Compiladores/Projecto de Compiladores/Projecto 2022-2023/Manual de Referência da Linguagem MML

*From Wiki**3*

< Compiladores | Projecto de Compiladores

AVISOS - Avaliação em Época Normal	**[Expand]**

Material de Uso Obrigatório	**[Expand]**

Este manual apresenta características básicas da linguagem MML (tipos de dados, manipulação de nomes); convenções lexicais; estrutura/sintaxe; especificação das funções; semântica das instruções; semântica das expressões; e, finalmente, alguns exemplos.

Tipos de Dados

A linguagem é fracamente tipificada (são efectuadas algumas conversões implícitas). Existem 4 tipos de dados básicos, todos compatíveis com a linguagem C, e com alinhamento em memória sempre a 32 bits:

- Tipos numéricos: os inteiros, em complemento para 2, ocupam 4 bytes; os reais, em vírgula flutuante, ocupam 8 bytes (IEEE 754).
- As cadeias de caracteres são vectores de caracteres terminados por ASCII NUL (carácter com o valor zero). Variáveis e literais deste tipo só podem ser utilizados em atribuições, impressões, ou como argumentos/retornos de funções. Os caracteres são valores de 8 bits não directamente manipuláveis.
- Os ponteiros representam endereços de objectos e ocupam 4 bytes. Podem ser objecto de operações aritméticas (deslocamentos) e permitem aceder ao valor apontado.

Os tipos suportados por cada operador e a operação a realizar são indicados na definição das expressões.

Existem ainda tipos associado a valores funcionais, i.e., tipos que descrevem a interface de funções (ver abaixo). Os valores em memória associados a estes tipos são efectivamente ponteiros, mas para funções e não para dados, podendo ser usados para invocar as funções correspondentes. Estes ponteiros não aceitam operações de aritmética de ponteiros ou de indexação (embora ponteiros para estes ponteiros as aceitem).

Manipulação de Nomes

Os nomes (identificadores) correspondem exclusivamente a variáveis. As funções são referenciadas através de ponteiros nomeados. Nos pontos que se seguem, usa-se o termo entidade para as designar indiscriminadamente, explicitando-se quando a descrição for válida apenas para um dos casos. Existem ainda nomes para funções externas: neste caso, os nomes referem directamente a posição do código dessas funções (à la C; ver **foreign** abaixo).

Espaço de nomes e visibilidade dos identificadores

O espaço de nomes global é único, pelo que um nome utilizado para designar uma entidade num dado contexto não pode ser utilizado para designar outras (ainda que de natureza diferente).

Os identificadores são visíveis desde a declaração até ao fim do alcance: ficheiro (globais) ou bloco (locais). A reutilização de identificadores em contextos inferiores encobre declarações em contextos superiores: redeclarações locais podem encobrir as globais até ao fim de um bloco. É possível utilizar símbolos globais nos contextos dos blocos das funções, mas não é possível declará-los (ver símbolos globais).

Validade das variáveis

Contents

- 1 Tipos de Dados
- 2 Manipulação de Nomes
  - 2.1 Espaço de nomes e visibilidade dos identificadores
  - 2.2 Validade das variáveis
- 3 Convenções Lexicais
  - 3.1 Caracteres brancos
  - 3.2 Comentários
  - 3.3 Palavras-chave
  - 3.4 Tipos
  - 3.5 Operadores de expressões
  - 3.6 Delimitadores e terminadores
  - 3.7 Identificadores (nomes)
  - 3.8 Literais
    - 3.8.1 Inteiros
    - 3.8.2 Reais em vírgula flutuante
    - 3.8.3 Cadeias de caracteres
    - 3.8.4 Ponteiros
- 4 Gramática
  - 4.1 Tipos, identificadores, literais e definição de expressões
  - 4.2 Left-values
  - 4.3 Ficheiros
  - 4.4 Declaração de variáveis
  - 4.5 Símbolos globais
  - 4.6 Inicialização
- 5 Funções
  - 5.1 Declaração
  - 5.2 Invocação
  - 5.3 Corpo
  - 5.4 Função principal e execução de programas
- 6 Instruções
  - 6.1 Blocos
  - 6.2 Instrução condicional
  - 6.3 Instrução de iteração
  - 6.4 Instrução de terminação
  - 6.5 Instrução de continuação
  - 6.6 Instrução de retorno
  - 6.7 Expressões como instruções
  - 6.8 Instruções de impressão
- 7 Expressões
  - 7.1 Expressões primitivas
    - 7.1.1 Identificadores
    - 7.1.2 Leitura
    - 7.1.3 Parênteses curvos
    - 7.1.4 Funções
  - 7.2 Expressões resultantes de avaliação de operadores
    - 7.2.1 Indexação de ponteiros
    - 7.2.2 Identidade e simétrico

- 7.2.3 Reserva de memória
- 7.2.4 Expressão de indicação de posição
- 7.2.5 Expressão de dimensão
- 8 Exemplos e Testes
- 9 Omissões e Erros

As entidades globais (declaradas fora de qualquer função), existem durante toda a execução do programa. As variáveis locais a uma função existem apenas durante a sua execução. Os argumentos formais são válidos enquanto a função está activa.

Convenções Lexicais

Para cada grupo de elementos lexicais (tokens), considera-se a maior sequência de caracteres constituindo um elemento válido. Assim, por exemplo, a designação **>=** é sempre um único elemento lexical (por oposição à situação ilegal de se terem dois símbolos: **>** seguido de **=**).

Caracteres brancos

São considerados separadores e não representam nenhum elemento lexical: **mudança de linha** ASCII LF (**0x0A**, **\n**), **recuo do carreto** ASCII CR (**0x0D**, **\r**), **espaço** ASCII SP (**0x20**,) e **tabulação horizontal** ASCII HT (**0x09**, **\t**).

Comentários

Existem dois tipos de comentários, que também funcionam como elementos separadores:

- **explicativos** -- começam com **//** e acabam no fim da linha; e
- **operacionais** -- começam com **/*** e terminam com ***/**, podendo estar aninhados.

Se as sequências de início fizerem parte de uma cadeia de caracteres, não iniciam um comentário (ver definição das cadeias de caracteres).

Palavras-chave

As seguintes palavras-chave são reservadas, não constituindo identificadores (devem ser escritas exactamente como indicado):

- tipos: **int double string void**
- declarações: **foreign forward public auto**
- instruções: **if elif else while stop next return**
- expressões: **input null sizeof**
- outras: **begin end**

Tipos

Os seguintes elementos lexicais designam tipos em declarações (ver gramática): **int** (inteiro), **double** (real), **string** (cadeia de caracteres).

Os tipos correspondentes a ponteiros são outros tipos delimitados por **[** e **]**, designando uma indirecção e não o objecto directo (ver gramática).

Operadores de expressões

São considerados operadores os elementos lexicais apresentados na definição das expressões.

Delimitadores e terminadores

Os seguintes elementos lexicais são delimitadores/terminadores:

- **,** (vírgula)
- **;** (ponto e vírgula)
- **!** e **!!** (operações de impressão)
- **(** e **)** (delimitadores de expressões)
- **{** e **}** (delimitadores de blocos)

Identificadores (nomes)

São iniciados por uma letra, seguindo-se 0 (zero) ou mais letras ou dígitos. O comprimento do nome é ilimitado e dois nomes são distintos se houver alteração de maiúscula para minúscula, ou vice-versa, de pelo menos um carácter.

Literais

São notações para valores constantes de alguns tipos da linguagem (não confundir com constantes, i.e., identificadores que, em algumas linguagens, designam elementos cujo valor não pode ser alterado durante a execução do programa).

Inteiros

Um literal inteiro é um número não negativo. Números negativos podem ser construídos pela aplicação do operador de negação unária (**-**) a um literal (sempre positivo).

Literais inteiros decimais são constituídos por sequências de 1 (um) ou mais dígitos de **0** a **9**, em que o primeiro digito não é 0 (zero), excepto no caso do número 0 (zero). Neste caso, é composto apenas pelo dígito 0 (zero) (em qualquer base).

Literais inteiros em base 8 começam sempre pelo dígito 0 (zero), sendo seguidos de um ou mais dígitos de **0** a **7** (note-se que **09** é um literal inválido em base 8). Exemplo: **006**.

Se não for possível representar um literal Inteiro na máquina, devido a overflow, deverá ser gerado um erro lexical.

Reais em vírgula flutuante

Os literais reais (sempre positivos) são expressos tal como em C (exclusivamente em base 10).

Não existem literais negativos (números negativos resultam da operação unária **-**).

Note-se que um literal sem **.** (ponto decimal) nem parte exponencial é do tipo inteiro.

Exemplos: **3.14**, **1E3** = 1000 (número inteiro representado em virgula flutuante). **12.34e-24** = 12.34 x 10^{-24} (notação cientifica).

Se não for possível representar um literal double na máquina, devido a overflow, deverá ser gerado um erro lexical.

Cadeias de caracteres

As cadeias de caracteres são delimitadas por aspas (**"**) e podem conter quaisquer caracteres, excepto ASCII NUL (0x00) e ASCII LF (0x0A). Nas cadeias, os delimitadores de comentários não têm significado especial. Se for escrito um literal que contenha **\0**, então a cadeia termina nessa posição. Exemplo: **ab\0xy** tem o mesmo significado que **ab**.

É possível designar caracteres por sequências especiais (iniciadas por \), especialmente úteis quando não existe representação gráfica directa. As sequências especiais correspondem aos caracteres ASCII HT, LF e CR (**\t**, **\n** e **\r**, respectivamente), aspa (**\"**). *backslash* (\\), ou a quaisquer outros especificados através de 1 a 3 dígitos em base 8, designando valores de 8 bits (e.g., **\012** ou apenas **\12** se o carácter seguinte não representar um dígito em base 8). Exemplo: **xy\012z** tem o mesmo significado que **xy\12z** e que **xy\nz**.

Elementos lexicais distintos que representem duas ou mais cadeias consecutivas são representadas na linguagem como uma única cadeia que resulta da concatenação. Exemplo: **"ab" "cd"** é o mesmo que **"abcd"**.

Ponteiros

O único literal admissível para ponteiros corresponde ao ponteiro nulo e é indicado pela palavra-chave **null**. Este literal é compatível com todos os tipos de ponteiro. Os inteiros não são convertíveis em ponteiros, pelo que o valor **0** (zero) não é um valor

inicial admissível para ponteiros.

Gramática

A gramática da linguagem está resumida abaixo. Considerou-se que os elementos em tipo fixo são literais, que os parênteses curvos agrupam elementos, que elementos alternativos são separados por uma barra vertical, que elementos opcionais estão entre parênteses rectos, que os elementos que se repetem zero ou mais vezes estão entre ⟨ e ⟩. Alguns elementos usados na gramática também são elementos da linguagem descrita se representados em tipo fixo (e.g., parênteses).

ficheiro	→	⟨ *declaração* ⟩ [*programa-principal*]
declaração	→	[*qualificador*] *tipo identificador* [**=** *expressão*] **;**
	→	[*qualificador*] [**auto**] *identificador* **=** *expressão* **;**
programa-principal	→	**begin** ⟨ *declaração* ⟩ ⟨ *instrução* ⟩ **end**
função	→	**( ) ->** *tipo bloco*
	→	**(** *variáveis* **) ->** *tipo bloco*
variáveis	→	*variável* ⟨ **,** *variável* ⟩
tipo	→	**int** | **double** | **string** | **void** | **[** *tipo* **]** | *tipo-de-função*
tipo-de-função	→	*tipo* **< >**
	→	*tipo* **<** *tipos* **>**
tipos	→	*tipo* ⟨ **,** *tipo* ⟩
bloco	→	**{** ⟨ *declaração* ⟩ ⟨ *instrução* ⟩ **}**
instrução	→	*expressão* **;** | *expressões* **!** | *expressões* **!!**
	→	**stop** [*literal-inteiro*] **;** | **next** [*literal-inteiro*] **;** | **return** [*expressão*] **;**
	→	*instrução-condicional* | *instrução-de-iteração* | *bloco*
instrução-condicional	→	**if (** *expressão* **)** *instrução* ⟨ **elif (** *expressão* **)** *instrução* ⟩ [**else** *instrução*]
instrução-de-iteração	→	**while (** *expressão* **)** *instrução*
expressões	→	*expressão* ⟨ **,** *expressão* ⟩

Tipos, identificadores, literais e definição de expressões

Algumas definições foram omitidas da gramática: tipos de dados. qualificadores e variável (ver declarações de variáveis), identificador (ver identificadores), literal (ver literais); expressão (ver expressões). Note-se que *função* é qualquer especificação de função (corpo ou ponteiro com o tipo apropriado). Neste sentido, as funções contam como expressões primitivas.

Quanto a tipos de dados, **int** designa valores inteiros, **double** designa valores reais, **string** designa cadeias de caracteres. Os ponteiros são tipos compostos por um tipo entre parênteses rectos, e.g., **[int]**.

Ponteiros para funções são definidos a partir dos tipos de dados anteriormente descritos e do tipo especial **void** (ver a seguir). O tipo é indicado com o formato indicado na gramática, i.e., tipo de retorno seguido dos tipos dos argumentos (zero ou mais separados por vírgulas).

Um ponteiro declarado com um tipo de função indica o endereço da função correspondente. Estes ponteiros não suportam aritmética de ponteiros.

O tipo **void** apenas pode ser usado para indicar a ausência de retorno ou para declarar um ponteiro genérico. Neste caso, o aninhamento é irrelevante, i.e., **[void]** e **[[[void]]]** são equivalentes. Um ponteiro deste tipo é compatível com todos os outros tipos de ponteiros. A aritmética de ponteiros decrementa/incrementa em uma unidade o valor de um ponteiro do tipo **[void]**.

Left-values

Os *left-values* são posições de memória que podem ser modificadas (excepto onde proibido pelo tipo de dados). Os elementos de uma expressão que podem ser utilizados como left-values encontram-se individualmente identificados na semântica das expressões.

Ficheiros

Um ficheiro é designado por principal se contiver a função principal (onde se inicia o programa).

Declaração de variáveis

Uma declaração de variável indica sempre um tipo de dados (implícito ou explícito) e um identificador.

Exemplos:

- Inteiro: **int i**
- Real: **double r**
- Cadeia de caracteres: **string s**
- Ponteiro para inteiro: **[int] p1** (equivalente a **int*** em C)
- Ponteiro para real: **[double] p2** (equivalente a **double*** em C)
- Ponteiro para cadeia de caracteres: **[string] p3** (equivalente a **char**** em C)
- Ponteiro para ponteiro para inteiro: **[[int]] p4** (equivalente a **int**** em C)

É possível usar o pseudo-tipo **auto** se a declaração tiver um valor inicial: neste caso, o tipo é o do valor inicial.

Exemplo:

- Variável inteira: **auto i = 1**
- Variável real: **auto f = 2.0**
- etc.

Símbolos globais

Por omissão, os símbolos são privados a um módulo, não podendo ser importados por outros módulos.

A palavra-chave **public** permite declarar um identificador como público, tornando-o acessível a partir de outros módulos. Quando usado com a palavra-chave **auto**, esta é opcional. Note-se que a declaração de uma variável tem de ter sempre, ou o qualificador, ou o tipo, podendo estar ambos presentes.

A palavra-chave **forward** permite declarar num módulo variáveis definidas noutros módulos. Neste caso, não pode ser especificado o valor inicial dessas variáveis, pelo que não é possível o uso de **auto** em conjunto com **forward**.

A palavra-chave **foreign** deve ser usada para declarar símbolos de função com uma convenção de chamada diferente da da linguagem MML, e.g. para importar funções definidas em C. Além de poderem ser usados para chamar as funções que designam, os símbolos assim declarados podem ser atribuídos a variáveis com o tipo apropriado.

Exemplos:

- Declarar variável privada ao módulo: **double d1 = 22.0**
- Declarar variável pública: **public double d2 = 7.0**
- Declarar variável pública: **public auto d2 = 7.0** (igual à anterior)

- Declarar variável pública: **public d2 = 7.0** (igual à anterior)
- Usar definição externa de variável pública: **forward double d2**
- Usar ponteiro para função (nativa MML) definida externamente: **forward int<int> factorial**
- Usar função definida noutra linguagem (e.g. em C): **foreign [void]<int> malloc**

Inicialização

Quando existe, é a designação do objecto que segue o símbolo **=**: inteiro, real, cadeia de caracteres, ponteiro ou função. Entidades reais podem ser inicializadas por expressões inteiras (conversão implícita). A expressão de inicialização deve ser um literal se a variável for global. A associação de valores funcionais a variáveis pode ser realizada quando os tipos forem covariantes.

A palavra **auto** pode ser usada em lugar do tipo para declarar uma variável. Quando usada, o tipo da variável é inferido a partir do valor inicial (nestes casos, o valor inicial é obrigatório).

Exemplos:

- Inteiro (literal): **int i = 3**
- Inteiro (expressão): **int i = j + 1**
- Real (literal): **double r = 3.2**
- Real (expressão): **double r = i - 2.5**
- Cadeia de caracteres (literal): **string s = "olá"**
- Cadeia de caracteres (literais): **string s = "olá" "mãe"**
- Ponteiro (literal): **[[[double]]] p = null**
- Ponteiro (expressão): **[int] p = q + 1**
- Função:

```
int<int> f1
int<double> g1
double<int> g2
int<int> f2 = f1  // ok: mesmo tipo
f2 = g1 // ok: tipos covariantes
f2 = g2 // ERRADO
```

Funções

Uma função permite agrupar um conjunto de instruções num corpo, executado com base num conjunto de parâmetros (os argumentos formais), quando é invocada a partir de uma expressão.

Declaração

As funções são anónimas, mas sempre referenciadas por identificadores ou ponteiros, tal como outros tipos de dados. O tipo de retorno de uma função que não produz valores de retorno é **void**.

As funções que recebam argumentos devem indicá-los no cabeçalho. Funções sem argumentos definem um cabeçalho vazio. Os qualificadores de exportação/importação **public** ou **forward** (ver símbolos globais) são aplicados às variáveis usadas para referir as funções. Não é possível aplicá-los às declarações dos argumentos de uma função. Não é possível especificar valores por omissão para os argumentos de uma função, nem para o valor de retorno.

A declaração de uma variável do tipo função sem ser iniciada é utilizada para caracterizar um identificador exterior ou para efectuar declarações antecipadas (utilizadas para pré-declarar funções que sejam usadas antes de ser definidas, por exemplo, entre duas funções mutuamente recursivas).

Invocação

Uma função pode ser invocada através de qualquer expressão (ponteiro) do tipo apropriado que refira essa função (ponteiro não

nulo). O símbolo **@** pode ser usado dentro da própria função para efectuar uma invocação recursiva. Não é possível o uso de **@** no programa principal.

Se existirem argumentos, na invocação da função, o ponteiro é seguido de uma lista de expressões delimitadas por parênteses curvos. Esta lista é uma sequência, possivelmente vazia, de expressões separadas por vírgulas. O número e tipo de parâmetros actuais deve ser igual ao número e tipo dos parâmetros formais da função invocada (a menos de conversões implícitas). A ordem dos parâmetros actuais deverá ser a mesma dos argumentos formais da função a ser invocada.

De acordo com a convenção Cdecl, a função chamadora coloca os argumentos na pilha e é responsável pela sua remoção, após o retorno da chamada. Assim, os parâmetros actuais devem ser colocados na pilha pela ordem inversa da sua declaração (i.e., são avaliados da direita para a esquerda antes da invocação da função e o resultado passado por cópia/valor). O endereço de retorno é colocado no topo da pilha pela chamada à função.

Corpo

O corpo de uma função consiste num bloco que contém declarações (opcionais) seguidas de instruções (opcionais). Não é possível aplicar os qualificadores de exportação (**public**) ou de importação (**forward** ou **foreign**) (ver símbolos globais) dentro do corpo de uma função.

Uma instrução **return** causa a interrupção da função. O valor devolvido por uma função, através de expressão usada como argumento da instrução **return**, deve ser do tipo declarado no cabeçalho da função.

É um erro especificar um valor de retorno se a função for declarada como não retornando um valor (indicada como **void**).

Qualquer bloco (usado, por exemplo, numa instrução condicional ou de iteração) pode definir variáveis, cujos valores podem ser outras funções. Funções definidas dentro de um bloco não têm acesso às variáveis em contexto na função actual onde ocorrem.

Função principal e execução de programas

Um programa é a sequência de instruções que seguem as declarações globais num ficheiro, delimitadas pelas palavras-chave **begin** e **end**. Esta sequência forma o que em algumas linguagens se chama a função principal. Os argumentos com que o programa foi chamado podem ser obtidos através de funções **argc** (devolve o número de argumentos); **argv** (devolve o n-ésimo argumento como uma cadeia de caracteres, com n>0); e **envp** (devolve a n-ésima variável de ambiente como uma cadeia de caracteres, com n>0). Apenas um dos módulos do programa pode definir a função principal, i.e., se existirem vários módulos, apenas um deles pode conter mais do que declarações de variáveis globais.

```
foreign int<> argc
foreign string<int> argv
foreign string<int> envp
```

O valor de retorno da função principal é devolvido ao ambiente que invocou o programa. Este valor de retorno segue as seguintes regras (sistema operativo): 0 (zero), execução sem erros; 1 (um), argumentos inválidos (em número ou valor); 2 (dois), erro de execução. Os valores superiores a 128 indicam que o programa terminou com um sinal. Em geral, para correcto funcionamento, os programas devem devolver 0 (zero) se a execução foi bem-sucedida e um valor diferente de 0 (zero) em caso de erro.

A biblioteca de run-time (RTS) contém informação sobre outras funções de suporte disponíveis. incluindo chamadas ao sistema (ver também o manual da RTS).

Instruções

Excepto quando indicado, as instruções são executadas em sequência.

Blocos

Cada bloco tem uma zona de declarações de variáveis locais (facultativa), seguida por uma zona com instruções (possivelmente vazia). É possível declarar e definir funções dentro de blocos.

A visibilidade das variáveis é limitada ao bloco em que foram declaradas. As entidades declaradas podem ser directamente utilizadas em sub-blocos ou passadas como argumentos para funções chamadas dentro do bloco. Caso os identificadores usados para definir as variáveis locais já estejam a ser utilizados para definir outras entidades ao alcance do bloco, o novo identificador passa a referir uma nova entidade definida no bloco até que ele termine (a entidade previamente definida continua a existir, mas não pode ser directamente referida pelo seu nome). Esta regra é também válida relativamente a argumentos de funções (ver corpo das funções).

Instrução condicional

Esta instrução tem comportamento semelhante ao da instrução **if-else** em C. As partes correspondentes a **elif** comportam-se como encadeamentos de instruções condicionais na parte **else** de um **if-else** à la C.

Instrução de iteração

Esta instrução tem comportamento idêntico ao da instrução **while** em C.

Instrução de terminação

A instrução **stop** termina o n-ésimo ciclo mais interior em que a instrução se encontrar (quando o argumento é omitido, assume-se **n=1**), tal como a instrução **break** em C. Esta instrução só pode existir dentro de um ciclo, sendo a última instrução do seu bloco.

Instrução de continuação

A instrução **next** reinicia o n-ésimo ciclo mais interior em que a instrução se encontrar (quando o argumento é omitido, assume-se **n=1**), tal como a instrução **continue** em C. Esta instrução só pode existir dentro de um ciclo, sendo a última instrução do seu bloco.

Instrução de retorno

A instrução **return**, se existir, é a última instrução do seu bloco. Ver comportamento na descrição do corpo de uma função.

Expressões como instruções

As expressões utilizadas como instruções são sempre avaliadas, mesmo que não produzam efeitos secundários.

Instruções de impressão

As notações **!** e **!!** podem ser utilizadas para apresentar valores na saída do programa. A primeira forma apresenta os valores sem mudar de linha; a segunda forma apresenta os valores mudando de linha depois de os apresentar a todos. Quando existe mais de uma expressão, as várias expressões são apresentadas sem separação. Valores numéricos (inteiros ou reais) são impressos em decimal. As cadeias de caracteres são impressas na codificação nativa. Ponteiros não podem ser impressos.

Expressões

Uma expressão é uma representação algébrica de uma quantidade: todas as expressões têm um tipo e devolvem um valor.

Existem expressões primitivas e expressões que resultam da avaliação de operadores.

A tabela seguinte apresenta as precedências relativas dos operadores: é a mesma para operadores na mesma linha, sendo as linhas seguintes de menor prioridade que as anteriores. A maioria dos operadores segue a semântica da linguagem C (excepto onde explicitamente indicado). Tal como em C, os valores lógicos são 0 (zero) (valor falso), e diferente de zero (valor verdadeiro).

Tipo de Expressão	Operadores	Associatividade	Operandos	Semântica

primária	**( ) [ ]**	não associativos	-	parênteses curvos, indexação, reserva de memória
unária	**+ - ?**	não associativos	-	identidade e simétrico, indicação de posição
multiplicativa	*** / %**	da esquerda para a direita	inteiros, reais	C (% é apenas para inteiros)
aditiva	**+ -**	da esquerda para a direita	inteiros, reais, ponteiros	C: se envolverem ponteiros, calculam: (i) deslocamentos, i.e., um dos operandos deve ser do tipo ponteiro e o outro do tipo inteiro; (ii) diferenças de ponteiros, i.e., apenas quando se aplica o operador **-** a dois ponteiros do mesmo tipo (o resultado é o número de objectos do tipo apontado entre eles). Se a memória não for contígua, o resultado é indefinido.
comparativa	**< > <= >=**	da esquerda para a direita	inteiros, reais	C
igualdade	**== !=**	da esquerda para a direita	inteiros, reais, ponteiros	C
"não" lógico	**~**	não associativo	inteiros	C
"e" lógico	**&&**	da esquerda para a direita	inteiros	C: o 2º argumento só é avaliado se o 1º não for falso.
"ou" lógico	**||**	da esquerda para a direita	inteiros	C: o 2º argumento só é avaliado se o 1º não for verdadeiro.
atribuição	**=**	da direita para a esquerda	todos os tipos	O valor da expressão do lado direito do operador é guardado na posição indicada pelo *left-value* (operando esquerdo do operador). Podem ser atribuídos valores inteiros a *left-values* reais (conversão automática). Nos outros casos, ambos os tipos têm de concordar. O literal **null** é compatível com todos os tipos de ponteiros.

Expressões primitivas

As expressões literais e a invocação de funções foram definidas acima.

Identificadores

Um identificador é uma expressão se tiver sido declarado. Um identificador pode denotar uma variável.

Um identificador é o caso mais simples de um *left-value*, ou seja, uma entidade que pode ser utilizada no lado esquerdo (*left*) de uma atribuição.

Leitura

A operação de leitura de um valor inteiro ou real pode ser efectuado pela expressão indicada pela palavra-chave **input**, que devolve o valor lido, de acordo com o tipo esperado (inteiro ou real). Caso se use como argumento dos operadores de impressão ou noutras situações que permitam vários tipos (e.g. **!** ou **!!**), deve ser lido um inteiro.

Exemplos: **a = input** (leitura para **a**), **f(input)** (leitura para argumento de função), **input!!** (leitura e impressão), **@(input)** (chamada recursiva a função com argumento que é lido da entrada).

Parênteses curvos

Uma expressão entre parênteses curvos tem o valor da expressão sem os parênteses e permite alterar a prioridade dos

operadores. Uma expressão entre parênteses não pode ser utilizada como *left-value* (ver também a expressão de indexação).

Funções

As funções (ponteiros ou o seu código: não confundir com chamadas a funções) podem ser usadas como expressões tipificadas como funções, i.e., ponteiros para funções (mesmo quando não se usa explicitamente um ponteiro).

Exemplo:

```
auto f = (int i) -> int { return i + 1; };
auto g = (int n, int<int> fun) -> int { return fun(n); };

begin
  g(3, f)!!  // escreve 4
  g(3, (int i) -> int { return i * 2; })!!  // escreve 6
end
```

Expressões resultantes de avaliação de operadores

Indexação de ponteiros

A indexação de ponteiros devolve o valor de uma posição de memória indicada por um ponteiro. Consiste em uma expressão ponteiro seguida do índice entre parênteses rectos. O resultado de uma indexação de ponteiros é um *left-value*. Não é possível indexar ponteiros que designem funções.

Exemplo (acesso à posição 0 da zona de memória indicada por **p**): **p[0]**

Identidade e simétrico

Os operadores identidade (**+**) e simétrico (**-**) aplicam-se a inteiros e reais. Têm o mesmo significado que em C.

Reserva de memória

A expressão reserva de memória devolve o ponteiro que aponta para a zona de memória, na pilha da função actual, contendo espaço suficiente para o número de objectos indicados pelo seu argumento inteiro.

Exemplo (reserva vector com 5 reais, apontados por **p**): **[double] p = [5]**

Expressão de indicação de posição

O operador sufixo **?** aplica-se a *left-values*, retornando o endereço (com o tipo ponteiro) correspondente.

Exemplo (indica o endereço de **a**): **a?**

Expressão de dimensão

O operador **sizeof** tem um único argumento e aplica-se a expressões, retornando a dimensão correspondente em bytes.

Exemplo: **sizeof(a)** (dimensão de **a**).

Exemplos e Testes

Os exemplos abaixo não são exaustivos e não ilustram todos os aspectos da linguagem.

Estão ainda disponíveis outros pacotes de testes.

O seguinte exemplo ilustra um programa com dois módulos: um que define a função **factorial** e outro que define a função principal.

Definição da função **factorial** no ficheiro **factorial.mml**:

```
public factorial = (int n) -> int {
  if (n > 1)
    return n * @(n - 1);  // could also use 'factorial' (buggy approach!)
  else
    return 1;
};
```

Exemplo da utilização da função **factorial** no ficheiro **main.mml**:

```
// external builtin functions (non-MML)
foreign int<> argc;
foreign string<int> argv;
foreign int<string> atoi;

// external user functions (MML)
forward int<int> factorial;

// the main function
begin
  auto value = 1;
  "Teste para a função factorial."!!
  if (argc() == 2) {
    string s = argv(1);
    value = atoi(s);
  }
  value, "! é ", factorial(value)!!
  return 0;
end
```

Como compilar:

```
mml --target asm factorial.mml
mml --target asm main.mml
yasm -felf32 factorial.asm
yasm -felf32 main.asm
ld -melf_i386 -o main factorial.o main.o -lrts
```

Omissões e Erros

Casos omissos e erros serão corrigidos em futuras versões do manual de referência.

Categories: **Projecto de Compiladores** **Compiladores** **Ensino**