OG Compiler

Tipos de Dados

A linguagem é fracamente tipificada (são efectuadas algumas conversões implícitas). Existem 5 tipos de dados (4 deles declaráveis explicitamente), 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 (IEEE 754).
- As **cadeias de caracteres** são 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 objectos e ocupam 4 bytes. Podem ser objecto de operações aritméticas (deslocamentos) e permitem aceder ao valor apontado.

Os tipos de dados podem ser manipulados individualmente, usando-se expressões desse tipo, ou em conjuntos (tipo estruturado) correspondentes a sequências de expressões (tuplos). O uso de tuplos é limitado a algumas situações (ver abaixo).

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

Manipulação de Nomes

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

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 declarações em contextos superiores: redeclarações locais podem encobrir as globais até ao fim de uma função. Não é possível importar ou definir símbolos globais nos contextos das funções (ver símbolos globais).

Não é possível definir funções dentro de blocos.

Validade das variáveis

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.

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 palavras indicadas de seguida estão reservados (palavras chave), não podendo ser utilizadas como identificadores. Estas palavras têm de ser escritas exactamente como indicado:

```
auto int real string ptr public require sizeof input nullptr
procedure break continue return if then elif else for do write writeln
```

O identificador **og**, embora não reservado, quando refere uma função, corresponde à função principal, devendo ser declarado público.

Tipos

Os seguintes elementos lexicais designam tipos em declarações (ver gramática): **int** (inteiro), **real** (real), **string** (cadeia de caracteres), **auto** (tipo inferido a partir do valor inicial), **ptr** (ponteiros). Ver gramática.

O tipo especial **auto** é utilizado para indicar que o tipo da variável ou do retorno da função deve ser inferido a partir do tipo do seu valor inicial. Quando aplicado a uma função, implica que o tipo de retorno deve ser inferido a partir da expressão que é usada na instrução **return** (se existirem múltiplas, todas devem concordar no tipo a retornar). Este tipo pode ser usado com uma sequência de identificadores, declarando um tuplo. Os tipos dos vários campos do tuplo são inferidos das posições correspondentes do valor inicial: pode ser uma sequência de expressões ou outro tuplo. Em ambos os casos, o número de elementos deve ser o mesmo que o dos identificadores declarados. Tuplos com um único elemento são idênticos ao tipo que contêm.

O tipo **auto** pode ser utilizado para definir ponteiros genéricos (como **void*** em C/C++), compatíveis com todos os tipos de ponteiros. São ainda o único tipo de ponteiro convertível para um número inteiro (o valor do inteiro é o valor do endereço de memória). O nível de aninhamento é irrelevante neste caso, i.e., **ptr<auto>** designa o mesmo tipo que **ptr<ptr<...ptr<auto>...>>**.

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 **(** e **)** (delimitadores de expressões).

Identificadores (nomes)

São iniciados por uma letra, seguindo-se 0 (zero) ou mais letras, dígitos ou **_** (sublinhado). 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 designam elementos cujo valor não pode ser alterado durante a execução do programa).

Inteiros

Um literal inteiro é um número não negativo. Uma constante inteira pode, contudo, ser negativa: números negativos são construídos pela aplicação do operador de negação aritmética unária (**-**) a um literal positivo.

Literais inteiros decimais são constituídos por sequências de 1 (um) ou mais dígitos de **0** a **9**.

Literais inteiros hexadecimais começam sempre com a sequência **0x**, seguida de um ou mais dígitos de **0** a **9** ou de **a** a **f** (sem distinguir maiúsculas de minúsculas). As letras 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.

Reais em vírgula flutuante

Os literais reais positivos são expressos tal como em C (apenas é suportada a base 10).

Não existem literais negativos (números negativos resultam da aplicação da operação de negação unária).

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).

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 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 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 consecutivas são representadas na linguagem como uma única cadeia que resulta da concatenação.

Exemplos:

- **"ab" "cd"** é o mesmo que **"abcd"**.
- **"ab" /* comentário com "cadeia de caracteres falsa" */ "cd"** é o mesmo que **"abcd"**.

Ponteiros

O único literal admissível para ponteiros é indicado pela palavra reservada **nullptr**, indicando o ponteiro nulo.

Gramática

A gramática da linguagem está resumida abaixo. Considerou-se

- que os elementos em tipo fixo são literais;
- que os parênteses curvos agrupam elementos: `(e)`;
- que elementos alternativos são separados por uma barra vertical: `|`;
- que elementos opcionais estão entre parênteses rectos: `[e]`;
- que os elementos que se repetem zero ou mais vezes estão entre `⟨e⟩`

.

ficheiro	declaração ⟨ declaração ⟩
declaração	variável ; | função | procedimento
variável	[public | require] tipo identificador [= expressão]
	[public] auto identificadores = expressões
função	[public | require] (tipo | auto) identificador ([variáveis]) [bloco]
procedimento	[public | require] procedure identificador ([variáveis]) [bloco]
identificadores	identificador ⟨ , identificador ⟩
expressões	expressão ⟨ , expressão ⟩
variáveis	variável ⟨ , variável ⟩
tipo	int | real | string | ptr< (tipo | auto) >
bloco	{ ⟨ declaração ⟩ ⟨ instrução ⟩ }
instrução	expressão ; | write expressões ; | writeln expressões ;
	break | continue | return [expressões] ;
	instrução-condicional | instrução-de-iteração | bloco
instrução condicional	if expressão then instrução
	if expressão then instrução ⟨elif expressão then instrução⟩ [else instrução]
instrução de iteração	for [variáveis] ; [expressões] ; [expressões] do instrução
	for [expressões] ; [expressões] ; [expressões] do instrução

Alguns elementos usados na gramática também são elementos da linguagem descrita se representados em tipo fixo (e.g., parênteses).

Tipos, identificadores, literais e definição de expressões

Algumas definições foram omitidas da gramática: tipos de dados, *identificador* (ver identificadores), *literal* (ver literais); *expressão* (ver expressões).

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 (a que inicia o programa).

Declaração de variáveis

Uma declaração de variável indica sempre um tipo de dados e um identificador.
Exemplos:

- Inteiro: **int i**
- Real: **real r**
- Cadeia de caracteres: **string s**
- Ponteiro para inteiro: **ptr<int> p1** (equivalente a **int*** em C)
- Ponteiro para real: **ptr<real> p2** (equivalente a **double*** em C)
- Ponteiro para cadeia de caracteres: **ptr<string> p3** (equivalente a **char**** em C)
- Ponteiro para ponteiro para inteiro: **ptr<ptr<int>> p4** (equivalente a **int**** em C)
- Ponteiro genérico: **ptr<auto> p5** (equivalente a **void*** em C)

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.
A palavra chave **require** (opcional para funções) permite declarar num módulo entidades definidas em outros módulos. As entidades não podem ser inicializadas nestas declarações.
Exemplos:

- Declarar variável privada ao módulo: **real pi = 22**
- Declarar variável pública: **public real pi = 22**
- Usar definição externa: **require real pi**

Inicialização

Quando existe, é uma expressão que segue o sinal **=** ("igual"): inteira, real, ponteiro. Entidades reais podem ser inicializadas por expressões inteiras (conversão implícita). A expressão de inicialização deve ser um literal se a variável for global.
As cadeias de caracteres são (possivelmente) inicializadas com uma lista não nula de valores sem separadores.

Exemplos:

- Inteiro (literal): **int i = 3**
- Inteiro (expressão): **int i = j+1**
- Real (literal): **real r = 3.2**
- Real (expressão): **real r = i - 2.5 + f(3)**
- Cadeia de caracteres (literal): **string s = "olá"**
- Cadeia de caracteres (literais): **string s = "olá" "mãe"**
- Ponteiro (literal): **ptr<ptr<ptr<real>>> p = nullptr**
- Ponteiro (expressão): **ptr<real> p = q + 1**
- Ponteiro (genérico): **ptr<auto> p = q**
- Tuplo (simples): **auto p = 2.1**
- Tuplo (função): **auto a, b, c, d = f(1)**
- Tuplo (sequência): **auto i, j, k = 1, 2, g + 1**

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 sempre designadas por identificadores constantes precedidos do tipo de dados devolvido pela função. Se a função não devolver um valor, é declarada como procedimento, usando-se a palavra chave **procedure** para o indicar.

As funções que recebam argumentos devem indicá-los no cabeçalho. Funções sem argumentos definem um cabeçalho vazio. Não é possível aplicar os qualificadores de exportação/importação **public** ou **require** (ver símbolos globais) 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 (neste caso, não pode utilizar-se a palavra chave **require**).

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. O número e tipo de parâmetros actuais deve ser igual ao número e tipo dos parâmetros formais da função invocada. 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.

Quando o tipo a retornar pela função a chamar não é primitivo (i.e., é um tuplo), o chamador deve reservar uma zona de memória compatível com o tipo de retorno e passá-la por ponteiro à função chamada como o seu primeiro argumento (ver abaixo).

Corpo

O corpo de uma função consiste num bloco que pode ter declarações (opcionais) seguidas de instruções (opcionais). Uma função sem corpo é uma declaração e é considerada indefinida.

Não é possível aplicar as palavras chave **public** ou **require** (ver símbolos globais) dentro do corpo de uma função.

Uma instrução **return** causa a interrupção imediata da função, assim como o retorno dos valores indicados como argumento da instrução. Os tipos destes valores têm de concordar com o tipo declarado. Quando o tipo a retornar não é primitivo (i.e., é um tuplo), os valores são copiados para a zona de memória reservada pelo chamador e passada por ponteiro à função como o seu primeiro argumento (ver acima).

É um erro especificar um valor de retorno para procedimentos.

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

Função principal e execução de programas

Um programa inicia-se com a invocação da função **og** (sem argumentos). Os argumentos com que o programa foi chamado podem ser obtidos através das seguintes funções:

- **int argc()** (devolve o número de argumentos);
- **string argv(int n)** (devolve o n-ésimo argumento como uma cadeia de caracteres) (**n>0**); e
- **string envp(int 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 retornar 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). Não é possível declarar ou 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 idêntico ao da instrução **if-else** em C.

Instrução de iteração

Esta instrução tem comportamento idêntico ao da instrução **for** em C. Na zona de declaração de variáveis, apenas pode ser usada uma declaração **auto**, devendo ser, nesse caso, a única.

Instrução de terminação

A instrução **break** termina o ciclo mais interior em que a instrução se encontrar, 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 **continue** reinicia o ciclo mais interior em que a instrução se encontrar, 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 avaliadas, mesmo que não produzam efeitos secundários. A notação é como indicada na gramática (expressão seguida de **;**).

Instruções de impressão

As palavras chave **write** e **writeln** podem ser utilizadas para apresentar valores na saída do programa. A primeira apresenta a expressão sem mudar de linha; a segunda apresenta a expressão mudando de linha. 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	Assoc	Operandos	Semântica
Primária	( ) [ ]	N	-	Parênteses curvos, indexação, reserva de memória
Unária	+ - ?	N	-	Identidade e simétrico, indicação de posição
Multiplicativa	* / %	E->D	int reais	C (% é apenas para inteiros)
Aditiva	+ -	E->D	int 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	< > <= >=	E->D	int reais	C
Igualdade	== !=	E->D	int reais ponteiros	C
“não” lógico	~	N	int	C
“e” lógico	&&	E->D	int	C: o 2º argumento só é avaliado se o 1º não for falso.
"ou" lógico	||	E->D	int	C: o 2º argumento só é avaliado se o 1º não for verdadeiro.
Atribuição	=	D->E	não tuplos	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.

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 reservada **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 várias tipos (**write** ou **writeln**), deve ser lido um inteiro.

Exemplos: **a = input** (leitura para **a**), **f(input)** (leitura para argumento de função), **write input** (leitura e impressão).

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).

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 de uma expressão ponteiro seguida do índice entre parênteses rectos. O resultado de uma indexação de ponteiros é um *left-value*.

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

Indexação de tuplos

A indexação de tuplos devolve o valor de uma posição de um tuplo indicada por um número de ordem (início em 1). Consiste de um tuplo seguido de um literal inteiro que indica a posição. O resultado de uma indexação de tuplos é um *left-value*.

Exemplo (acesso à segunda posição do tuplo **a**): **a@2**

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**): **ptr<real>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** aplica-se a expressões, retornando a dimensão correspondente em bytes. Aplicado a um tuplo, retorna a soma das dimensões dos seus componentes.

Exemplos: **sizeof(a)** (dimensão de **a**); **sizeof(1, 2)** (8 bytes).

Exemplos

Programa com vários módulos

Definição da função *factorial* num ficheiro (**factorial.og**):

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

Exemplo da utilização da função *factorial* num outro ficheiro (**main.og**):

```
// external builtin functions
require int argc()
require string argv(int n)
require int atoi(string s)

// external user functions
require int factorial(int n)

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

Como compilar:

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