TÉCNICO LISBOA

Projecto de Compiladores
Ano lectivo 2012/2013, 2º Semestre

MAYFLY (manual de referência)
17 de Fevereiro de 2013

Esclarecimento de dúvidas:

- Consultar sempre o corpo docente atempadamente: presencialmente ou através do endereço **co13@l2f.inesc-id.pt**.
- Não utilizar fontes de informação não oficialmente associadas ao corpo docente (podem colocar em causa a aprovação à disciplina).
- Não são aceites justificações para violações destes conselhos: quaisquer consequências nefastas são da responsabilidade do aluno.

Requisitos para desenvolvimento, material de apoio e actualizações do enunciado (ver informação completa na secção **[Projecto]** no Fénix):

- O material de apoio é de uso obrigatório e não pode ser alterado.
- Verificar atempadamente (mínimo de 48 horas antes do final de cada prazo) os requisitos exigidos pelo processo de desenvolvimento.

Processo de avaliação (ver informação completa nas secções **[Projecto]** e **[Método de Avaliação]** no Fénix):

- Datas: **2013/03/26 12:00** (inicial); **2013/04/19 12:00** (intercalar); **2013/05/21 12:00** (final); **2013/05/27–2013/05/31** (teste prático).
- A entrega inicial, sendo crucial para o projecto, é obrigatória e sua não realização implica a exclusão da avaliação do projecto e, por consequência, da avaliação da disciplina em 2012/2013.
- Verificar atempadamente (até 48 horas antes do final de cada prazo) os requisitos exigidos pelo processo de avaliação, incluindo a capacidade de acesso ao repositório CVS.
- **Apenas se consideram para avaliação os projectos existentes no repositório CVS oficial.**
- Trabalhos não presentes no repositório no final do prazo têm classificação 0 (zero) (não são aceites outras formas de entrega). Não são admitidas justificações para atrasos em sincronizações do repositório. A indisponibilidade temporária do repositório, desde que inferior a 24 horas, não justifica atrasos na submissão de um trabalho.
- A avaliação do projecto pressupõe o compromisso de honra de que o trabalho correspondente foi realizado pelos alunos correspondentes ao grupo de avaliação. **Fraudes na execução do projecto terão como resultado a exclusão dos alunos implicados do processo de avaliação em 2012/2013**.

A linguagem MAYFLY é uma linguagem imperativa e é apresentada de forma intuitiva neste manual. São apresentadas características básicas da linguagem (§1, §2); convenções lexicais (§3); estrutura/sintaxe (§4); especificação das funções (§5); semântica das instruções (§6); semântica das expressões (§7); e, finalmente, alguns exemplos (§8).

1 Tipos de dados

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

- Tipos numéricos: os **inteiros**, em complemento para dois, ocupam 4 bytes; os **reais**, em vírgula flutuante, ocupam 8 bytes.
- As **cadeias de caracteres** são definidas como vectores de caracteres terminados por ASCII NULL (`0x00`, `\0`). Variáveis e literais deste tipo só podem ser utilizados em atribuições, impressões, ou como argumentos/retornos de funções.
- Os **ponteiros** representam endereços de outros objectos (§4), ocupando 4 bytes. Podem ser objecto de operações aritméticas (deslocamentos) e permitem aceder ao valor apontado.

Os tipos de dados condicionam os resultados de alguns operadores. Os tipos suportados por cada operador e a operação a realizar são indicados na definição das expressões (§7).

2 Manipulação de nomes

Os nomes (§3.6) correspondem a constantes, variáveis e funções. Nos pontos que se seguem, usa-se o termo entidade para as designar indiscriminadamente, usando-se listas explícitas quando a descrição for válida apenas para um subconjunto.

2.1 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 função (locais). A reutilização de identificadores em contextos inferiores encobre possíveis declarações em contextos superiores. Em particular, redeclarações de identificadores numa função encobrem os globais até ao fim da função. É possível utilizar símbolos globais nos contextos das funções, mas não é possível defini-los (§4.4.3).

2.2 Validade das variáveis

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

3 Convenções Lexicais

Para cada grupo de elementos lexicais (*tokens*), considera-se a maior sequência de caracteres constituindo um elemento válido.

3.1 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`).

3.2 Comentários

Existem dois tipos de comentários, que também funcionam como elementos separadores: **explicativos**, começam com >> (desde que >> não faça parte de uma cadeia de caracteres), e acabam no fim da linha; e **operacionais**, começam com =< e terminam com => (se não fizerem parte de cadeias de caracteres), podendo estar aninhados.

3.3 Palavras chave

As seguintes palavras são reservadas e não constituem identificadores (devem ser escritas exactamente como indicado):

```
void integer number string public const if then else
do while for in step upto downto continue break return
```

3.4 Operadores de expressões

São considerados operadores os elementos lexicais apresentados na tabela 1 (página 6).

3.5 Delimitadores e terminadores

Os elementos lexicais seguintes são considerados delimitadores/terminadores: `, ; : ! !! ( )`

3.6 Identificadores (nomes)

São iniciados por uma letra ou por _ (sublinhado), seguindo-se 0 (zero) ou mais letras, dígitos ou _. 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.

3.7 Literais

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

3.7.1 Inteiros

Um literal inteiro é um número não negativo (uma constante inteira pode, contudo, ser negativa: números negativos são construídos pela aplicação do operador menos unário (−) a um literal positivo).

Um literal inteiro em decimal é constituído por uma sequência de 1 (um) ou mais dígitos de `0` a `9`, em que o primeiro dígito não é um `0` (zero), excepto no caso do número 0 (zero). Neste caso, é composto apenas pelo dígito `0` (zero) (em qualquer base).

Um literal inteiro em octal começa sempre pelo dígito `0` (zero), sendo seguido de um ou mais dígitos de `0` a `7` (note-se que `09` é um literal octal inválido com valor `011`). Exemplo: `007`.

Um literal inteiro em hexadecimal começa sempre pela sequência `0x`, sendo seguido de um ou mais digitos de `0` a `9`, de `a` a `f` ou de `A` a `F`. As letras de `a` a `f`, ou de `A` a `F`, representam os valores de 10 a 15 respectivamente. Exemplo: `0x07`.

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

3.7.2 Reais em vírgula flutuante

Os literais reais são expressos em notação científica (tal como em C). Um literal sem `.` (ponto decimal) nem parte exponencial é do tipo inteiro. Exemplo: `12.34e-24` = 12.34×10^{-24}

3.7.3 Cadeias de caracteres

As cadeias de caracteres são delimitadas por aspas (`"`) e podem conter quaisquer caracteres, excepto ASCII NULL (`0x00` `\0`). Nas cadeias, os delimitadores de comentários não têm qualquer significado especial.

Além dos caracteres individuais, são ainda possíveis sequências especiais (iniciadas por `\`) que designam caracteres que não têm representação gráfica directa. As sequências especiais correspondem aos caracteres ASCII LF, CR e HT (`\n`, `\r` e `\t`, respectivamente), aspa (`\"`), barra (`\\`), ou a quaisquer outros especificados através de 1 ou 2 digitos hexadecimais (e.g. `\0a` ou apenas `\A` se o carácter seguinte não representar um dígito hexadecimal).

Elementos lexicais distintos que representem duas ou mais cadeias de caracteres consecutivas são representadas na linguagem como uma única cadeia que resulta da concatenação.

3.7.4 Ponteiros

O único literal admissível para ponteiros é `0` (zero), indicando o ponteiro nulo.

4 Gramática

A gramática da linguagem pode ser resumida pelas regras 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* ⟩
declaração	→	*variável* **;** | *função*
variáveis	→	*variável* ⟨ **,** *variável* ⟩
variável	→	[**public**] [**const**] *tipo ident* [**=** *expressão*]
função	→	[**public**] (*tipo* | **void**) *ident* **(** [*variáveis*] **)** [**=** *literal*] [*corpo*]
tipo	→	**integer** | **number** | **string** | *tipo* *****
corpo	→	*bloco*
bloco	→	{ ⟨ *declaração* ⟩ ⟨ *instrução* ⟩ }
instrução	→	*expressão* (**;** | **!** | **!!**)
	→	**break** [*literal-inteiro*] **;** | **continue** [*literal-inteiro*] **;** | **return**
	→	*instrução-condicional* | *instrução-de-iteração* | *bloco*
instrução-condicional	→	**if** *expressão* **then** *instrução* [**else** *instrução*]
instrução-de-iteração	→	**for** *left-value* **in** *expressão* (**upto** | **downto**) *expressão* [**step** *expressão*] **do** *instrução*
	→	**do** *instrução* **while** *expressão* **;**

4.1 Tipos, elementos lexicais e definição de expressões

Algumas definições foram omitidas: tipos de dados (§1), *identificador* (§3.6), *literal* (§3.7); expressões (§7).

4.2 *Left value*

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 em §7.

4.3 Ficheiros

Um ficheiro é designado por principal se contiver a função principal (a que inicia o programa) (§5.4).

4.4 Declaração de variáveis e constantes

As declarações de variáveis ou constantes incluem os componentes descritos nos pontos seguintes.

4.4.1 Declarações

Um declaração indica sempre um tipo de dados (§1) e um nome de variável (§2). Exemplos:

Inteiro:	`integer i`	Real:	`number r`
Ponteiro:	`const integer *pi`	Cadeia de caracteres:	`string s`

4.4.2 Constante

A linguagem define identificadores constantes, impedindo que o identificador declarado possa ser utilizado em operações que modifiquem o seu valor. Caso um identificador designe uma constante inteira não pública (ver §4.4.3) o seu valor deverá ser directamente substituído no código, não ocupando espaço. Exemplos:

Constante:	`const integer ci = 2`
Ponteiro para constante:	`const integer *pci = &ci`

4.4.3 Símbolos globais

A palavra `public` gere a exportação/importação de identificadores globais. Se aplicado a uma entidade inicializada/definida, declara-a como globalmente acessível (i.e., por outros módulos). Se aplicado a uma entidade não inicializada/definida, declara-a como estando definida num outro módulo.

4.4.4 Inicialização

A existir, define-se com um valor que segue o sinal = ("igual"): inteiro (uma expressão inteira), real (uma expressão real), ponteiro (uma expressão do tipo ponteiro). A expressão corresponde a um literal se a variável não for local. Exemplos:

Inteiro (literal):	`integer i = 3`	Real (literal):	`number r = 3.2`
Inteiro (expressão):	`integer i = j+1`	Real (expressão):	`number r = s - 2.5 + f(3)`
Cadeia de caracteres (literal):	`const string cs = "olá"`	Ponteiro (literal):	`integer *p = 0`
Cadeia de caracteres (literal):	`string s = "olá" "mãe"`	Ponteiro (expressão):	`number *p = q+1`

As cadeia de caracteres são (possivelmente) inicializadas com uma lista não nula de valores sem separadores (opcionalmente, poderão existir caracteres brancos entre os elementos lexicais que compõem a cadeia de caracteres) (§3.7.3). Estes valores são sempre constantes, independentemente de o identificador que as designa ser constante ou não.

Notar que declarações de constantes sem valor inicial só são possíveis se corresponderem a identificadores pré-declarados, pertencentes a outros módulos (§4.4.3).

5 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.

5.1 Declaração

As funções são sempre designadas por identificadores constantes precedidos do tipo de dados devolvido pela função.

As funções que recebam argumentos devem indicá-los no cabeçalho. Funções sem argumentos definem um cabeçalho vazio. Não é possível aplicar o qualificador `public` (§4.4.3) às declarações dos argumentos de uma função.

A declaração de uma função sem corpo é utilizada para tipificar 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). Caso a declaração tenha corpo, define-se uma nova função.

5.2 Invocação

A função só pode ser invocada através de um identificador que refira uma função previamente declarada ou definida.

Se existirem argumentos, na invocação da função, o identificador é seguido de uma lista de expressões delimitadas por parênteses curvos. Esta lista é uma sequência, possivelmente vazia, de expressões separadas por vírgulas. As expressões são avaliadas da direita para a esquerda antes da invocação da função (Cdecl) e o resultado passado por cópia (passagem de argumentos por valor).

O número e tipo de parâmetros actuais deve ser igual ao número e tipo dos parâmetros formais da função invocada. Caso existam parâmetros opcionais, iniciados na declaração da função, os seus valores são utilizados na falta de parâmetros actuais e apenas se forem os últimos. A ordem dos parâmetros actuais deverá ser a mesma dos argumentos formais da função a ser invocada. Os parâmetros actuais devem ser colocados na pilha de dados pela ordem inversa da sua declaração (o primeiro no topo da pilha) e o endereço de retorno no topo da pilha.

A função chamadora coloca os argumentos na pilha e é responsável pela sua remoção, após o retorno da chamada (Cdecl).

5.3 Corpo

O corpo de uma função consiste num bloco que contém declarações seguidas de instruções. Ambos os grupos são opcionais. Não é possível aplicar o qualificador `public` (§4.4.3) dentro do corpo de uma função.

O valor devolvido por uma função, através de atribuição ao *left-value* especial com o mesmo nome da função, deve ser sempre do tipo declarado.

Se existir um valor declarado por omissão para o retorno da função (indicado pela notação `:=` seguindo a assinatura da função), então deve ser utilizado se não for especificado outro. A especificação do valor de retorno por omissão é obrigatoriamente um literal do tipo indicado. É um erro especificar um valor de retorno se o tipo de retorno for `void`. Uma função cujo retorno seja inteiro ou ponteiro retorna 0 (zero) por omissão (i.e., se não for especificado o valor de retorno). Em todos os outros casos, o valor de retorno é indeterminado se não for definido explicitamente.

Uma instrução `return` causa a interrupção da função e o retorno do seu valor actual ao chamador.

Um sub-bloco de função (usado, por exemplo, numa instrução condicional ou de iteração) pode definir variáveis ou constantes.

5.4 Função principal e execução de programas

Um programa inicia-se com a invocação da função `mayfly` (sem argumentos). Os argumentos com que o programa foi chamado podem ser obtidos através de funções `argc()` (devolve o número de argumentos); `string argv(integer n)` (devolve o n-ésimo argumento como uma cadeia de caracteres) ($n > 0$); e `string envp(integer n)` (devolve a n-ésima variável de ambiente como uma cadeia de caracteres) ($n > 0$).

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).

6 Instruções

Excepto quando indicado, as instruções são executadas em sequência.

6.1 Blocos

Cada bloco tem uma zona de declarações de constantes e variáveis locais (facultativa), seguida por uma zona com instruções.

A longevidade das variáveis, bem como a sua visibilidade, 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 constantes e 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 ao 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 (§5.3).

6.2 Instrução condicional: if-then-else

Comportamento idêntico ao da instrução `if-then-else` em C.

6.3 Instruções de iteração: for

Por cada valor da variável que conta entre dois limites, de acordo com um passo (unitário, por omissão), é executada uma instrução. Se a variável estiver fora dos limites, não executa a instrução. O comportamento do incremento e execução são análogos aos da instrução `for` em C.

6.4 Instruções de iteração: do-while

Comportamento idêntico ao do ciclo `do-while` em C.

6.5 Instrução de terminação: break

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 em C. Esta instrução só pode existir dentro de um ciclo. Se existir, é a última instrução do seu bloco.

6.6 Instrução de continuação: continue

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 em C. Esta instrução só pode existir dentro de um ciclo. Se existir, é a última instrução do seu bloco.

6.7 Instrução de retorno: return

Se existir, é a última instrução do seu bloco. Ver comportamento em §5.3.

6.8 Expressão como instrução

As expressões podem ser utilizadas como instrução, mesmo que não produzam efeitos secundários. Expressões terminadas por `!` ou `!!` (impressão com mudança de linha) o seu valor deve ainda ser impresso. Valores numéricos (inteiros ou reais) são impressos em decimal. Valores do tipo `string` são impressos na codificação nativa. Ponteiros não podem ser impressos.

7 Expressões

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

Na tabela 1, a precedência dos operadores é a mesma para operadores na mesma linha, sendo as linhas seguintes de menor prioridade que as anteriores. As secções abaixo descrevem as várias expressões e operações.

	Operadores	**Associatividade**
primária	`( ) [ ]`	não associativos
unária	`* & - ++ --`	não associativos
multiplicativa	`* / %`	da esquerda para a direita
aditiva	`+ -`	da esquerda para a direita
comparativa	`< > <= >=`	da esquerda para a direita
igualdade	`== <>`	da esquerda para a direita
"não" lógico	`~`	não associativo
"e" lógico	`&`	da esquerda para a direita
"ou" lógico	`|`	da esquerda para a direita
atribuição	`=`	da direita para a esquerda

Tabela 1: Grupos e precedências de expressões.

Tal como em C, os valores lógicos são 0, para falso, e diferente de zero, para verdadeiro.

7.1 Expressões primárias

7.1.1 Identificadores

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

Um identificador é o caso mais simples de um left-value (§4.2), ou seja, uma entidade que pode ser utilizada no lado esquerdo (left-value) de uma atribuição. O valor de retorno de uma função é definido por um left-value especial (§5.3).

7.1.2 Literais

Os literais são como definidos nas convenções lexicais (§3.7).

7.1.3 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 §7.1.4).

7.1.4 Indexação

Uma expressão indexação devolve o valor contido numa posição indicada por um ponteiro. Consiste de uma expressão ponteiro seguida do índice entre parênteses rectos. Se a expressão ponteiro for um left-value, então a expressão indexação poderá também ser um left-value (excepto nas condições proibidas pelo tipo §4.2). O operador prefixo `*` indexa a primeira posição. Exemplos: `*p` e `p[0]` (acesso à posição apontada por `p`).

7.1.5 Invocação

Uma função só pode ser invocada através de um identificador que corresponda a uma função previamente declarada ou definida.

7.1.6 Leitura

A operação de leitura de um valor inteiro ou real pode ser efectuado pela expressão `@`, que devolve o valor lido, de acordo com o tipo esperado (inteiro ou real). Caso se use como argumento dos operadores de impressão (`!` ou `!!`), deve ser lido um inteiro.

7.2 Expressões unárias

7.2.1 Identidade e simétrico

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

7.2.2 Incremento e decremento

Os operadores de incremento (++) e decremento (−−) aplicam-se a inteiros e ponteiros. Têm o mesmo significado que em C.

7.2.3 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. O tipo de retorno é idêntico ao do left-value utilizado e o espaço deve ser calculado em função do tipo apontado. Exemplo: `integer *pi = #5`

7.3 Expressões aditivas

Os operadores aditivos aplicam-se a inteiros e reais. Têm o mesmo significado que em C.

Estas operações podem ainda realizar-se sobre ponteiros, tendo o significado das operações correspondentes em C/C++: (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). Operações sobre ponteiros para constantes produzem sempre ponteiros para constantes.

7.4 Expressões multiplicativas

Os operadores multiplicativos aplicam-se a inteiros e reais. Têm o mesmo significado que em C.

7.5 Expressões de grandeza

Os operadores de grandeza aplicam-se a inteiros e reais. Têm o mesmo significado que em C.

7.6 Expressões de igualdade

Os operadores de igualdade aplicam-se a inteiros, reais e ponteiros. Têm o mesmo significado que em C.

7.7 Expressões de negação lógica

A operação é aplicável a valores inteiros, devolvendo o valor falso caso o argumento verdadeiro e vice-versa.

7.8 Expressões de junção lógica

A operação é aplicável a valores inteiros. Devolve verdadeiro caso ambos os argumentos sejam verdadeiros e falso caso contrário. Se o primeiro argumento for falso, o segundo argumento não é avaliado.

7.9 Expressões de alternativa lógica

A operação é aplicável a valores inteiros. Devolve falso caso ambos os argumentos sejam falsos e verdadeiro caso contrário. Se o primeiro argumento for verdadeiro, o segundo argumento não é avaliado.

7.10 Expressões de atribuição

O valor da expressão do lado direito do operador é guardado na posição indicada pelo left-value do lado esquerdo do operador de atribuição. Só podem atribuídos valores a left-values do mesmo tipo.

7.11 Expressão de indicação de posição

Expressa pelo operador prefixo `&`, indica o endereço do objecto (endereçável) indicado como argumento. Retorna um valor do tipo do ponteiro apropriado para o objecto em causa. Exemplos: `&a` (indica o endereço de `a`).

8 Exemplos

Os exemplos não são exaustivos e não ilustram todos os aspectos da linguagem. Podem obter-se outros na página da disciplina.

Exemplo da definição de função num ficheiro (ficheiro `factorial.mf`):

```
public integer factorial(integer n = 1) = 1 {
  if n > 1 then factorial = n * factorial(n-1); else factorial = 1;
}
```

Exemplo da utilização da função noutro ficheiro (ficheiro `main.mf`):

```
// external builtin functions
public integer argc()
public string argv(integer n)
public integer atoi(string s)

// external user functions
public integer factorial(integer n)

// the main function
public integer main() = 0 {
  integer f = 1;
  "Teste para a função factorial"!!
  if argc() == 2 then
    f = atoi(argv(1));
  f! "! = "! factorial(f)!!
}
```

9 Omissões e Erros

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