
   elsif b > c
      then /* ações 2 */
      else /* ações 3 */
   fi teste;
```

Ação *case*

Uma ação *case* pode ser constituída por diversos seletores, formando uma tabela de decisão. A tabela tem que ser completa, isto é, todos os valores possíveis devem ser especificados nos rótulos das alternativas da ação. Uma alternativa *else* pode ser especificada; a sua lista de ações é executada se nenhuma das outras alternativas especificar rótulos que "casem" com os valores dos seletores. Uma faixa de valores discretos pode fazer parte de uma especificação de rótulos de uma alternativa, isto é, os valores não precisam ser todos enumerados um a um. Existe um rótulo *else*, que indica todos os valores não especificados nos demais rótulos para o seletor. Existe um rótulo *, que indica qualquer valor para o seletor. Por exemplo:

```
dcl
   i, j int (1:10);

   case i, j of
      (1,3,8),  (*):   /* ações 1 — i ∈ { 1, 3, 8 }          ∧ j ∈ {1:10} */
      (else),   (1:5): /* ações 2 — i ∈ { 2, 4, 5, 6, 7, 9, 10 } ∧ j ∈ {1:5} */
      else             /* ações 3 — i ∈ { 2, 4, 5, 6, 7, 9, 10 } ∧ j ∈ {6:10} */
   esac;
```

Pode-se restringir a faixa de valores possíveis para um seletor. Por exemplo:

```
synmode
   Dias_do_mes = int (1:31);
dcl
   i int;

   teste:
      case i of Dias_do_mes;
         (1:28):   /* ações 1 */
         (29:31): /* ações 2 */
      esac teste;
```

Ação *do*

A ação *do* tem uma cláusula *for* e uma cláusula *while* para controlar repetições, e uma cláusula *with* para acesso a campos de estruturas.

A cláusula *for* pode especificar diversos contadores de repetição. A ação acaba se pelo menos um dos contadores indicar a terminação. A iteração pode ser feita por enumeração de valor ou enumeração de variável. A enumeração de valor pode ser: por passo, em um intervalo discreto e em um conjunto. Por exemplo:

```
newmode
   Indice = int (1:100),
   Pind   = powerset Indice;
dcl
   vetor array (Indice) int;

   passo:
      do for j := 1 by 2 to 100;
         vetor (j) := 0;
      od passo;
   intervalo:
      do for i in Indice;
         vetor (i) := 0;
      od intervalo;
   conjunto:
      do for c in Pind [2, 4, 8, 16, 32, 64];
         vetor (c) := 2;
      od conjunto;
   variavel:
      do for elemento in vetor;
         elemento := 0;
      od variavel;
```

Os contadores de iterações (*j*, *i*, *c* e *elemento*) são implicitamente declarados. O contador de nome *elemento* no exemplo de enumeração de variável denota um elemento da variável *vetor*, como em uma declaração de variável identidade.

As cláusulas *for* e *while* podem ser combinadas. A cláusula *while* é avaliada depois da cláusula *for* e somente se a cláusula *for* não provoca a terminação da ação *do*. Por exemplo:

```
i := 1; /* índice do elemento procurado no vetor */
busca:
   do for k := 1 to 100
      while ( a(k) /= 0) and (a (k) /= x);
         i +:= 1;
   od busca;
```

Ação *exit*

A ação *exit* é usada para sair de uma ação fechada. Na ação *exit* o rótulo é especificado, o que permite sair de mais de um ação fechada aninhada. Por exemplo:

```
busca:
   do while a (i) /= 0;
      if a (i) = x then exit busca; fi;
      i +:= 1;
   od busca;
```

Ação de certificar

A ação de certificar provê um mecanismo para avaliar uma condição. Se a condição avaliada for falsa será causada a exceção *assertfail* (veja a seção 7). Por exemplo:

```
assert a > 0 and b > 0;
```

Ação *result* e ação *return*

A ação *result* serve para estabelecer o resultado de um procedimento. A ação *return* serve para retornar precocemente de um procedimento (veja a seção 4.1). A ação *return* pode também, opcionalmente, estabelecer o resultado do procedimento.

4 Estrutura de Programas

A estrutura de programas é determinada por módulos, regiões, procedimentos, processos e blocos *begin-end*. Eles controlam o tempo de vida de variáveis e procedimentos e a visibilidade de nomes em um programa.

O tempo de vida de uma variável é o tempo durante o qual ela existe em um programa. Um nome é dito *visível* em um certo ponto de um programa se ele pode ser usado nesse ponto. O escopo de um nome constitui todos os pontos onde ele é visível, isto é, onde o objeto denotado é identificado por esse nome.

O termo bloco é usado para denotar diversas construções que restrigem tanto a visibilidade de nomes quanto o tempo de vida de variáveis e procedimentos neles criados. Procedimentos, processos e blocos *begin-end* são exemplos de blocos.

Módulos e regiões restrigem apenas a visibilidade de nomes, constituindo uma forma de proteger esses nomes contra usos não autorizados. Em combinação com os comandos de visibilidade *grant* e *seize* é possível controlar a visibilidade de nomes nas diversas partes de um programa.

Um programa CHILL consiste de uma lista de módulos e/ou regiões que é circundada por um processo imaginário. Esse processo imaginário é iniciado pelo sistema operacional sob controle do qual o programa é executado.

4.1 Procedimentos

Um procedimento é um mecanismo para abstração de tarefas ou operações.

Existem basicamente dois modos de passagem de parâmetros: passagem por valor e passagem por referência. Na passagem por valor, o parâmetro formal se comporta como uma variável local do procedimento, que recebe um valor na entrada do procedimento. Esse valor é avaliado antes da entrada do procedimento. A linguagem define três variações de passagem por valor, especificadas pelos atributos *in*, *out* e *inout*. *In* é o atributo padrão, adotado se nenhum atributo for especificado para o modo de passagem de parâmetros. No caso dos atributos *in* e *inout*, esse valor é o denotado pelo parâmetro real correspondente e, no caso do atributo *out*, é um valor *indefinido*. Se o atributo *out* ou *inout* é especificado, então o parâmetro real tem que ser uma variável e, antes do retorno do procedimento, o valor corrente do parâmetro formal é armazenado nessa variável.

Na passagem por referência, o parâmetro formal se comporta como uma variável identidade, que constitui um novo nome de acesso à variável especificada pelo parâmetro real. O parâmetro real é avaliado antes da entrada do procedimento. O tipo do parâmetro real pode ser um tipo dinâmico se o atributo *dynamic* for especificado.

Um valor ou uma variável podem ser retornadas como resultado de um procedimento. No primeiro caso, um valor tem que ser especificado em qualquer ação *result* e, no segundo caso, uma variável. O valor ou variável retornado é determinado pela última ação *result* executada antes do retorno do procedimento. Se nenhuma ação *result* for executada o procedimento retorna um valor indefinido ou uma variável indefinida. Nesse caso, a chamada do procedimento só pode ser feita como uma ação (de chamada de procedimento) e não como uma (sub-)expressão ou variável.

No exemplo abaixo, o procedimento *Soma* retorna a soma dos valores contidos nos elementos de um vetor—que pode ter um número diferente de elementos a cada chamada do procedimento—passado como parâmetro. Esse vetor não pode ter mais de *m* elementos. Ele pode, por exemplo, ser uma fatia de um vetor de tipo dinâmico.

```
newmode Vetor = array (1:m) int;
syn m = 1000;
Soma:
   proc (v Vetor loc dynamic) returns (int);
      dcl s int := 0;
      do for e in v;
         s +:= e;
      od;
      result s;
   end Soma;
```

No exemplo abaixo, o procedimento *Alguem* retorna um elemento de um vetor de tipo *Empregados* que tem um determinado nome. O vetor e o nome são passados como parâmetros. Foi considerado, por simplicidade, que existe pelo menos um elemento com o nome passado como parâmetro.

```
newmode
   Empregados = array (1:m) Pessoa;
   Pessoa     = struct(nome Nome,
                       cargo Cargo,
                       registro int);
syn m = 1000;

Alguem:
   proc (f Empregados loc dynamic, n Nome) returns (Pessoa loc);
      do for funcionario in f;
         if n = funcionario.nome then
            return funcionario;
         fi;
      od;
   end Alguem;
```

Note o uso do atributo *loc* na especificação do resultado do procedimento. Ele implica que o procedimento retorna uma variável, e não um valor. Portanto uma chamada desse procedimento pode ser colocada no lado esquerdo de uma ação de atribuição. Pode-se fazer por exemplo:

```
dcl
   Programadores Empregados (100);

Alguem (Programadores,"Sr. CHILL").cargo := Chefia;
```

Um procedimento pode propagar exceções, o que siginifica que exceções causadas no domínio do procedimento são consideradas como se causadas no ponto de chamada do procedimento (veja seção 7). Isto permite que o solicitante de uma tarefa ou operação possa tratar as exceções eventualmente dela resultante.

As exceções que podem ser propagadas por um procedimento devem ser explicitamente especificadas na definição do procedimento. No exemplo abaixo, o procedimento *empilha* propaga a exceção *pilha_cheia* se não há espaço disponível na pilha para empilhar um valor. O procedimento *desempilha* propaga a exceção *pilha_vazia* se não há valor na pilha a ser desempilhado.

```
Pilha:
   module
      seize elem;
      grant Empilha, Desempilha;
      syn max = 1000, min = 1;
      dcl pilha array (min:max) elem, indice int := min;

      Empilha:
         proc (e elem) exceptions (pilha_cheia);
            if indice = max then
               cause pilha_cheia;
```

```
            fi;
            pilha (indice) := e;  indice +:= 1;
        end Empilha;

    Desempilha:
        proc ( ) returns (elem) exceptions (pilha_vazia);
            if indice = min then
                cause pilha_vazia;
            fi;
            indice -:= 1;  result pilha (indice);
        end Desempilha;
end Pilha;
```

4.2 Blocos *begin-end*

Um bloco *begin-end* é uma ação que pode conter definições e declarações locais.

A localidade associada à estrutura de blocos de linguagens de programação é uma característica poderosa para se obter programas mais seguros, legíveis, estruturados e de mais fácil manutenção. Um bloco *begin-end* permite ao programador tornar manifesto na estrutura do programa a associação entre uma variável e o código que a utiliza. O programador limita assim o trecho de influência dessa variável, reduzindo a interferência entre partes do programa. Por exemplo:

```
MDC:
    proc (dividendo, divisor int) returns (int)
    exceptions (argumento_invalido);
        assert dividendo > 0 and divisor > 0;
        if dividendo > divisor then
            Troca (dividendo, divisor);
        fi;
        calcula_mdc:
            begin
                dcl resto int := dividendo mod divisor;
                do while resto /= 0;
                    dividendo := divisor; divisor := resto;
                    resto := dividendo mod divisor;
                od;
            end calcula_mdc;
        result divisor;
    end
        on (assertfail): cause argumento_invalido;
    end MDC;
```

4.3 Módulos e Regiões

Módulos e regiões constituem as principais unidades de composição da estrutura de um programa CHILL. Eles constituem a base para:

- o controle da visibilidade (e conseqüentemente do uso) de nomes;
- o encapsulamento e a construção de tipos abstratos de dados;

- a decomposição de um programa em partes que podem ser desenvolvidas de forma segura e independente;
- no caso de regiões, o acesso mutuamente exclusivo aos objetos declarados localmente à região, durante a execução concorrente de processos (veja a seção 5).

Um módulo (ou região) pode ser considerado como uma unidade de programa que define uma fronteira de visibilidade. Nomes criados no domínio de um módulo (ou região) não são visíveis externamente, e nomes visíveis externamente não são visíveis no seu interior. Essa fronteira de visibilidade só pode ser "atravessada" usando os comandos de visibilidade *grant* e *seize*. O comando *grant* provê uma maneira de estender a visibilidade de nomes em um domínio de um módulo (ou região) para um domínio que o engloba diretamente. O comando *seize* provê uma maneira de extender a visibilidade de nomes visíveis no domínio de um grupo para os domínios de módulos (ou regiões) encaixados diretamente nesse grupo.

Um exemplo de implementação de uma pilha como um tipo abstrato de dados é dado na seção 4.1.

4.4 Decomposição de Programas

Quando programas grandes são desenvolvidos, é adequado subdividi-los em partes menores e desenvolvê-las separada e paralelamente. As estratégias para compilação dessas partes podem ser classificadas em:

Compilação Independente: O texto fonte de uma unidade a ser compilada pode conter "definições e/ou declarações externas" ou "definições e/ou declarações globais". Quando as unidades são reunidas para formar um programa, nenhum teste é feito para verificar se as "definições e/ou declarações externas" estão de acordo entre si e com as "definições e/ou declarações globais".

Compilação Separada: O compilador tem acesso a arquivos que contêm informações sobre os nomes exportados pelas unidades. Ele pode verificar, assim, que o uso de um nome está compatível com a sua definição/declaração.

Compilação em Partes: As estratégias anteriores são combinadas. O compilador tem acesso diretamente a uma parte ou a uma especificação dessa parte. A consistência da especificação com a (implementação da) parte pode ser verificada quando as partes são agrupadas para formar um programa ou progressivamente à medida em que são feitas as compilações. A compilação em partes pode ser chamada de "compilação segura e independente".

A estratégia de compilação em partes é adotada pela linguagem CHILL.

Módulos e regiões são as unidades nas quais um programa CHILL completo pode ser subdividido e compilado em partes. Os textos fonte dessas partes são indicados por construções denominadas partes remotas. A linguagem define a sintaxe e a semântica de programas completos, nos quais cada ocorrência de uma parte remota é virtualmente substituída pelo texto fonte por ela referido.

A idéia básica do esquema de compilação em partes definido pela linguagem é a de que, quando uma parte de um programa é compilada, as *ocorrências definidoras*[5] contidas em partes do pro-

[5]Construções da linguagem onde nomes e os objetos por eles denotados são criados. Por exemplo, uma declaração de variável.

grama não disponíveis ao compilador são substituídas por ocorrências definidoras virtuais, ou, usando o termo definido na linguagem, *quase*-ocorrências definidoras. *Quase*-ocorrências definidoras ocorrem em módulos ou regiões de especificação e contextos. Essas partes não disponíveis ao compilador são módulos ou regiões que constituem um programa que foi subdividido para programação em partes. Os exemplos a seguir ajudam a esclarecer o exposto acima.

Considere o seguinte esqueleto de um programa CHILL

```
    ⋮
m:
  module
    ⋮
  end m;
    ⋮
```

Caso seja desejado programar a parte (módulo) *m* separadamente, é necessário introduzir um módulo de especificação que descreve essa parte:

```
    ⋮
m:
  spec module
    ⋮
  end m;
    ⋮
```

O módulo de especificação contém *quase*-comandos cujo efeito é especificar as propriedades estáticas dos nomes exportados pelo módulo *m*.

Considere agora a compilação de uma parte, independentemente de seu contexto real. A fim de compilar a parte *m* isoladamente, é necessária a especificação de um contexto:

```
context
  /* Especificação do contexto para o módulo m */
for
m:
  module
    /* Corpo do módulo m */
  end m;
```

Os *quase*-comandos contidos em módulos ou regiões de especificação e em contextos são formas restritas dos comandos de definição e declaração de dados da linguagem. Eles contêm informações restritas à descrição das propriedades estáticas dos nomes que eles definem. Por exemplo, uma *quase*-definição de procedimento não contém o corpo do procedimento, que não contribui para definir as propriedades estáticas do procedimento.

Partes remotas constituem um meio para representar o texto fonte de um programa como um conjunto de arquivos interconectados. Elas podem ser regiões ou módulos remotos, especificações remotas ou contextos remotos. Um exemplo de contexto remoto é:

```
context remote m_ctx
for
m:
   module
      /* Corpo do módulo m */
   end m;
```

O designador de texto *m_ctx* é usado para acesso ao texto fonte que deve ser um texto de um contexto não remoto.

O programa com regiões/módulos remotos, especificações remotas ou contextos remotos é equivalente ao programa obtido pela substituição de cada parte remota pelo texto fonte CHILL referido pelo seu designador de texto.

Um módulo de especificação pode ser um *módulo de especificação simples* ou um *módulo de especificação bijetor*. No caso de módulos de especificação simples, não precisa existir uma correspondência biunívoca entre um módulo de especificação e sua parte correspondente de implementação. Ou seja, para um único módulo de especificação podem existir diversos módulos de implementação e, vice-versa, vários módulos de especificação podem corresponder a um único módulo de implementação. No caso de módulos de especificação bijetores, existe uma correspondência biunívoca entre um módulo de especificação e a sua parte de implementação. Neste caso, são *quase*-ocorrências definidoras apenas aquelas especificadas em declarações de variáveis e definições de procedimento ou processo. Definições de tipos, sinônimos e sinais, no entanto, são reais. Isto implica em que elas não precisem ser repetidas no domínio real correspondente, evitando assim a duplicação de definições—no *quase*-domínio e no domínio real—que é indesejável e uma fonte de erros para o programador.

5 Execução Concorrente

Execução concorrente ocorre quando duas ou mais partes de um programa são executadas ao mesmo tempo—i.e., em paralelo. A execução é dita paralela sob o ponto de vista lógico, pois execução paralela não significa necessariamente que ela ocorra fisicamente ao mesmo tempo.

Processos são unidades de execução concorrente da linguagem. Eles são descritos na seção 5.1.

Quando vários processos são requeridos para a realização de uma tarefa comum, é necessário que eles cooperem entre si [3]. Isto significa que os processos precisam estar sincronizados e possam trocar informações em determinados pontos de sua execução. Outra necessidade de coordenação surge quando vários processos competem por um recurso comum. Neste caso os processos precisam ser escalonados de forma que somente um processo consiga acesso ao recurso em um dado instante. Os mecanismos de comunicação e sincronização de processos são descritos nas seções 5.2 e 5.3.

5.1 Processos

Um processo é a parte dinâmica do programa que pode ser executada concorrentemente com outros processos do programa. Um programa CHILL consiste de um ou mais processos. Dentro de um processo, contudo, a execução é sempre sequencial.

Processos são declarados de modo semelhante a procedimentos. A passagem de parâmetros é análoga à de procedimentos[6].

Uma ativação (ou instância) de um processo é criada através da avaliação de uma expressão *start*; a avaliação desta expressão devolve um valor de tipo instância. Este valor pode ser atribuído a uma variável de tipo instância. Um processo pode ser ativado mais de uma vez, com parâmetros possivelmente diferentes. Cada ativação de um processo cria uma nova instância do processo, que pode ser identificada por um valor unívoco de tipo instância. Valores de instâncias podem ser usados para comunicação e sincronização entre processos.

O programa CHILL em execução é chamado de processo imaginário mais externo e é considerado como criado pela avaliação de uma expressão *start* executada pelo sistema operacional sob controle do qual este programa é executado.

Como exemplo de ativações de processos, seja:

```
module
  dcl
     xinst, yinst instance;

     p:
        process (f int);
           /* Corpo do processo p */
        end p;

     xinst := start p(1);
     yinst := start p(2);
     start p(3);
end;
```

Um processo pode terminar (antes de se atingir o fim do corpo do processo) executando uma ação *stop*. Há uma restrição para a terminação do processo imaginário mais externo. Para que o processo imaginário mais externo termine, todos os processos por ele criados devem ter terminado.

Um processo está sempre, ao nível da linguagem CHILL, em um de dois estados: "ativo" ou "em espera". A transição de "ativo" para "em espera" é chamada de retardamento de um processo e o inverso de reativação de um processo. A ativação de um processo se dá pela avaliação de uma expressão *start*. O término de um processo se dá pela execução da ação *stop*, ou atingindo o fim do corpo do processo[7] (*stop* implícito).

Quando um processo está ativo, ele pode ficar em estado de espera ao executar uma ação como a de espera por um evento (*delay*) ou sinal (*receive signal case*). Um processo pode retardar a

[6] No entanto, ao se fazer a ativação de um processo não se "espera" pelo término do processo criado; segue-se que passagem de parâmetros com atributos *inout* e *out* não são usadas.

[7] Ou atingindo o fim de um tratador de exceções anexado à definição do processo—veja seção 6.

sí próprio mas não outro processo.

Quando um processo está em estado de espera, ele pode ser reativado se outro processo executar uma ação como a que indica a ocorrência de um evento (*continue*) ou a que envia um sinal (*send signal*).

5.2 Exclusão mútua

Uma região (ou região crítica) é um mecanismo oferecido pela linguagem que garante aos processos em execução o acesso mutuamente exclusivo a operações consideradas críticas. Essas operações aparecem como procedimentos internos à região e por ela exportados.

Para resolver problemas de acesso mutuamente exclusivo, o mecanismo de exclusão mútua provido por regiões é suficiente. Entretanto, o programador não tem controle sobre a sequência na qual os processos entram na região e acessam os recursos compartilhados. Caso isso seja desejado, variáveis de tipo evento devem ser usadas (veja a seção 5.3).

Do ponto de vista da estrutura de um programa, regiões são semelhantes a módulos. As regras de visibilidade válidas para módulos o são também para regiões, com a ressalva de que regiões não podem exportar objetos de dados internos à região. O acesso a estes objetos só pode ser feito internamente à região através dos procedimentos por ela exportados (chamados de procedimentos críticos).

Em um dado instante, no máximo um procedimento crítico de uma dada região pode estar sendo executado (um procedimento crítico não pode chamar outro procedimento crítico da mesma região[8]).

5.3 Comunicação e sincronização

O tipo evento é um tipo pré-definido na linguagem. Variáveis de tipo evento (ou simplesmente eventos) podem ser declaradas. Eventos provêem uma maneira de se obter sincronização entre processos através das ações *continue*, *delay* e *delay case*.

Ao executar uma ação *delay* sobre um dado evento, um processo fica no estado "em espera". Ele pode ser reativado por outro processo que execute uma ação *continue* sobre o mesmo evento. Se existirem vários processos em estado de espera em uma mesma variável de tipo evento, apenas um deles é reativado a cada ação *continue*. Se não existir nenhum processo em estado de espera pela variável de tipo evento especificada, o comando *continue* não tem nenhum efeito—eventos não são "persistentes".

Na definição de um tipo evento, um parâmetro pode ser especificado. Este parâmetro é o número máximo de processos que podem ficar em estado de espera associado a uma variável deste tipo[9].

Nas ações *delay* e *delay case* uma prioridade pode ser especificada. Esta prioridade é usada para selecionar um processo se vários processos estiverem em espera por um mesmo evento.

Como exemplo de uso de eventos e regiões, seja:

[8] Se isto ocorrer indiretamente, o processo estará em "deadlock" e o programa estará incorreto.

[9] Uma exceção (*delayfail*) ocorre se um processo executa um comando *delay* e existem n processos em espera pelo evento, onde n é o parâmetro especificado na declaração do evento.

```
Alocador_Recursos:
  region
    grant Aloca, Dealoca;
    syn min = 0, max = 9;
    newmode Recursos = int (min:max);
    dcl alocado array (Recursos) bool
        init := | (else):false|;
    dcl recursos_livres event;

    Aloca:
      proc () returns (Recursos);
        do for ever;
          do for i in Recursos;
            if not alocado(i) then
              alocado(i) := true;
              return i;
            fi;
          od;
          delay recursos_livres;
        od;
      end Aloca;

    Dealoca:
      proc (i Recursos);
        alocado(i) := false;
        continue recursos_livres;
      end Dealoca;
  end Alocador_Recursos;
```

O tipo *buffer* é um tipo pré-definido na linguagem. Variáveis do tipo *buffer* (ou simplesmente *buffers*) podem ser declaradas. *Buffers* representam uma área de memória onde os processos podem depositar e retirar mensagens de um certo tipo.

Na definição de um tipo *buffer* o tipo das mensagens que podem ser enviadas e retiradas deve ser explicitado. Um parâmetro (indicando um número máximo de "lugares" que podem conter mensagens) pode ser especificado, em cujo caso um processo é retardado quando tenta depositar um valor em um *buffer* cheio e é reativado quando um lugar torna-se disponível. Um exemplo de definição de um *buffer* é:

```
dcl buf buffer (10) int;
```

Neste exemplo, foi especificado um parâmetro igual a *10*, o que determina o número máximo de valores que o *buffer* pode conter. Os valores devem ser do tipo inteiro.

As operações definidas em variáveis de tipo *buffer* são as ações *send buffer* e *receive case buffer* e a expressão *receive*. A ação *send buffer* coloca um valor em um *buffer*. Se o *buffer* não está cheio, o valor (juntamente com a prioridade) é armazenado no *buffer*. Um *buffer* está cheio se o número de valores nele armazenados é igual ao seu comprimento. Se uma ação *send buffer* é executada quando o *buffer* está cheio, o processo fica em estado de espera. A prioridade é usada para selecionar um valor a ser recebido dentre um conjunto de valores possivelmente existentes em um *buffer*.

A expressão *receive* e a ação *receive buffer case* obtêm valores de um *buffer*. A expressão *receive* retorna um valor retirado do *buffer* especificado[10] Se um processo avalia uma expressão *receive* quando o *buffer* está vazio[11], ele é colocado em estado de espera.

A ação *receive buffer case* pode receber um valor contido em um dentre um conjunto de um ou mais *buffers*. Uma lista de ações pode ser executada, correspondente a cada *buffer*. Uma cláusula *set* pode ser usada para obter o valor da instância do processo que executou a ação *send buffer*. O valor recebido do *buffer* é denotado por um nome, existente para cada alternativa.

Como exemplo de uso de *buffers*, seja:

```
comunicacao:
  module
    dcl buf buffer (n) Info;
    syn n = 1;
    newmode Info = int;

    Produtor:
      process ();
        dcl valor Info;
        do for ever;
          /* produz "valor" */
          send buf (valor);
        od;
      end Produtor;

    Consumidor:
      process ();
        dcl valor Info;
        do for ever;
          valor := receive buf;
          /* consome "valor" */
        od;
      end Consumidor;
  end comunicacao;
```

Assim como *buffers*, sinais provêem uma forma de comunicação e sincronização entre processos. A comunicação é feita através do envio de valores de um processo a outro. Sinais devem ser definidos em comandos de definição de sinal (um sinal não é um tipo e portanto não existem variáveis de tipo sinal). Um sinal define uma função de composição de valores a serem transmitidos entre processos.

As operações definidas sobre sinais são as ações *send signal* e *receive signal case*. A ação *send signal* envia informações de sincronização e possivelmente outras informações de um processo a outro. Uma prioridade pode ser usada para selecionar um sinal a ser recebido dentre um conjunto de sinais especificados em uma ação *receive case*. Uma ação *send signal* nunca causa o retardamento de um processo. Se uma instância de processo é especificada, somente esta instância pode receber o sinal. Neste caso, se o sinal tem um nome de processo associado (especificado na definição do sinal), a instância especificada na ação *send signal* deve ser uma

[10] Ou enviado por um processo no caso de *buffers* de comprimento zero.

[11] Ou se nenhum processo está esperando para colocar um valor no *buffer* no caso deste ter comprimento zero.

instância deste processo[12]. Se um sinal é enviado e nenhum processo está esperando pelo mesmo (em uma ação *receive signal case*), o sinal é mantido até que um processo o receba—sinais são "persistentes".

A ação *receive signal case* especifica o recebimento de um dos sinais especificados nas suas alternativas. Se o sinal tem uma lista de valores, eles são recebidos junto com o sinal. Se nenhum dos sinais especificados existe para ser recebido, as ações da cláusula *else* são executadas; se esta cláusula não existir, o processo é colocado em estado de espera. A cláusula *set* pode ser usada para obter o valor da instância que enviou o sinal.

Como exemplo de uso de sinais, seja:

```
module
   seize Dado;
   grant Push, Pop;
   signal
      Push = (Dado) to Pilha,
      Pedido to Pilha,
      Valor = (Dado);

   Pop:
      proc(individuo instance) returns (Dado);
         send Pedido to individuo; /* Especifica a pilha requerida */
         receive case
            (Valor in res): result res;
         esac;
      end Pop;

   Pilha:
      process ();
         newmode Elemento = struct (info Dado, next ref Elemento);
         dcl top,new ref Elemento := null,
            enviador instance;

         do for ever;
            if top = null then
               receive case
                  (Push in val):
                     top := allocate (Elemento,|val,null|);
               esac;
            fi;
            receive case set usuario;
               (Push in val):
                  new := allocate (Elemento,|val,top|); top := new;
               (Pedido):
                  send Valor(top->.info) to usuario;
                  new := top; top := top->.next;
                  terminate (new);
            esac;
         od;
      end Pilha;
end;
```

[12] A exceção senão ocorre em caso contrário.

6 Supervisão de tempo

As facilidades de supervisão de tempo da linguagem provêem meios de supervisionar eventos que dependem do tempo. Essa supervisão pode ser baseada em intervalos decorridos de tempo, tempo absoluto ou em intervalos cíclicos.

Uma supervisão de tempo pode ser iniciada, pode expirar e pode deixar de existir. Varias supervisões de tempo podem estar associadas a um mesmo processo. Um processo pode ser interrompido somente em pontos temporizáveis durante a sua execução. Quando o programa se encontra num desses pontos[13] e ocorre uma interrupção de tempo, o controle é transferido para um tratador de temporização.

A linguagem provê rotinas pré-definidas destinadas à suspensão de um processo que fica em espera por uma ou mais interrupções de tempo—*wait*— e à detecção da expiração de uma supervisão de tempo—*expired*.

Os tipos usados para a supervisão de tempo são os tipos duração[14] que define valores que representam períodos de tempo, e o tipo tempo absoluto[15] que define valores que representam um determinado ponto no tempo. Valores de tempo são criados pelas rotinas pré-definidas: *abstime*, *millisecs*, *secs*, *minutes*, *hours*, *days*.

No exemplo abaixo, uma temporização de 3 segundos é iniciada para a espera do recebimento do sinal *s*. A temporização deixa de existir se o sinal *s* é recebido antes de 3 segundos. Se a temporização expirar, o controle é transferido para as ações do tratador *timeout*.

```
after secs(3) delay in
   receive case
      (s): /* ações */
   esac
timeout
   /* ações */
end;
```

No exemplo abaixo o sinal *s* é enviado com um atraso inicial de 3 minutos, a cada 5 segundos, durante 2 horas.

```
after minutes (3) in wait ();
timeout
   /* nada */
end;
after hours (2) in
   cycle secs (5) in
      send s to inst.destino;
   end;
timeout
   /* nada */
end;
```

[13] Um processo torna-se temporizável durante a execução de ações específicas como *delay*, *receive case* e as rotinas pré-definidas *wait* e *expired*.

[14] *duration* é um nome pré-definido de tipo duração

[15] *abstime* é o nome pré-definido do tipo tempo absoluto

7 Tratamento de exceções

A linguagem CHILL permite o tratamento de exceções (condições excepcionais que ocorrem durante a execução do programa). Estas exceções são causadas ou pela violação de uma condição dinâmica[16] imposta pela linguagem, ou explicitamente no programa sob condições determinadas.

Ao ocorrer uma exceção, se existir um tratador definido para a mesma, o controle é para ele transferido; senão, o controle pode ser transferido para um tratador padrão definido pela implementação; senão, o programa estará errado.

Os tratadores são anexados a comandos—como ações, declarações com iniciação, regiões e definições de processos e procedimentos—e não diretamente às operações que poderiam causar a exceção. Por exemplo:

```
a := b * c
  on
     (overflow) : /* Ações */
  end;
```

A operação de multiplicação pode causar a exceção *overflow*. O lugar mais "próximo" para um tratador desta exceção é no fim da ação de atribuiçao[17].

A posição de um tratador em um texto de programa determina o escopo no qual este tratador é designado e para onde o controle é transferido ao se chegar ao fim do tratamento. Por exemplo, um tratador anexado a uma ação de atribuição só pode tratar exceções que ocorrem nesta ação, e a execução da ação de atribuição termina ao se chegar ao fim do tratamento.

Exceções causadas no escopo de um procedimento podem ser tratadas interna ou externamente ao mesmo. Para que uma exceção possa ser tratada externamente, ela deve ser mencionada explicitamente no cabeçalho do procedimento ("propagação explícita de exceções"). Uma exceção propagada é considerada como causada no ponto de chamada do procedimento e pode ser aí tratada ou em um ponto externo ao ponto de chamada. Um exemplo de definição de um procedimento que propaga uma exceção é dado na seção 4.1.

Para cada comando são conhecidas, estaticamente, as exceções que podem ocorrer e os respectivos tratadores—se eles estiverem presentes.

Uma alternativa *else* pode ser especificada em um tratador. Esta alternativa corresponde a qualquer exceção que não as mencionadas explicitamente nas outras alternativas do tratador.

Toda exceção em CHILL tem um nome. Este nome pode denotar uma exceção definida pela linguagem, pela implementação ou pelo usuário no seu programa. Exceções definidas pela linguagem ou pela implementação podem ser causadas por condições anormais durante a execução do programa ou através da ação *cause*. Exceções definidas pelo usuário só podem ser causadas através da ação *cause*.

Algumas das exceções pré-definidas na linguagem CHILL são:

- *assertfail*: a condição na ação *assert* é falsa.

[16] Condição dinâmica é aquela que só pode ser verificada durante a execução do programa.

[17] Isto advém do fato de que CHILL é uma linguagem orientada para ações e não para expressões (como, por exemplo, ALGOL 68).

- *empty*:
  - "de-referenciar" um valor *null* de uma referência;
  - *min* ou *max* aplicados em um conjunto vazio;
  - chamada de procedimento de valor *null*;
  - envio de um sinal, ou valor de um *buffer*, para uma instância de valor *null*;
  - chamada a *terminate* com valor *null* de uma referência.
- *extinct*: sinal enviado a uma instância não existente.
- *overflow*: "estouro" em operações com inteiros.
- *rangefail*: valor fora dos limites de um tipo.
- *tagfail*: acesso a um campo não existente de um tipo estrutura com partes variantes.

O tratador de uma exceção que pode ocorrer em um determinado ponto P do programa é procurado na seguinte ordem:

- Anexado à ação A que engloba P diretamente. Ao final do tratamento, a execução da ação A termina. Por exemplo:

```
a(i) := 1
   on (rangefail): /* Ações */
   end;
```

- Anexado à ação composta, módulo ou região que engloba A diretamente. Ao final do tratamento, a execução desta ação fechada, módulo ou região termina. Por exemplo:

```
do while i < j;
   /* . . . */
   a(j-i) := 1;
   /* . . . */
od
   on (rangefail): /* Ações */
   end;
```

- Se A está no domínio de um procedimento, então:
  - Anexado à definição do procedimento. Ao final do tratamento, a execução do procedimento termina.
  - Senão, se a exceção é mencionada na lista de exceções do procedimento, então a exceção é considerada como causada no ponto de chamada.
  - do contrário, não há tratador.
- Se A está no domínio de um processo, então:
  - Anexado à definição do processo. Ao final do tratamento, a execução do processo termina.
  - do contrário, não há tratador.
- Se A está em uma alternativa de um tratador, uma exceção ocorrida em A é considerada como ocorrida no comando ao qual este tratador está anexado e como se o tratador não tivesse sido especificado.

8 Considerações Finais

CHILL é hoje uma linguagem de programação "madura" [10]. Ela é usada por um grande número de programadores[18] que desenvolvem e dão manutenção em inúmeros sistemas de telecomunicações no mundo inteiro. CHILL é considerada também uma linguagem "viva". No seu processo de evolução, ela vem sofrendo simplificações, modificações e extensões que procuram torná-la mais uniforme e mais ortogonal, assim como aumentar seu poder de expressão. CHILL se distingue de outras linguagens por prover diversos mecanismos e facilidades, que em geral não são englobados por uma só linguagem, e por conseguir integrar esses mecanismos de maneira ortogonal. Tendo em vista novas tendências e técnicas na área de linguagens de programação e um incremento no número de requisitos para o desenvolvimento de sistemas dedicados (eficiência, extensibilidade, portabilidade etc.), as seguintes áreas da linguagem são reconhecidas como possíveis extensões:

Aritmética real: O objetivo principal é o de prover suporte a aplicações numéricas e estatísticas

Generalidade: As vantagens de se incluir generalidade são: aumento na reusabilidade de *software*, controle e confiabilidade maiores do *software* e abstração de dados;

Orientação a objetos: Conceitos de classes, tipos abstratos e herança (extensões de tipos) que caracterizam a diferença básica entre os paradigmas de programação procedural e orientada a objetos.

Referências

[1] UIT/CCITT, *CCITT High Level Language (CHILL), Recommendation Z.200*, Genebra, 1988.

[2] C. H. Smedema, *Some Issues in the International Standardization of CHILL and Ada™*, Computers & Standards, Vol. 4, Pages 95–100, 1985.

[3] C. A. R. Hoare, *Communicating Sequential Processes*, Communications of the ACM, Vol. 21, No. 8, Pages 666-677, August 1978.

[4] P. Kropf, *A comparision between the languages CHILL and OCCAM*, 4th CHILL Conference, Munich, pg. 145–151, 1986.

[5] C. Breeus and L. Preumont, *Piecewise Programming Features: Analysis and Implementation*, 4th CHILL Conference, Munich, pg. 169–177, 1986.

[6] M. Chandrasekharan and Barbara G. Moore, *A dicipline for designing abstractions: CHILL's style compared with ADA and CLU*, 3rd CHILL Conference, Cambridge University, pg. 109–114, 1984.

[7] J. F. H. Winkler, *The Realization of Data Abstractions in CHILL*, 3rd CHILL Conference, Cambridge University, pg. 175–181, 1984.

[18] Estudos recentes calculam que esse número seja da ordem de 10 a 15 mil.

[8] H. F. van Rietschote, *The Influence of CHILL on the Programming Environment*, 3rd CHILL Conference, Cambridge University, pg. 93 97. 1984.

[9] R. H. Bourgonjon, *Programming Languages, Environments and CHILL*, 2nd CHILL Conference, Chicago, pg. 1–6, 1983.

[10] R. A. Conroy, *Impacts of CHILL on system design*, 2nd CHILL Conference, Chicago, pg. 53–55, 1983.

[11] K. Rekdal, *CHILL, The Standard Language for Programming SPC Systems*, IEEE Transactions on Communications, Vol. COM-30, Nº 6, Junho 1982.

[12] C. H. Smedema, M. Boasson, *An Introductions to the Programming Languages Pascal, Modula, CHILL and Ada*, Prentice Hall, Englewood Cliffs, 1983.