Universidade Federal de Minas Gerais

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Laboratório de Linguagens de Programação

# Machĭna: A Linguagem de Especificação de ASM

by

**Fabio Tirelo** **Roberto da Silva Bigonha**
**Marcelo de Almeida Maia** **Vladimir Oliveira di Iorio**

**LLP 008/99**

**Caixa Postal, 702**

**30.161-970 – Belo Horizonte**

**Minas Gerais – Brazil**

**August, 1999**

**Resumo**

Este texto contém a definição da Linguagem de Programação Machĭna, que é uma linguagem baseada no paradigma de Programação Algébrica, derivado do modelo de Máquinas de Estado Abstratas (ASM). Nesta definição, apresentamos a semântica informal de Machĭna, descrevendo os aspectos semânticos da linguagem por meio de linguagem natural e exemplos de utilização das construções da linguagem. Ao final, apresentamos vários exemplos de programas escritos em Machĭna.

# Sumário

# Capítulo 1

# A Linguagem Machĭna

## 1.1 Introdução

Neste capítulo, apresentamos a linguagem Machĭna, que é baseada na metodologia de Máquinas de Estado Abstratas (ASM), com suporte à modularidade, tipos e construções de alto nível. Um programa em Machĭna tem o estilo de Programação Algébrica, isto é, o estado da computação é descrito por uma álgebra e existe uma regra de transição que promove a mudança de estados. Desta forma, a escrita de um programa algébrico consiste em definir:

- um vocabulário, que é o conjunto de símbolos da especificação, isto é, o conjunto dos identificadores e operadores conhecidos;
- uma álgebra inicial, que é a interpretação inicial do vocabulário;
- uma regra de transição, que promove a transição de estados (álgebras).

## 1.2 Comentários

Comentários em Machĭna podem ser de dois tipos:

- um texto que se inicia com `/*` e termina com `*/`, podendo haver aninhamento;
- um texto que se inicia com `//` e vai até o fim da linha.

## 1.3 Literais

Os literais de Machĭna são:

- caracteres do conjunto ASCII, representados por um símbolo entre apóstrofos, p.ex. `'5'` e `'a'`, ou por um código denotando caracteres especiais, como `'\n'`, `'\0'`, etc.
- números inteiros – números inteiros podem ser escritos em decimal (seqüência de dígitos), octal (`0` seguido de seqüência de dígitos de `0` a `7`) ou em hexadecimal (`0x` seguido de seqüência de dígitos ou letras de `A` a `F` ou `a` a `f`). Os números inteiros estão no intervalo [-2147483648,2147483647].
- números de ponto flutuante – seqüência de dígitos, seguida por uma parte fracionária opcional, seguida por uma parte de expoente opcional; a parte fracionária consiste em um ponto (`"."`) seguido por uma seqüência de dígitos; a parte do expoente consiste em

`e` ou `E`, seguido por um sinal opcional (`+` ou `-`), seguido por uma seqüência de dígitos. Números de ponto flutuante têm valor máximo ±1.79769313486231570E+308 e valor mínimo ±4.94065645841246544E-324, seguindo o padrão IEEE 754 para ponto flutuante.

- cadeias de caracteres – seqüência de caracteres entre aspas, como por exemplo, `"string"`, `"isto e\' um teste"`, `"fim de linha\n"`, etc.;

## 1.4 Palavras Reservadas

As palavras reservadas de Machĭna são:

```
action      algebra   and          as        case
choose      create    default      derived   do
dynamic     else      elseif       end       enum
external    false     forall       if        import
in          init      interleaved  is        let
list        machina   module       nil       not
of          or        otherwise    public    satisfying
self        set       static       stop      then
transition  true      type         with
```

## 1.5 Identificadores

Identificadores em Machĭna são divididos em dois tipos:

- identificadores de tipos e módulos – formados por uma ou mais letras, onde a primeira é maiúscula;
- identificadores de variáveis – formado por uma ou mais letras, onde a primeira é minúscula; podem ser decoradas, isto é, terminadas, com números;

## 1.6 Notação Para Sintaxe

Para descrição da gramática, utilizaremos a BNF estendida [?], ou XBNF. As construções utilizadas são semelhantes à BNF, com as seguintes extensões:

| | |
|---|---|
| \| | Opções |
| { X } | Zero ou mais ocorrências de X |
| [ X ] | Zero ou uma ocorrência de X |

Os caracteres que representam meta-símbolos, se utilizados na gramática, serão escritos em aspas, como por exemplo "{".

## 1.7 Uma Especificação em Machĭna

Uma especificação em Machĭna consiste em uma coleção de um ou mais módulos, denominados *módulos de programa*, ou somente *módulos*. Associado a uma especificação, pode haver uma ou mais "definições de máquina", na qual definimos e disparamos os agentes que executam as regras dos módulos.

## 1.8 Módulos

Um módulo especifica um nome para o programa de um agente, a regra que este agente executa, e sua álgebra subjacente, isto é, símbolos, universos e interpretação dos símbolos. Um módulo tem a forma:

**module** nome-módulo
    **import**:
        elementos importados
    **algebra**:
        álgebra subjacente – tipos, funções, ações, inicializações
    **transition**:
        regra de transição
**end**

Uma definição de máquina especifica o seu nome e possui uma seção de inicializações, onde são os agentes criados e disparados. Cada definição possui a seguinte forma:

**machina** nome-máquina
    declarações do ambiente de execução
    inicializações – criação dos agentes
**end**

Módulos possuem três partes:

- Importações – define os módulos que serão importados (Seção 1.11);
- Álgebra – define a álgebra subjacente ao modelo, contendo os *sorts* ou tipos, as funções, as ações e as regras de inicialização;
- Regra de transição – define a regra de transição do módulo, isto é, o programa dos agentes.

A seção de álgebra é divida nas seguintes subseções:

- Definição de tipos – define tipos (*sorts*) de dados que serão utilizados na especificação (Seção 1.12);
- Funções – agrupa as definições de funções por classe, podendo ser *estática*, *dinâmica* e *derivada*. Por exemplo, uma declaração da forma

  ```
  static
    f := ...
    g := ...
    h := ...
  dynamic
    a := ...
    b := ...
  ```

  define `f`, `g` e `h` como funções estáticas, e `a` e `b` como dinâmicas (Seção 1.17);
- Funções externas – define os cabeçalhos das funções externas (Seção 1.18);
- Ações – define as transições parametrizadas, que são abstrações de regras de transição (Seção 1.20);
- Inicializações – regra de inicialização de funções dinâmicas e definição de funções estáticas (Seção 1.19);

A sintaxe de um módulo é a seguinte:

| | | |
|---|---|---|
| module | ::= | **module** module-name module-body **end** |
| module-body | ::= | [import-part] [algebra-part] [transition-part] [invariant-part] |
| import-part | ::= | **import**: module-import { , module-import } |
| algebra-part | ::= | **algebra**: { algebra-section } |
| transition-part | ::= | [**interleaved**] **transition**: transition-rule |
| invariant-part | ::= | **invariant**: expression |
| module-import | ::= | module-name [imported-list] |
| imported-list | ::= | ( imported-element { , imported-element } ) |
| imported-element | ::= | id \| type-name |
| algebra-section | ::= | type-section |
| | \| | function-section |
| | \| | external-section |
| | \| | action-section |
| | \| | init-section |
| type-section | ::= | **type** type-pdecl { ; type-pdecl } |
| type-pdecl | ::= | [**public**] type-declaration |
| function-section | ::= | class-modifier function-pdecl { ; function-pdecl } |
| class-modifier | ::= | **static** \| **dynamic** \| **derived** |
| function-pdecl | ::= | [**public**] function-declaration |
| external-section | ::= | **external** external-pdecl { ; external-pdecl } |
| external-pdecl | ::= | [**public**] external-declaration |
| action-section | ::= | **action** action-pdecl { ; action-pdecl } |
| action-pdecl | ::= | [**public**] action-declaration |
| init-section | ::= | **init** transition-rule |

## 1.9 Agentes

A cada módulo pode ser associado um conjunto de agentes, que executam a sua regra. Agentes são criados por meio de uma regra especial de nome *create*, como será visto na Seção 1.21.

A regra de transição de um módulo declarada como *interleaved* é executada repetidamente de forma entremeada com a de outras regras. Após cada transição, o controle volta para o módulo máquina, que pode livremente escalar o próximo agente a prosseguir em execução. Transições que não são declaradas como *interleaved* são executadas repetidamente até o seu término.

Agentes podem ser referenciados, de dentro de um programa, por meio de funções, assim como qualquer valor. Para isto, são definidos em Machina o tipo `Agent` e o construtor de tipos **agent of** `M`, onde `M` é um nome de módulo. O tipo **agent of** `M` é o conjunto de agentes que executam a regra do módulo `M`. Mais detalhes sobre este tipo serão mostrados na Seção 1.12.9.

Todo agente possui uma função especial de nome `self` ∈ **agent of** `M`, que retorna a sua identificação, onde `M` é o módulo que contém a regra que o agente executa. Desta forma, representamos informações locais a um agente.

Uma máquina deve definir a criação dos agentes que iniciarão a execução. Considere, por exemplo, a definição dos módulos e a criação dos agentes dada por:

```
module A          module B          module C          machina D
  decls of A        decls of B        decls of C        create a : agent of A
transition:       transition:       transition:         create b : agent of B
  rule of A         rule of B         rule of C         create c : agent of C
end               end               end               end
```

A relação entre os agentes definidos na **machina** `D` e os módulos `A`, `B` e `C` pode ser vista na Figura 1.1.

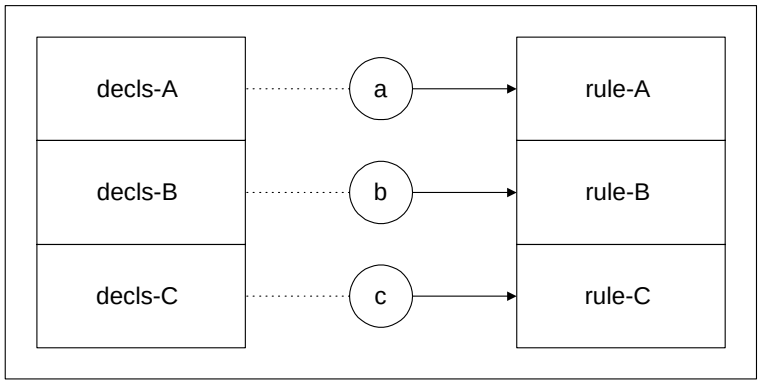


Figura 1.1: Exemplo de espaço de declarações e regras e acesso de agentes.

Na Seção 1.11, mostraremos como estender este modelo para lidar com o compartilhamento de declarações entre módulos.

## 1.10 Término da Execução

A execução termina com a execução do comando **stop**, ou quando a guarda principal não for verdadeira. Por exemplo, a execução da especificação abaixo só terminará com a execução do comando **stop**.

```
module A
...
transition:
  if cond-parada then
    stop
  else
    ...
  end
end
```

Se todas as guardas do programa avaliarem em falso, então a execução terminará. Entretanto, se houver alguma chamada a função externa nas guardas, a execução ficará suspensa, esperando que a função externa seja modificada e alguma guarda se torne verdadeira.

```
module B
external f :  Int;
dynamic g(x :  Int) :  Int;
transition:
  if f = 10 then
    g(10) := 0
  else
    g(10) := 1
  end
end
```

Quando a regra for executada a primeira vez, se o valor da função externa $f$ for 10, o valor de $g$ no ponto 10 será atualizado para 0. Enquanto o valor de $f$ não mudar, a regra `g(10) := 0`

será executada, o que não causará mudança alguma no estado. Neste caso, a execução não será abortada, pois não sabemos se no próximo estado, o valor da função externa será diferente de 10.

## 1.11 Mecanismos de Visibilidade

Um módulo pode importar os elementos (declarações de funções, tipos e ações) de outros módulos. Toda declaração é privada a um módulo, isto é, visível somente dentro do módulo no qual foi feita. Um elemento só não é privado a um módulo se for explicitamente declarado como público, por meio do modificador de visibilidade **public**.

Os elementos públicos de um módulo M1 tornam-se visíveis dentro de outro módulo M2, quando M1 for importado em M2. Isto é feito na parte de importações do módulo M2, de uma das seguintes maneiras:

| | | |
|---|---|---|
| **import:** | `M1` | Torna visíveis todos os elementos públicos de `M1` |
| **import:** | `M1(a,b)` | Torna visíveis somente os elementos públicos `a` e `b` de `M1` |

Da primeira maneira, todos os elementos públicos de `M1` são visíveis dentro de `M2`. Um elemento `a`, público de `M1` é acessado em `M2` da forma `M1.a`. Da segunda maneira, os elementos `a` e `b` de `M1` são diretamente visíveis em `M2`, não sendo necessária a qualificação do módulo para acessar `a` ou `b`, isto é, estes identificadores podem ser acessados diretamente em `M2`, sem a necessidade de especificar o módulo a que pertencem.

Por exemplo, considere os módulos abaixo:

```
module X
  public a;
  b
  ...
end
```

```
module Y
  import:  X;
  public m,n;
  ...  X.a ...
end
```

```
module Z
  import:  X, Y(n);
  ...  X.a ...
  ...  n ...
end
```

O módulo `X` define um elemento de nome `a`, que é visível a partir de outros módulos, enquanto a declaração de `b` é visível somente dentro de `X`, isto é, é privada a este módulo. A linha **import** `X`, no módulo `Y`, importa todos os elementos públicos de `X`, o que significa que o elemento `a`, em `X`, é visível dentro de `Y`, sob o nome de `X.A`. Da mesma maneira, `a` é visível no módulo `Z`, sendo necessária a qualificação do módulo, assim como no módulo `Y`. Entretanto, como a importação do módulo `Y` em `Z` explicitou quais identificadores são visíveis, podemos utilizar o nome `n` em `Z` diretamente, sem a necessidade de escrever `Y.n`. Além disso, observe que o nome `m`, definido como público em `Y` também é visível em `Z`, mas requer completa qualificação, pois não foi explicitado na importação.

É importante ressaltar, ainda, que a cláusula **import** *não causa a duplicação* de declarações; ela apenas torna as declarações públicas de um módulo *visíveis* a outros módulos.

Na Figura 1.2, ilustramos como os agentes que executam a regra definida em um módulo acessam as declarações visíveis do exemplo abaixo.

```
module A
  public ax;
  ay;
transition:
  rule of A
end
```

```
module B
  public bx;
  by;
transition:
  rule of B
end
```

```
module C
  public cx;
  cy;
transition:
  rule of C
end
```

```
machina D
  create a :  agent of A
  create b :  agent of B
  create c :  agent of C
end
```

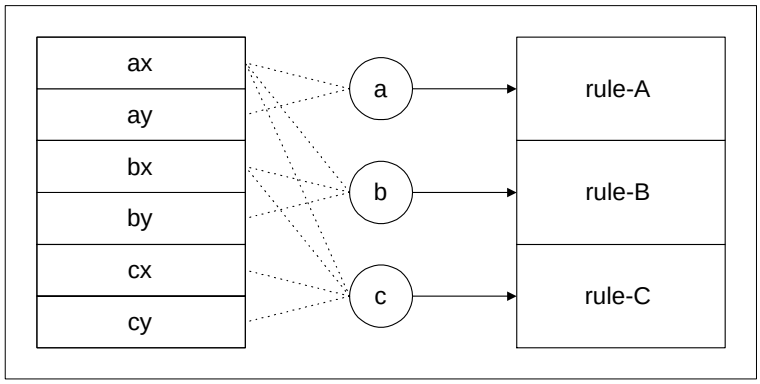


Figura 1.2: Exemplo de Visibilidade entre Módulos. Linhas pontilhadas ligando agentes a declarações significam que o agente pode acessar a declaração e linhas não-tracejadas ligando agentes a regras mostra qual regra de transição o agente está executando.

## 1.12 Declaração de Tipos

Os tipos (ou *sorts*) primitivos de Machina são `Bool`, `Char`, `Int`, `Real` e `String`, `Set`, `List`, `?`.
Os tipos estruturados, isto é, formados por meio de geradores de tipos são:

- enumeração de valores;
- tupla de elementos;
- conjunto de um dado tipo;
- lista de um dado tipo;
- tipo funcional (não permitindo funções de ordem mais alta);
- união disjunta;

Um nome de tipo é uma seqüência de letras, onde a primeira letra é maiúscula.
O tipo `()` (null) é utilizado em definições de tipos recursivos.

### 1.12.1 Enumerações

Tipos enumerados são definidos por meio de enumeração de identificadores, como no exemplo abaixo:

```
RGBColor is enum { red, green, blue }.
```

Enumerações diferentes devem definir identificadores diferentes. Isto significa que não é permitida uma seqüência de declarações da forma:

```
...
enum A is { a, b, c }
enum B is { a, e, i, o, u }
...
```

pois o identificador `a` está sendo definido em duas enumerações diferentes.

### 1.12.2 Uniões Disjuntas

Uma união disjunta é feita em Machĩna utilizando o operador "|". Por exemplo, o tipo `Numero` abaixo é a união disjunta entre `Int` e `Real`:

```
Numero is Int | Real.
```

Observação: em uma união disjunta, os tipos componentes não podem ser equivalentes.

### 1.12.3 Tipo ?

O tipo `?` é definido como a união disjunta de todos os tipos possíveis, com o valor *default* igual a `undef`. Isto significa que, se `x` é do tipo `?`, então `x` pode receber valores de quaisquer tipos. O tipo `?` só possui as operações de atribuição e comparação por `=` ou `!=`. Conversão para o tipo real do valor armazenado deve ser feita via comandos ou expressões **with** e **is**.

### 1.12.4 Intervalos

Uma definição de intervalo é feita como no exemplo:

```
UmACem is 1..100
```

As expressões que determinam os limites inferior e superior do intervalo devem ser manifestas e seus tipos devem ser iguais e discretos.

### 1.12.5 Tuplas

As tuplas devem ser rotuladas. Pode-se utilizar a tupla vazia, `()`, em definições de tipos recursivos. Por exemplo, considere a declaração:

```
Tree is () | (info:Info;left,right:Tree).
```

### 1.12.6 Conjuntos

O construtor de tipos "**set of**" é utilizado para denotar o tipo conjunto de elementos de um mesmo tipo. Por exemplo

```
Intset is set of Int
```

é um tipo cujos elementos são conjuntos de números inteiros. O tipo pré-definido `Set` é definido como

```
Set is set of ?.
```

Isto significa que `Set` é um conjunto genérico, onde seus elementos não precisam de ser do mesmo tipo. As regras para utilização do tipo `Set` são vistas na Seção 1.23.9.

### 1.12.7 Listas

O construtor de tipos "**list of**" é utilizado para denotar o tipo lista de elementos de um mesmo tipo. Por exemplo,

```
Intlist is list of Int
```

é um tipo cujos elementos são listas de inteiros. Assim como em conjuntos, o tipo pré-definido `List` é definido como:

```
List is list of ?.
```

Desta forma, `List` representa uma lista genérica, onde seus elementos não precisam de ser do mesmo tipo. As regras para utilização do tipo `List` são vistas na Seção 1.23.8.

### 1.12.8 Tipos Funcionais

O tipo funcional, cujo construtor é "`->`", é um tipo cujos valores são funções entre domínios dados. Por exemplo, o tipo `Transition` definido abaixo é uma função de `State` × `Element` em `State`:

```
Transition is (State,Element) -> State.
```

Não é permitida a definição de funções de ordem mais alta.

### 1.12.9 Tipos Agentes

O construtor de tipos **agent of** `M` é utilizado para denotar o tipo dos agentes que executam a regra do módulo `M`. Existe também o tipo pré-definido `Agent`, que representa agentes de quaisquer tipos. As operações que podem ser feitas sobre agentes são a criação (Seção 1.21.4), a eliminação (Seção 1.21.4), a atribuição, a passagem como parâmetro, o retorno de função e a comparação por igualdade e diferença.

### 1.12.10 Definições de Tipos

Para permitir a escrita de tipos mutuamente recursivos, Machĩna permite que, na declaração de um tipo, possam aparecer nomes de tipos que ainda não foram declarados. Entretanto, na definição de uma função, os tipos utilizados devem estar completamente definidos.

Em uma definição de tipos, pode haver também uma cláusula **default**, que permite estabelecer o valor *default* para inicialização automática de funções do tipo dado.

Uma seção de declaração de tipos tem a seguinte sintaxe:

| | | |
|---|---|---|
| type-declaration | ::= | type-id **is** type-expression [**default** expression] |
| type-expression | ::= | () |
| | \| | ? |
| | \| | type-name |
| | \| | [**public**] **enum** "{"id { , id } "}" |
| | \| | type-expression "\|" type-expression |
| | \| | expression .. expression |
| | \| | [**public**] ( tuple-element { ; tuple-element } ) |
| | \| | **set of** type-expression |
| | \| | **list of** type-expression |
| | \| | **file of** type-expression |
| | \| | **agent of** type-expression |
| | \| | type-expression `->` type-expression |
| tuple-element | ::= | [**public**] id { , id } [: type-expression] |
| type-name | ::= | `Bool` \| `Char` \| `Int` \| `Real` \| `String` \| `List` \| `Set` |
| | \| | `Agent` \| `InputStream` \| `OutputStream` \| type-id |

## 1.13 Regras de Visibilidade para Tipos

Quando um tipo declarado como público é importado, o seu nome se torna visível no ponto de importação, mas não a sua estrutura. Entretanto, para os tipos tupla e enumeração, pode ser necessária a exportação da estrutura. Para isso, Machĩna define as seguintes regras para exportação de estrutura:

- Para exportar os *elementos de uma enumeração*, escreve-se a palavra-chave **public** antes da palavra **enum**. Por exemplo, no código abaixo, são visíveis externamente os nomes de `A`, `B`, `x`, `y`, `z`, ao passo que os nomes `a`, `b` e `c` não são visíveis, pois pertencem a uma enumeração não-pública.

```
type
  public A is enum {a,b,c};
  public B is public enum {x,y,z};
```

- Para exportar um *campo de uma tupla*, escreve-se a palavra chave **public** antes do nome do campo. A palavra chave **public** antes de uma lista de identificadores separados por vírgula torna público todos os identificadores da lista. Para tornar públicos todos os campos da tupla, pode-se utilizar o modificador **public** antes da expressão de tipo. Por exemplo, no código abaixo, são visíveis externamente os identificadores `A`, `B`, `C`, `b`, `e`, `f`, `m`, `n` e `p`, ao passo que os nomes `x`, `y`, `z`, `a`, `c` e `d` são privados ao módulo.

```
type
  public A is (x:Int;y,z:Real);
  public B is (a:Int; public b:Real; c,d:Bool; public e,f:Char);
  public C is public (m,n:A; p:B)
```

## 1.14 Regras para Dedução de Tipos

Machĩna possui as seguintes convenções para dedução de tipos:

- nomes de tipos são uma seqüência de *letras*, sendo que a primeira letra é *sempre maiúscula*;
- nomes de funções, que aparecem em declarações com especificação de tipos, são uma seqüência de letras, onde o primeiro caractere é *sempre uma letra minúscula*;
- se o tipo de um identificador for explicitado na declaração, como em `a:X`, então o tipo do identificador é o tipo dado e o nome do identificador não pode conter decoração;
- se o tipo não for explicitado na declaração, então as seguintes regras são utilizadas para determinação do tipo:
  - nomes de funções e variáveis que contêm decorações (números no final) possuem o mesmo tipo da função ou variável sem a decoração; por exemplo, os identificadores `x1` e `x2` têm o mesmo tipo de `x`;
  - se o tipo do identificador sem decoração não foi declarado, capitaliza-se a primeira letra do nome do identificador; se houver algum tipo com este nome, então este é o tipo do identificador;
  - caso contrário, o tipo do identificador será `?`.

Por exemplo, considere a seqüência de declarações abaixo:

```
...
A is ...; // Declaração do tipo A
X is ...; // Declaração do tipo X
Y is Int; // Declaração do tipo Y
a,b,c:X; // Todas possuem o tipo X
x; // O tipo de x é X
y := 1; // O tipo de y é Int
z; // O tipo de z é ?
...
```

## 1.15 Equivalência de Tipos

Em Machĭna, tudo que diz respeito a tipos é decidido em tempo de compilação, de modo que, a menos das expressões de conversão, os objetos não levam para a execução informações sobre os seus tipos. A disciplina de tipos de Machĭna é a equivalência estrutural, definida da seguinte maneira:

Dois tipos $A$ e $B$ são *equivalentes* se e somente se uma das seguintes condições abaixo se aplicarem:

- $A$ e $B$ são nomes de tipos idênticos;
- $A$ e $B$ são nomes de tipos distintos, mas cada um é a referência recursiva para a definição de tipo reflexivo que ocorre exatametne na mesma posição correspondente nas suas estruturas de tipos;
- $A$ e $B$ são o tipo `?` de união disjunta de todos os tipos;
- $A$ e $B$ são nomes de tipos distintos, cujas definições visíveis são, respectivamente, as cadeias mais longas $A\mathbf{is}A_1\mathbf{is}A_2\mathbf{is}...\mathbf{is}A_n$ e $B\mathbf{is}B_1\mathbf{is}B_2\mathbf{is}...\mathbf{is}B_m$, tal que $A_i$, para $1 \leq i \leq n$, $B_j$, $1 \leq j \leq n$ são nomes de tipos, e $A_n$ e $B_m$ são os primeiros e únicos nomes repetidos em suas respectivas cadeias;
- $B$ é uma expressão de tipos e $A$ é um nome de tipos, cuja definição visível é tal que $A\mathbf{is}d$ ou $A\mathbf{is}A_1\mathbf{is}A_2\mathbf{is}...\mathbf{is}A_n\mathbf{is}d$, onde $A_i$, para $1 \leq i \leq n$, são nomes de tipos, e a expressão de tipos $d$ é equivalente à expressão de tipos $B$;
- $A$ é uma expressão de tipos e $B$ é um nome de tipos, cuja definição visível é tal que $B\mathbf{is}d$ ou $B\mathbf{is}B_1\mathbf{is}B_2\mathbf{is}...\mathbf{is}B_n\mathbf{is}d$, onde $B_i$, para $1 \leq i \leq n$, são nomes de tipos, e a expressão de tipos $d$ é equivalente à expressão de tipos $A$;
- $A$ e $B$ são domínios de listas, $A$ é da forma **list of** $A_1$ e $B$ é da forma **list of** $B_1$, e $A_1$ é equivalente a $B_1$;
- $A$ e $B$ são o tipo polimórfico de listas vazias;
- $A$ e $B$ são domínios de tuplas, $A$ é da forma $(A_1, ..., A_n)$, $B$ é da forma $(B_1, ..., B_n)$ e, para $1 \leq i \leq n$, $A_i$ é equivalente a $B_i$; nomes de campos não são relevantes para a determinação da equivalência;
- $A$ e $B$ são uniões disjuntas, $A$ é da forma $A_1|...|A_n$, $B$ é da forma $B_1|...|B_n$ e, para $1 \leq i \leq n$, $A_i$ é equivalente a $B_i$;
- $A$ e $B$ são tipos funcionais, $A$ é da forma $A_1$`->`$A_2$, $B$ é da forma $B_1$`->`$B_2$, $A_1$ é equivalente a $B_1$ e $A_2$ é equivalente a $B_2$.

## 1.16 Compatibilidade de Tipos

Dizemos que um tipo $A$ é compatível com um tipo $B$, se uma expressão do tipo $A$ puder ser usada em um lugar onde é esperada uma expressão do tipo $B$. Por exemplo, se definimos que $x$ é do tipo $A$ e $y$ é do tipo $B$, a atribuição $x := y$ só é possível se $A$ for compatível com $B$.

A regra de compatibilidade de Machĭna é a seguinte: um tipo $A$ é compatível com um tipo $B$ se e somente se $A$ e $B$ forem equivalentes ou $B$ é uma união disjunta de tipos, da qual $A$ é um dos componentes. Por exemplo, nos casos abaixo, $A$ é sempre compatível com $B$:

```
A is (a,b:Int); B is (x,y:Int);
A is ...; B is ...  | A | ...;
A is ...; B is ?;
```

Os casos especiais de compatibilidade são relativos aos tipos genéricos. Supondo que $T$ é um tipo qualquer e $M$ é um nome de módulo, as regras de compatibilidade para estes tipos são as seguintes:

- O tipo **list of** $T$ é compatível com `List`;
- O tipo **set of** $T$ é compatível com `Set`;
- O tipo **agent of** $M$ é compatível com `Agent`.

## 1.17 Declaração de Funções

As funções em Machĭna podem ser:

- estáticas – são as que não podem sofrer atualizações e nem acessar funções dinâmicas ou externas;
- dinâmicas – são as que podem sofrer atualizações;
- derivadas – são funções que não podem sofre atualizações, mas podem acessar funções dinâmicas ou externas;
- externas – são funções definidas e atualizadas no ambiente externo.

A classe da função depende da seção onde ela foi declarada. Por exemplo, uma função estática deve ser declarada na seção **static**. O corpo de uma função estática ou derivada pode ser definido tanto na seção de declarações quanto na seção **init** (Seção 1.19).

O ambiente de avaliação do corpo de funções estáticas é formado pelas outras funções estáticas já definidas e pelos seus parâmetros. Funções externas ou dinâmicas não podem ser chamadas a partir do corpo de uma função estática.

O nome da função pode servir para determinar o tipo de retorno da função, caso ele não seja explicitado, como mostrado na Seção 1.14. Da mesma forma, o nome de um parâmetro pode servir para determinar o seu tipo. Quando o tipo de algum elemento não é especificado e o compilador não conseguir deduzi-lo a partir de sua definição, o seu tipo será "?".

Abaixo, mostramos a sintaxe de definição de funções e relações:

| | | |
|---|---|---|
| function-declaration | ::= | declared-element [: type-expression] [:= expression] |
| declared-element | ::= | id { , id } |
| | \| | id ( parameters ) |
| parameters | ::= | parameter { , parameter } |
| parameter | ::= | id : type-expression \| type-expression \| id |

## 1.18 Declaração de Funções Externas

Uma função externa é uma função definida externamente. Como o seu valor é definido pelo ambiente externo, elas não possuem um corpo para ser avaliado. Desta forma, a definição de uma função externa contém somente a especificação do sua assinatura.

A sintaxe para declaração de funções externas é dada por:

external-declaration ::= declared-element [: type-expression]

Machĭna permite que se escreva um programa em outra linguagem, como C, que avalie uma função externa e retorne o valor para a especificação. Para isso, definimos o protocolo de comunicação de funções externas abaixo:

### Protocolo de Comunicação de Funções Externas

Funções externas podem ser funções escritas em C, tais que, dados os argumentos, ela retorna o valor desejado. Desta maneira, é definido um protocolo para a passagem de parâmetros e o valor de retorno das funções, de modo que possamos obter precisão na avaliação das funções externas. O protocolo é seguinte:

- para cada parâmetro da função externa, existe um parâmetro na função escrita em C;
- a ordem dos parâmetros nas duas funções deve ser a mesma;
- os tipos dos parâmetros e de retorno na função escrita em C devem seguir o tipo do cabeçalho definido para a função externa, seguindo a seguinte equivalência:
  - `Bool`, `Char` e enumerações equivalem ao tipo `char` de C;
  - `Int` equivale ao tipo `int` de C;
  - `Real` equivale ao tipo `double` de C;
  - `String` equivale ao tipo `char*` de C;
  - um tipo tupla equivale a uma estrutura de C, de modo que os campos da tupla seguem este protocolo de equivalências com os campos da estrutura na implementação em C;
  - um tipo lista é recebido e retornado como um arranjo em C, cujo tipo dos elementos correspondem segundo este protocolo.

Por exemplo, os cabeçalhos de funções externas abaixo são equivalentes aos cabeçalhos de funções em C dados em seguida.

```
/* Em Machĩna */
type
  T is (Int, Bool)
external
  f(x:Int):Int;
  g(x,y:Int;c:Char):Bool;
  h(a:Int;b:T):T;
transition:
  ...  x := f(10) ...
  ...  y := g(a,b,c) ...
  ...  z := h(1,d) ...


/* Em C */
typedef int equivT; ...    * equivT;
char g(int x, int y, char c);
equivT h(int a, equivT b);
...  x = f(10) ...
...  y = g(a,b,c) ...
...  z = (1,d) ...
```

## 1.19 Definição das Regras de Inicialização

A seção **init** é utilizada para definição de valores iniciais de funções e disparo de agentes. É da forma

**init** regra

e tem o efeito de executar a regra definida. Esta regra é executada da mesma forma que qualquer regra, e os valores retornados por funções chamadas é o valor que definido na declaração da função ou o valor *default* do tipo de retorno da função.

Na seção **init**, pode-se utilizar também a regra **create** para criar os agentes que irão executar a regra de transição do módulo.

A sintaxe é dada por:

init-part ::= **init** transition-rule

## 1.20 Declaração de Abstrações de Regras

Uma abstração de regras de transição é semelhante a um procedimento de uma linguagem imperativa como Pascal. Chamaremos este tipo de abstração de ação. Uma ação pode ser parametrizada. Abaixo mostramos a sintaxe de uma ação:

action-declaration ::= id [(parameters)] := transition-rule **end**

As regras para dedução dos tipos dos parâmetros de uma transição é idêntica à dedução dos tipos dos parâmetros de uma função. Qualquer inconsistência de tipos causará um erro de compilação.

## 1.21 Seção de Definição de Regras de Transição

A seção de definição de regras pode aparecer somente em módulos de programas, o que significa que não pode aparecer em um módulo Machĭna.

As regras em Machĭna são as regras definidas no Lipari Guide, acrescidas de outras regras, criadas com o objetivo de tornar a especificação mais simples e legível.

Abaixo, mostramos a sintaxe para as regras de transição:

transition-rule ::= basic-rule
| forall-rule
| choose-rule
| create-rule
| destroy-rule
| stop-rule
| let-rule
| case-rule
| with-rule
| action-call-rule

### 1.21.1 Regras Básicas

As regras básicas são as regras de atualização, bloco e condicional. A sintaxe é dada por:

basic-rule ::= id [(arguments)] := expression
| transition-rule { , transition-rule }
| **if** expression **then** transition-rule { elseif-part } [else-part] **end**
elseif-part ::= **elseif** expression **then** transition-rule
else-part ::= **else** transition-rule
arguments ::= expression { , expression }

A regra de atualização é formada por um identificador, que deve ser um nome de função declarada como dinâmica, uma tupla, que são os argumentos para a determinação do endereço da atualização, e uma expressão, cujo valor será atribuído ao endereço formado pelo identificador e pela tupla. Na atualização, uma função dinâmica só pode receber funções definidas por meio de mapeamento explícito, isto é, sem corpo para ser avaliado. Funções estáticas e derivadas não podem ser atualizadas. Exemplos de atualizações são:

```
...
x,y : Int;
f,g : Int -¿ Int;
...
x := y, f(x) := 10, g := f
...
```

A regra bloco é composta por uma seqüência de regras, separadas por vírgula, que devem ser executadas simultaneamente.

Uma regra condicional tem o efeito de executar a regra, cuja guarda seja a primeira guarda a avaliar em verdadeiro, na seqüência dada. As expressões que formam as guardas devem ter o tipo `Bool`.

### 1.21.2 Regra *forall*

Uma regra *forall* tem a forma

**forall** $e_1, e_2, ..., e_k$ **do** `regra` **end**

onde os elementos $e_i$ são da forma `a:A`, onde `A` é um tipo *escalar*, ou `a:sa`, onde `sa` é um conjunto, ou ainda, da forma `a`, desde que o tipo de `a` possa ser deduzido do contexto. O seu efeito é criar diversas instâncias de `regra`, uma para cada valor possível de $e_1, e_2, ..., e_k$ nos domínios dados. Por exemplo, considere o trecho de código abaixo:

```
type Num = 1..3;
dynamic f(Int,Int):Int;
transition:
  forall x:Num,y:Num do
    f(x,y) := x + y
  end
end
```

Este código é equivalente ao código abaixo:

```
type Num = 1..3;
dynamic f(Int,Int):Int;
transition:
  f(1,1) := 2,
  f(1,2) := 3,
  f(1,3) := 4,
  f(2,1) := 3,
  f(2,2) := 4,
  f(2,3) := 5,
  f(3,1) := 4,
  f(3,2) := 5,
  f(3,3) := 6
end
```

É importante ressaltar que os identificadores definidos em um forall são meta-variáveis, que não podem, desta maneira, ser lados esquerdos de atualizações.

A sintaxe da regra *forall* é:

| | | |
|---|---|---|
| forall-rule | ::= | **forall** typed-pars-seq **do** transition-rule **end** |
| typed-pars-seq | ::= | element-seq { , element-seq } |
| id-seq | ::= | id \| id : type-expression \| id : set-expression |
| set-expression | ::= | expression |

### 1.21.3 Regra *choose*

A regra *choose* tem a forma:

**choose** $e_1, e_2, ..., e_k$ **satisfying** `g` **do** `regra` **end**

onde $e_1, e_2, ..., e_k$ são da mesma forma que na regra *forall*. O seu efeito é escolher aleatoriamente elementos dos domínios dados que satisfaçam a guarda `g` e executar a regra dada, associando os nomes de identificadores em $e_1, e_2, ..., e_k$ aos elementos escolhidos.

| | | |
|---|---|---|
| choose-rule | ::= | **choose** type-pars-seq [**satisfying** expression] **do** transition-rule **end** |
| typed-pars-seq | ::= | element-seq { , element-seq } |
| id-seq | ::= | id \| id : type-expression \| id : set-expression |

### 1.21.4 Regras Para Criação e Eliminação de Agentes

Para a criação de um agente, utilizamos a regra **create**, que tem a forma:

**create** $a_1, a_2, ..., a_n$ **do** `regra` **end**

onde $a_i$ é da forma `a:T`, ou `a`, ou `T`, onde `a` é um identificador e `T` é um tipo agente, da forma **agent of** `M`. O efeito desta regra é criar um agente que executará a regra do módulo `M`. Após a criação, este agente é referenciado pela variável `a`, e a regra de transição `regra` é executada, interpretando esta variável como o novo agente.

Se o nome do módulo não for explicitado, o compilador tentará inferi-lo pelo contexto; se não for possível, ocorrerá um erro de compilação.

Para a eliminação de um agente, utilizamos a regra **destroy**, que tem a forma:

**destroy** `id`

onde `id` é um nome de função ou variável, de um tipo agente. O seu efeito é encerrar a execução do agente referenciado por `id`.

Por exemplo, o código abaixo cria um agente `x` que executa a regra associada ao módulo `X`, executando as regras de inicialização dadas:

```
create x:X do
  numagentes := numagentes + 1,
  ready(x) := true
end
```

O código abaixo encerra a execução de todos os agentes que executam a regra do módulo `X` e encerra a sua própria execução.

```
forall x :  agent of X do
  destroy x
end,
destroy self
```

É interessante notar que **destroy** `self` é equivalente à regra **stop**.

A sintaxe para as regras **create** e **destroy** é dada por:

| | | |
|---|---|---|
| create-rule | ::= | **create** typed-pars-seq **do** transition-rule **end** |
| destroy-rule | ::= | **destroy** id |
| typed-pars-seq | ::= | element-seq { , element-seq } |
| id-seq | ::= | id \| id : type-expression |

### 1.21.5 Regra Para Parada

De acordo como o modelo ASM, a regra de transição de um agente é executada repetidamente. A regra para parada da máquina é a regra **stop**. Quando executada, esta regra faz com que a regra do módulo do agente não seja executada novamente, para que o agente encerre sua execução. Isto significa que, quando o **stop** for executado, o passo em que ele for executado é terminado, para, a partir deste momento, a execução da máquina realmente terminar.

Por exemplo, no estado final da execução abaixo, o valor de $a$ é 20:

```
module C
dynamic a := 0;
transition:
  if a < 10 then
    a := a + 10
  else
    a := a + 10,
    stop
  end
end
```

A sintaxe da regra **stop** é:

| | | |
|---|---|---|
| stop-rule | ::= | **stop** |

### 1.21.6 Regra *let*

Uma regra *let* é utilizada para fazer declarações que sejam locais a uma regra. Tem a seguinte sintaxe:

| | | |
|---|---|---|
| let-rule | ::= | **let** declaration-in-let { ; declaration-in-let } **in** transition-rule **end** |
| declaration-in-let | ::= | id = expression |

Por exemplo, o trecho de código

```
...
let
  a = head(L);
  b = head(tail(L));
  c = head(tail(tail(L)));
  x = a + b + c
in
  g(10) := x * (a + 1) * (b + c)
end
...
```

declara os identificadores `a`, `b`, `c` e `x`, cujo escopo é a regra

```
g(10) := x * (a + 1) * (b + c)
```

É importante ressaltar que os identificadores definidos em uma regra *let* são meta-variáveis que representam *right values*, não podendo ser, desta maneira, alvos de atribuições.

### 1.21.7 Regra *case*

Uma regra *case* é uma regra condicional, na qual todas as guardas são da forma

*expression* = *valor manifesto*

A sintaxe para esta regra é dada por:

| | | |
|---|---|---|
| case-rule | ::= | **case** expression **of** { case-rule-atom } [case-rule-otherwise] **end** |
| case-rule-atom | ::= | expression `->` transition-rule |
| case-rule-otherwise | ::= | **otherwise** `->` transition-rule |

Por exemplo, a regra abaixo é equivalente à regra mostrada acima:

```
case head(s) * head(tail(s)) of
  1 -> ...
  4 -> ...
  9 -> ...
  16 -> ...
  25 -> ...
  otherwise -> ...
end
```

### 1.21.8 Regra *with*

Uma regra *with* é uma forma de case, na qual, ao invés de compararmos o valor de uma expressão com expressões manifestas, comparamos o seu tipo com nomes de tipo.

A sintaxe para esta regra é a seguinte:

| | | |
|---|---|---|
| with-rule | ::= | **with** expression **as** { with-rule-atom } [with-rule-otherwise] **end** |
| with-rule-atom | ::= | [id :] type-expression `->` transition-rule |
| with-rule-otherwise | ::= | **otherwise** `->` transition-rule |

Por exemplo, a regra abaixo utiliza o tipo do primeiro elemento de uma lista para determinar a ação a ser tomada:

```
list :  List;
...
with head(list) as
  i:Int -> transition rules for integer head
  b:Bool -> transition rules for boolean head
  s:String -> transition rules for string head
  otherwise -> transition rules for other cases
end
```

### 1.21.9 Regra de Chamada a Ações

Uma regra de *chamada a ação* é equivalente a uma chamada de procedimentos em uma linguagem imperativa, com a exceção de que a sua execução gera um conjunto de atualizações que é tratado como o conjunto de atualizações de uma regra comum. Os parâmetros de uma ação, que forem atualizados, serão tratados como sinônimos para os argumentos no ponto de chamada. Demais argumentos são avaliados e substituídos pelos parâmetros formais da declaração da ação.

A utilização de ações permite escrevermos definições menores e mais legíveis, utilizando abstrações de regras.

A sintaxe de chamadas a ações é:

| | | |
|---|---|---|
| action-call-rule | ::= | id [(arguments)] |
| arguments | ::= | expression { , expression } |

Por exemplo, a regra abaixo é uma chamada a uma transição:

```
...
action t(x:Int) // declara a ação t
  ...
end
...
// Rules...
t(1), t(2), t(3) // chama a ação t...
...
```

## 1.22 Invariante da Execução

Em um módulo, pode-se definir uma condição que deve ser satisfeita toda vez que a sua regra de transição for executada. Chamamos esta condição de *invariante*. Para definirmos o invariante da regra de transição, definimos uma seção **invariant**, que contém a expressão booleana do invariante. A sua forma é

**invariant**: expressão-booleana

O invariante deve ser testado antes e depois de cada passo. Se em algum momento ele não for satisfeito, ocorre um erro de execução.

Em um módulo Machĭna, pode haver também uma seção de invariante, semelhante a um módulo qualquer. Entretanto, os elementos que podem aparecer na expressão do invariante são somente os literais booleanos, **true** e **false**, os operadores lógicos, **and**, **or**, **not** e **xor**, e invariantes de módulos, referenciados como `M.`**invariant**, onde `M` é um nome de módulo. Este invariante vale para a execução de todo os módulos criados a partir do módulo Machĭna, sendo igualmente testado antes e depois da execução de cada passo, independente de qual agente será executado.

## 1.23 Sintaxe de Expressões

As expressões de Machĭna são as seguintes: expressões sobre os tipos básicos, listas, conjuntos, tuplas, mapeamentos, conversão de tipos, condicionais, let e outras expressões.

### 1.23.1 Operadores de Machĭna

Os operadores de Machĭna são mostrados na tabela abaixo. A ordem de apresentação mostra a precedência dos operadores, sendo que os de precedência mais alta estão na primeira linha e os de precedência mais baixa estão no fim.

| | |
|---|---|
| `-` (unário) | precedência mais alta |
| `* / %` | |
| `+ -` | |
| `= != < > <= >=` | |
| **and** | |
| **or xor** | |
| **in** | |
| `::` | |
| `->` | precedência mais baixa |

### 1.23.2 Valores Booleanos

As expressões com o tipo `Bool` são:

- os literais **true** e **false**;
- qualquer aplicação de função (ver Seção 1.23.7), ou operação binária, como por exemplo, a aplicação dos operadores relacionais, cujo tipo de retorno seja equivalente ao tipo `Bool`;
- a operação unária de negação lógica, utilizando o operador pré-fixado **not**;
- as operações binárias que utilizam os operadores lógicos **and** (conjunção), **or** (disjunção) e **xor** (**or** exclusivo).

Exemplos de expressões booleanas são: `f(x) = 21`, `y * a > 5` **and** `cond`, `b` **and** `c` **xor** `d` **or** `e`.

### 1.23.3 Caracteres

As expressões com o tipo `Char` são:

- os literais caracteres, que representam os caracteres da tabela ASCII;
- qualquer aplicação de função (ver Seção 1.23.7), cujo tipo de retorno seja equivalente ao tipo `Char`;
- as funções pré-definidas sobre o tipo `Char`, dadas abaixo:

  `ord(Char):Int` – Retorna o código ASCII do caractere;

  `chr(Int):Char` – Retorna o caractere com o código ASCII dado.

Exemplos de expressões com o tipo `Char` são: `'a'`, `'1'`, `'\n'`, `chr(13)`, `ord('\n')`.

### 1.23.4 Números Inteiros

As expressões com o tipo `Int` são:

- os literais inteiros, que representam os números inteiros, com valor mínimo -2147483648 e máximo 2147483647. Estes literais podem ser representados na forma decimal (seqüência de um ou mais dígitos de 0 a 9), octal (seqüência de 1 ou mais dígitos de 0 a 7, precedida sempre por um 0) ou hexadecimal (seqüência de 1 ou mais dígitos de 0 a 9 ou letras de A a Z, precedida sempre por 0x);
- a operação unária de negação, utilizando o operador `-`;
- as operações binárias de adição (`+`), subtração (`-`), multiplicação (`*`), divisão (`/`) e resto da divisão (`%`);

- qualquer aplicação de função (ver Seção 1.23.7), cujo tipo de retorno seja equivalente ao tipo `Int`;
- as funções pré-definidas sobre o tipo `Int`, dadas abaixo:

  `abs(Int):Int` – valor absoluto de um número inteiro;

  `max(Int,Int):Int` – máximo entre dois inteiros;

  `min(Int,Int):Int` – mínimo entre dois inteiros;

  `sqr(Int):Int` – quadrado de um número inteiro.

Exemplos de expressão com números inteiros são: `x + 1`, `x * 2 + y * 5 - 10 % v`, `abs(x) + max(a,b) - sqr(min(x,y) + 1)`.

### 1.23.5 Números Reais

As expressões com o tipo `Real` são:

- os literais reais, que representam números de ponto flutuante do padrão IEEE 754, com ± 1.7976931348623157OE+308 como valor máximo e ±4.94065645841246544E-324 como valor mínimo;
- a operação unária de negação, utilizando o operador `-`;
- as operações binárias de adição (`+`), subtração (`-`), multiplicação (`*`) e divisão (`/`);
- qualquer aplicação de função (ver Seção 1.23.7), cujo tipo de retorno seja equivalente ao tipo `Real`;
- as funções pré-definidas sobre o tipo `Real`, dadas abaixo:

  `abs(Real):Real` – valor absoluto de um número real;

  `max(Real,Real):Real` – máximo entre dois números reais;

  `min(Real,Real):Real` – mínimo entre dois números reais;

  `sqr(Real):Real` – quadrado de um número real;

  `sqrt(Real):Real` – raiz quadrada de um número real.

### 1.23.6 Cadeias de Caracteres

As expressões sobre o tipo `String` são:

- os literais cadeias de caracteres, que são seqüências de zero ou mais caracteres, envolvidas por aspas; o padrão de cadeias de caracteres de Machĭna é de cadeias de caracteres terminadas com nulo;
- a operação binária de concatenação, utilizando o operador `+`;
- qualquer aplicação de função (ver Seção 1.23.7), cujo tipo de retorno seja equivalente ao tipo `String`;
- a função pré-definida `length(String):Int`, que retorna o comprimento da cadeia de caracteres.

Exemplos de expressões com cadeias de caracteres são: `"string\n"`, `length(‘‘abcd"+ nome)`.

### 1.23.7 Aplicação de Funções

A aplicação de função tem a forma geral

*nome de função*(*argumentos*)

onde *argumentos* é uma lista de expressões, cujo comprimento é igual à aridade da função da aplicação. Se a aridade da função for igual a zero, então não se utilizam os parênteses, somente o nome da função. Os tipos dos argumentos devem ser equivalentes aos tipos esperados como parâmetros.

Exemplos de aplicações de função: `x`, `f(1,2,true)`, `f(x,g(y+1,h))`, `chr(0)`.

### 1.23.8 Listas

Machĩna possui dois tipos de listas: listas genéricas e listas de elementos de um tipo específico. Uma lista vazia é dada pelo literal **nil**. Sobre listas, estão definidas as seguintes operações:

| | |
|---|---|
| `x::list` | retorna uma lista cuja cabeça é `x` e cuja cauda é `list` |
| `concat(list1, list2)` | retorna a concatenação das listas `list1` e `list2` |
| `head(list)` | retorna o elemento na cabeça de `list` |
| `tail(list)` | retorna a cauda de `list` |
| `lenght(list)` | retorna o comprimento de `list` |

Em tempo de compilação, são feitos testes para consistência e inferência de tipos. As regras para consistência de tipos são as seguintes:

1. na operação `x::list`, se `x` possuir o tipo `X`, então `list` deve ser do tipo **list of** `X` ou `List`;
2. na operação `cat(list1, list2)`, `list1` e `list2` devem ser listas do mesmo tipo, a menos que uma delas seja do tipo `List`; se o tipo de uma delas for `List`, então o resultado é do tipo `List`;
3. se `list` for do tipo **list of** `X`, então `head(list)` é do tipo `X`; se `list` for do tipo `List`, então `head(list)` será do tipo "?";
4. a expressão `tail(list)` tem o mesmo tipo de `list`.

Um agregado lista é da forma

$$[t_1, t_2, ..., t_n].$$

Se todos as expressões forem do mesmo tipo X, então o tipo da expressão toda é **list of** X; caso contrário, o tipo será `List`.

### 1.23.9 Conjuntos

Assim como listas, conjuntos podem ser genéricos ou de um tipo específico. As operações envolvendo conjuntos são:

| | |
|---|---|
| `s1 + s2` | união |
| `s1 - s2` | diferença |
| `s1 * s2` | interseção |
| `x in s` | pertinência |
| `s(x)` | pertinência |
| `s1 relop s2` | inclusão (`<=`), igualdade (`=`), etc. |
| `card(s)` | cardinalidade do conjunto |

Um agregado conjunto é da forma

$$\{t_1, t_2, ..., t_n\}.$$

Se todos as expressões forem do mesmo tipo X, então o tipo da expressão toda é **set of** X; caso contrário, o tipo será `Set`.

Conjuntos podem ser utilizados em regras da seguinte maneira:

- Regra de atualização:

  `s(t) :=` **true** $\equiv$ `s := s + { t }`

  e

  `s(t) :=` **false** $\equiv$ `s := s - { t }`

- Regra *forall*: se `s` é do tipo **set of** `X`, então a construção

  **forall** `x:s` **do** `regra` **end**

  instancia `regra` substituindo `x` com cada valor pertencente a `s`, executando todas as instâncias em paralelo;

- Regra *choose*: se `s` é do tipo **set of** `X`, então a construção

  **choose** `x:s` **do** `regra` **end**

  escolhe aleatoriamente um elemento $e$ pertencente ao conjunto `s` e executa `regra`, associando `x` a $e$.

### 1.23.10 Tuplas

Uma tupla de Machina é uma seqüência de tamanho fixo, cujos elementos podem ser de quaisquer tipos.

Uma expressão da forma `T(t1, t2, t3)` cria uma tupla do tipo `T`, formada pelos elementos `t1`, `t2` e `t3`. O acesso aos elementos de uma tupla é feito sempre pelo seus rótulos, como no exemplo abaixo:

```
...
type Pair is (x,y:Int);
p1, p2 :  Pair;
...
p1 := Pair(1,2),
p2.x := p1.y
...
```

### 1.23.11 Mapeamentos

Um mapeamento é um conjunto de associações de valores de um domínio a valores de um contradomínio. Por exemplo, considere o fragmento de código abaixo:

**static** `f := {1 -> 2, 2 -> 3, 3 -> 4}`

`f` é uma função de inteiros para inteiros, tal que:

`f(1) = 2, f(2) = 3, f(3) = 4`

O valor da aplicação de `f` a algum outro valor, que não especificado no conjunto, é o valor **default** do contradomínio.

O operador `->` cria um par que pode ser inserido em algum conjunto. Um mapeamento é semelhante a um conjunto de pares, isto é, uma relação, com a diferença que, se $x$ é um conjunto de pares e $y$ é um mapeamento, então é possível que, para quaisquer $a$, $b$ e $c$, $(a, b) \in x$ e $(a, c) \in x$. Entretanto, $(a, b) \in y$ e $(a, c) \in y$ só é possível se $b = c$.

As operações que podem ser realizadas sobre mapeamentos são a aplicação de função, da forma $f(a_1, ..., a_n)$, a atualização de funções dinâmicas, da forma $f(a_1, ..., a_n) := b$ e a comparação por `=` (igualdade) e `!=` (diferença).

### 1.23.12 Conversão de Tipos

As expressões abaixo realizam a conversão de tipos, para os casos em que é necessário, como por exemplo, utilização de valores do tipo `?`. Os dois tipos de expressões que têm essa finalidade são as expressões **with** e **is** e a conversão explícita de tipos.

A expressão **with** tem a forma:

**with** $e$ **as**
  $e_1 : T_1$ `->` $b_1$
  $e_2 : T_2$ `->` $b_2$
  ...
  $e_n : T_n$ `->` $b_n$
  **otherwise** `->` $b_{n+1}$
**end**

Se as expressões $b_1, b_2, ..., b_n, b_{n+1}$ tiverem o mesmo tipo `X`, então o tipo da expressão é `X`. Caso contrário é `?`. A cláusula **otherwise** é obrigatória, a menos que as cláusulas de coerção sejam exaustivas. Assim, se o tipo de $e$ for `?`, esta cláusula é obrigatória. O valor desta expressão é obtido, comparando-se o tipo de $e$ com os tipos das cláusulas **as**. Se a $i$-ésima cláusula casar com o tipo de $e$, então, associamos $e_i$ ao valor de $e$ e retornamos o valor de $b_i$. Se nenhuma cláusula casar com o tipo de $e$, então o valor retornado é o valor de $b_{n+1}$.

Por exemplo, considere o fragmento de código:

```
...
dynamic x :  List; // Os elementos de x podem ser de qualquer tipo
...
with head(x) as
  a:Int -> 1 + a
  b:Bool -> if b then 1 else 0 end
  c:Real -> round(c)
  otherwise -> 4
end
...
```

No exemplo abaixo, a cláusula **otherwise** não é necessária, pois as cláusulas **as** são exaustivas.

```
...
type X is Bool | Int | Real;
dynamic x :  X;
...
with x as
  a:Int -> 1
  b:Bool -> 2
  c:Real -> 3
end
```

```
...
```

O operador binário **is** é um operador de inspeção de tipos, que recebe como argumentos uma expressão e um nome de tipo e retorna verdadeiro se o tipo da expressão for igual ao tipo dado. Um exemplo da utilização do operador **is** está no código abaixo:

```
...
type X is Bool | Int | Real;
dynamic x :  X;
...
if x is Int then
  ...
elseif x is Bool then
  ...
else
  ...
end
...
```

O *casting* explícito de tipos é uma expressão da forma $(T)e$. O seu efeito é fazer a conversão explícita de $e$ para um valor do tipo $T$. Se a conversão não puder ser feita, ocorrerá um erro de execução.

### 1.23.13 Condicionais e Let

Nesta categoria se enquadram as expresssões condicionais **if** e **case** e a expressão **let**. A estrutura destas expressões é equivalente às regras de transição de mesmos nomes, com algumas diferenças:

- a parte **else** da expressão **if** é obrigatória;
- a cláusula **otherwise** da expressão **case** é obrigatória, a menos que as cláusulas sejam exaustivas, como por exemplo, se cobrirem todos os elementos de uma enumeração;
- se os tipos das expressões resultantes de cada parte do **if** e do **case** são o mesmo tipo $X$, então o tipo da expressão toda é $X$; caso contrário é `?`.

### 1.23.14 Expressões Especiais

A expressão **default** retorna o valor *default* do tipo esperado. Para cada tipo existe um valor *default* pré-definido. A expressão **self** retorna a identificação ao agente que estiver executando a regra.

Exemplos:

```
algebra:
  static
    a :  Int := default // x := 0
    b :  Bool := default // x := false
    c :  Real := default // x := 0.0
    d :  Set := default // x := {}
    e :  list of Int := default // x := nil
    f := default // f := undef, pois undef é o default de ?
transition:
  ...
  hungry(self) := true, // Atualiza as funções locais ao agente
```

```
    thinking(self) := false
  ...
```

### 1.23.15 Gramática das Expressões

A sintaxe das expressões é:

| | | |
|---|---|---|
| expression | ::= | literal |
| | \| | unop expression |
| | \| | expression binop expression |
| | \| | mod-function-call |
| | \| | agregate |
| | \| | if-expression |
| | \| | case-expression |
| | \| | let-expression |
| | \| | with-expression |
| | \| | expression . expression |
| | \| | [module-name .] id [(arguments)] |
| | \| | **self** |
| | \| | **default** |
| | \| | (expression) |
| literal | ::= | **false** \| **true** \| char-literal \| integer-literal |
| | \| | float-literal \| string-literal \| **nil** |
| unop | ::= | − \| **not** \| (type-name) |
| binop | ::= | ∗ \| / \| % \| + \| − \| = \| ! = \| < \| > \| <= \| >= |
| | \| | **and** \| **or** \| **xor** \| :: \| **in** \| **is** |
| agregate | ::= | "["expression { , expression }"]" |
| | \| | "{"expression { , expression } "}" |
| | \| | type-name ( expression { , expression } ) |
| if-expression | ::= | **if** expression **then** expression { if-expr-elseif } if-expr-else **end** |
| if-expr-elseif | ::= | **elseif** expression **then** expression |
| if-expr-else | ::= | **else** expression |
| let-expression | ::= | **let** declaration-in-let { ; declaration-in-let } **in** expression **end** |
| case-expression | ::= | **case** expression **of** { case-expr-atom } [case-expr-default] **end** |
| case-expr-atom | ::= | expression → expression |
| case-expr-default | ::= | **otherwise** → expression |
| with-expression | ::= | **with** term { with-expr-atom } [with-expr-default] **end** |
| with-expr-atom | ::= | [id :] type-name → expression |
| with-expr-default | ::= | **otherwise** → expression |

## 1.24 Manipulação de Arquivos

Para manipulação de arquivos, existem os tipos pré-definidos `InputStream`, `OutputStream` e o construtor de tipos **file of**, que recebe como argumento um tipo.

O tipo pré-definido `InputStream` representa um arquivo texto para leitura com caracteres de leiaute. As operações (ações) que podem ser feitas sobre este tipo são:

`open(InputStream input, String s)` – ação pré-definida que abre, para leitura, o arquivo de nome `s`, colocando uma referência para este arquivo no parâmetro `input`;

`close(InputStream input)` – ação pré-definida, que fecha o arquivo referenciado por `input`;

`readInt(InputStream input, Int i)` – lê um valor inteiro do arquivo `input` e coloca o resultado em `i`;

`readChar(InputStream input, Char c)` – lê um caractere do arquivo `input` e coloca o resultado em `c`;

`readReal(InputStream input, Real r)` – lê um número ponto-flutuante do arquivo `input` e coloca o resultado em `r`;

`readString(InputStream input, String s)` – lê uma cadeia de caracteres do arquivo `input` e coloca o resultado em `s`.

O tipo pré-definido `OutputStream` é análogo ao tipo `OutputStream`, e representa um arquivo texto, para escrita, com caracteres de leiaute. As operações (ações) que podem ser feitas sobre este tipo são:

`open(OutputStream output, String s)` – ação pré-definida que abre, para escrita, o arquivo de nome `s`, colocando uma referência para este arquivo no parâmetro `output`;

`close(OutputStream output)` – ação pré-definida, que fecha o arquivo referenciado por `output`;

`writeInt(OutputStream output, Int i)` – escreve o valor de `i` no arquivo `output`;

`writeChar(OutputStream output, Char c)` – escreve o valor de `c` no arquivo `output`;

`writeReal(OutputStream output, Real r)` – escreve o valor de `r` no arquivo `output`;

`writeString(OutputStream output, String s)` – escreve o valor de `s` no arquivo `output`;

Uma expressão de tipo **fileof**$T$, onde $T$ é um tipo, cria um tipo que representa arquivos, cujos elementos são do tipo $T$. Estes arquivos são *binários*, com acesso *seqüencial* ou *direto*. As operações (ações) que podem ser feitas são:

`open(`**file of** `T tFile, String s, Bool readOnly)` – ação pré-definida que abre, para leitura ou para escrita, o arquivo de nome `s`, colocando uma referência para este arquivo no parâmetro `tFile`; o arquivo será de leitura de o parâmetro `readOnly` for verdadeiro, caso contrário, será de escrita;

`close(`**file of** `T tFile)` – ação pré-definida, que fecha o arquivo referenciado por `tFile`;

`read(`**file of** `T tFile, T t)` – lê um valor do tipo `T`, do arquivo referenciado por `tFile`, e coloca este valor em `t`;

`write(`**file of** `T tFile, T t)` – escreve o valor de tipo `t`, no arquivo referenciado por `tFile`;

Sobre todos os arquivos são definidas as funções `eof` e `status`. A função `eof(f:File):Bool` recebe como argumento um arquivo de qualquer tipo e retorna **true**, se o arquivo estiver no fim, ou **false**, caso contrário. A função `status(f:File):Int` retorna o *status* do arquivo como um número inteiro, de acordo com a tabela abaixo:

| | |
|---|---|
| 0 | arquivo fechado |
| 1 | arquivo aberto |
| 2 | erro de leitura no arquivo |
| 3 | erro de escrita no arquivo |

A menos da função `status`, qualquer operação sobre um arquivo fechado gera um erro de execução.

É importante salientar que, por causa do modelo algébrico, qualquer operação de leitura em arquivos só é percebida no próximo estado. Da mesma forma, qualquer chamada a `status` em um mesmo estado retorna o mesmo valor, que é o *status* do arquivo no passo anterior. Assim como qualquer ação de Machĩna, a ordem de execução de duas operações sobre arquivos, no mesmo, estado não é garantida. Por exemplo, considere o código abaixo:

```
...
f : InputStream;
g : OutputStream;
transition:
    readInt(f,a),
    readInt(f,a),
    readInt(f,b),
    writeInt(g,x),
    writeInt(g,y),
end
```

Serão lidos três valores do arquivo `f` e escrito dois valores no arquivo `g`. Entretanto, não há como determinar qual será o valor de `a` e qual será o valor de `b`, pois isto dependerá da ordem em que o compilador escolher executar este código. Da mesma forma, não é possível determinar qual valor será escrito antes em `g`, se o valor de `x` ou se o valor de `y`.

# Capítulo 2

# Exemplos

## 2.1 Pesquisa Binária

Os módulos abaixo especificam um agente que realiza a pesquisa binária.

```
module PesqBin
algebra:
  static n :  Int := 100;
  type
    Index is 1..n;
    Array is Index -> Int;
  external
    key :  Int;
    array;
  dynamic
    k, pos, inf, sup :  Int;
    encontrado :  Bool;
    array1;
  init
    array1 := array
    k := key, pos := (n+1)/2, inf := 1, sup := n
    encontrado := false
transition:
  let mean = (inf+sup)/2
    in
    if not encontrado and inf < sup then
      if array1(mean) = k then encontrado := true, pos := mean
      elseif array1(mean) < k then inf := mean + 1
      else sup := mean - 1
      end
    end
  end
end

machina PesquisaBinaria
  create PesqBin
end
```

## 2.2 Ordenação por Seleção

O módulo abaixo especifica a ordenação por seleção.

```
module Selecao
algebra:
  static n :  Int := 100;
  type
    Index is 1..n;
    Array1 is Index -> Int;
  external
    n :  Int;
    array;
  dynamic
    modo, i, k, j :  Int;
    array1;
  init
    array1 := array,
    modo := 1, i := 1, k := 1, j := 1
transition:
  if modo = 1 and i < n then
    k := i, j := i + 1, modo := 2
  elseif modo = 2 then
    j := j + 1,
    if j > n then
      modo := 3
    elseif f(j) < f(k) then
      k := j
    end
  else modo = 3 then
    i := i + 1, modo := 1,
    if k != i then
      f(k) := f(i), f(i) := f(k)
    end
  end
end

machina OrdenacaoPorSelecao
  create ops :  Selecao
end
```

## 2.3 Números Primos

O programa abaixo especifica um algoritmo para busca de números primos.

```
machina MarcaPrimos
algebra:
  type
    Num is 2..1000;
  dynamic
    primo :  Num -> Bool;
  init
    forall n:Num do primo(n) := true end
transition:
```

```
  forall num1,num2 do
    if num2 < num1 and num1%num2 = 0 then
      primo(num1) := false
    end
  end
end
```

O programa abaixo é equivalente ao anterior, mas utiliza outros recursos de Machĭna:

```
machina MarcaPrimos
algebra:
  type
    Num is 2..1000;
    Numset is set of Num default {2..1000};
  dynamic
    primo :  Numset;
transition:
  forall num1,num2 do
    if num2 < num1 and num1%num2 = 0 then
      primo(num1) := false
    end
  end
end

machina DeterminaPrimos
  create mp :  MarcaPrimos
end
```

## 2.4 Especificação da Semântica de *Tiny*

*Tiny* é uma linguagem imperativa de pequeno porte que contém somente comandos e expressões. A definição de *Tiny* será feita em cinco módulos:

1. Módulo principal (`Tiny`) – contém a regra principal da especificação e as rotinas de inicialização;
2. Módulo de Expressões (`Expressions`) – contém as regras de avaliação de expressões;
3. Módulo de Comandos (`Commands`) – contém as regras de execução de comandos;
4. Módulo de Operações (`Operations`) – contém as regras de operação na pilha.
5. Módulo de Globais (`Globals`) – contém as declarações dos principais tipos e funções da especificação de *Tiny*.

### 2.4.1 Módulo de Globais (`Globals`)

```
module Globals
algebra:
  type
    public Undefined is public enum {undefined};
    public Id is String;
    public Program is List;
    public Input is list of Int;
```

```
    public Output is list of Int;
    public Value is Bool | Int;
    public Memory is Id → (Value | Undefined) default undefined;
    public KeyWord is
      {tadd, tassign, tcond, texchange, tequals tnot, toutput, tread, twhile};
    public Opstack is List;
  external
    p:Program;
    i:Input;
  dynamic
    public program := p, input := i, output := nil, opstack := nil;
    public memory;
    public error := false
end
```

### 2.4.2 Módulo de Operações (Operations)

```
module Operations
import:
  Globals(program,output,input,opstack,error,memory);
algebra:
  action
    treatOutput :=
      if head(program) is x:Int then
        output := concat(output, [x]),
        program := tail(program),
        opstack := tail(opstack)
      else
        error := true
      end
    end;
    treatAssign
      opstack := tail(tail(opstack)),
      memory((String)head(tail(opstack))) := (Globals.Value)head(program)
    end;
    treatExchange :=
      opstack := tail(opstack),
      program := head(tail(program))::head(program)::tail(tail(program))
    end;
    treatAdd :=
      let
        hp = head(program);
        htp = head(tail(program));
      in
        if hp is Int and htp is Int then
          program := ((Int)x + (Int)y)::tail(tail(program)),
          opstack := tail(opstack)
        else
          error := true
        end
      end
    end;
    treatNot :=
```

```
      if head(program) is Bool then
        program := (not ((Bool)b))::tail(program),
        opstack := tail(opstack)
      else
        error := true
      end
    end;
    treatCond :=
      if head(program) is Bool then
        if (Bool) head(program) then
          program := head(tail(program))::tail(tail(tail(program)))
        else
          program := tail(tail(program))
        end,
        opstack := tail(opstack)
      else
        error := true
      end
    end;
    treatEquals :=
      let
        hp = head(program);
        htp = head(tail(program));
      in
        if hp is Int and htp is Int then
          program := ((Int)x = (Int)y)::tail(tail(program)),
          opstack := tail(opstack)
        else
          error := true
        end
      end
    end;
    public treatValue :=
      if head(opstack) is Globals.KeyWord then
        case (Globals.KeyWord) head(opstack) of
          Globals.toutput -> treatOutput;
          Globals.tassign -> treatAssign;
          Globals.texchange -> treatExchange;
          Globals.tadd -> treatAdd;
          Globals.tnot -> treatNot;
          Globals.tcond -> treatCond;
          Globals.equals -> treatEquals
        end
      else
        error := true
      end
    end
end
```

### 2.4.3 Módulo de Expressões (Expressions)

```
module Expressions
import:
```

```
  Globals(program,output,input,opstack,error,memory);
algebra:
  action
    public readMemory :=
      let
        id = (String) head(program)
      in
        if memory(id) = undef then
          error := true
        else
          program := memory(s) ::  tail(program)
        end
      end
    end;
    public treatNot :=
      program := tail(program),
      opstack := Globals.tnot::opstack
    end;
    public treatEquals :=
      program := tail(program),
      opstack := Globals.tequals::opstack
    end;
    public treatAdd :=
      program := tail(program),
      opstack := Globals.texchange::Globals.tadd::opstack
    end;
    public treatRead :=
      if input = nil then
        error := true
      else
        program := head(input)::tail(program),
        input := tail(input)
      end
    end
  end
```

### 2.4.4 Módulo de Comandos (Comandos)

```
module Commands
import:
  Globals(program,output,input,opstack,error,memory);
algebra:
  action
    public treatAssign :=
      program := tail(tail(program)),
      opstack := Globals.tassign::head(tail(program))::opstack
    end;
    public treatOutput :=
      program := tail(program),
      opstack := Globals.toutput::opstack
    end;
    public treatConditional :=
      program := tail(program),
```

```
      opstack := Globals.tcond::opstack
    end;
    public treatWhile :=
      let
        second = head(tail(program));
        third = head(tail(tail(program)));
        whileExp = second;
        whileCom = Globals.twhile::second::third::nil;
        whileTrue = third::whileCom::nil;
        whileCont = tail(tail(tail(program)));
      in
        program := whileExp::whileTrue::nil::whileCont,
        opstack := Globals.tcond::opstack
      end
    end
  end
```

### 2.4.5 Módulo principal (Tiny)

```
module Tiny
import:
  Commands, Expressions, Operations,
  Globals(program,output,input,opstack,error,memory);
algebra:
  actionflatProgram :=
    program := concat((List) head(program), tail(program))
  end
transition:
  if program != nil and not error then
    with head(program) as
      l:List -> flatProgram;
      s:String -> Expressions.readMemory;
      p:Globals.KeyWord ->
        case p of
          Globals.tnot -> Expressions.treatNot;
          Globals.tequals -> Expressions.treatEquals;
          Globals.tadd -> Expressions.treatAdd;
          Globals.tread -> Expressions.treatRead;
          Globals.tassign -> Commands.treatAssign;
          Globals.toutput -> Commands.treatOutput;
          Globals.tcond -> Commands.treatConditional;
          Globals.twhile -> Commands.treatWhile;
        end;
      x:Globals.Value -> Operations.treatValue
    end
  end
end
```

### 2.4.6 Disparo do Agente de Execução

```
machina TinySemantics
  create tny :  Tiny
end
```

## 2.5 Jantar dos Filósofos

Este é um exemplo de especificação com vários agentes, onde todos executam o mesmo programa, de maneira concorrente. Para explicitar a concorrência, definiremos a regra de transição como intercalada (*interleaved*).

```
module DinPh
algebra:
  static n :  Int := 5;
  type PhId is 1..n;
  dynamic
    initialized, thinking, hungry, eating :  DinPh -> Bool;
    leftFork, rightFork :  PhId -> Bool;
    myId :  DinPh -> PhId;
transition:
  if not inicializado(self) then
    thinking(self) := true,
    initialized(self) := true
  else
    if thinking(self) then
      thinking(self) := false, hungry(self) := true
    elseif hungry(self) then
      if not leftFork(myId(self)) and not rightFork(myId(self)) then
        eating(self) := true, hungry(self) := false,
        leftFork(myId(self)) := true,
        rightFork(myId(self)) := true
      end
    else
      eating(self) := false, hungry(self) := true
    end
  end
end
```

O módulo Machina que dispara os agentes é dado abaixo:

```
machina DiningPhilosophers
  create5 DinPh
end
```