Linguagem Minor

Manual de Referência

(10 de Fevereiro de 2020)

1 - Introdução

Este manual de referência descreve a linguagem de programação **minor**. Embora procurando ser o mais preciso possível, utiliza-se português para descrever a linguagem. Desta forma o documento torna-se mais intuitivo, mas menos rigoroso, que uma descrição formal.
A linguagem **minor** é uma linguagem imperativa, não estruturada, e fortemente tipificada.

1.1 - Tipos de dados:

A linguagem define 3 tipos de dados:

1.1.1 - números inteiros:

designados por **number**, representam números inteiros positivos, negativos ou nulos, ocupando 4 bytes em complemento para dois, alinhados a 32 bits.

1.1.2 - cadeias de carateres:

designadas por **string**, representam ponteiros para sequências de carateres UTF-8, terminadas pelo caráter com o valor 0 ASCII (NULL). As variáveis e literais do tipo **string** podem apenas ser utilizados em atribuições, em impressões e como argumentos de funções.

1.1.3 - vetores inteiros:

designados por **array**, representam ponteiros para conjuntos de números inteiros sequencialmente colocados em memória.

1.2 - Verificação de tipos:

Na linguagem **minor** as operações dependem dos tipos de dados a que são aplicadas. Os tipos suportados por cada operador e a operação a realizar são indicados na definição das expressões (ver seção 5).

1.3 - Manipulação de nomes:

Os nomes são usados para designar as entidades de um programa em **minor**, ou seja, variáveis ou funções.

1.3.1 - espaços de nomes:

o espaço de nomes global é único, pelo que um nome utilizado para designar uma variável não pode ser utilizado para designar uma função.

1.3.2 - alcance das variáveis:

as variáveis globais (declaradas fora de qualquer função), e as restantes entidades, existem do início ao fim da execução do programa. As variáveis locais a uma função existem apenas durante a execução desta, e os argumentos formais estão válidos enquanto a função está ativa.

1.3.3 - visibilidade dos identificadores:

os identificadores estão visíveis desde a sua declaração até ao fim do seu alcance, ficheiro (globais) ou à função (locais). A redeclaração de um mesmo identificador numa função cria uma nova variável que encobre a global até ao fim da respetiva função. Uma função não pode declarar no seu corpo principal identificadores com mesmo nome dos seus argumentos formais.

2 - Convenções lexicais

A linguagem de programação **minor** é constituída por seis grupos de elementos lexicais (tokens), devendo o elemento lexical ser constituído pela maior sequência de carateres que constitua um elemento lexical válido:

2.1 - Carateres brancos:

são considerados carateres brancos aqueles que, embora servindo para separar os elementos lexicais, não representam nenhum elemento lexical. São considerados carateres brancos: **espaço** ASCII SP (0x20, ou ' '), **mudança de linha** ASCII LF (0x0A, ou '\n'), **recuo do carreto** ASCII CR (0x0D, '\r') e **tabulação horizontal** ASCII HT (0x9, ou '\t').

2.2 - Comentários:

Os comentários funcionam como separadores de elementos lexicais. Existem dois tipos de comentários:

2.2.1 - explicativos:

começam pela sequência ('$$'), desde que a sequência não faça parte de uma cadeia de carateres, e acabam no fim da linha.

2.2.2 - operacionais:

são iniciados por ('$') e terminam com ('$') quando separados por outros carateres, desde que não façam parte de cadeias de carateres.

2.3 - Identificadores:

por vezes designados por nomes , são constituídos por uma letra (maiúscula ou minúscula) seguida por 0 (zero) ou mais letras, dígitos decimais ou o caráter ('_'). O número de carateres que constituem um identificador é ilimitado e dois nomes designam identificadores distintos se houver alteração de maiúscula para minúscula, ou vice-versa, de pelo menos um caráter.

2.4 - Palavras chave:

os identificadores indicados na tabela abaixo estão reservados pela linguagem **minor**, são palavras chave da linguagem, e não podem ser utilizados como identificadores normais. Estes identificadores têm de ser escritos exatamente como na lista abaixo: program, module, start, end, void, const, number, array, string, function, public, forward, if, then, else, elif, fi, for, until, step, do, done, repeat, stop, return.

2.5 - Literais:

são notações para elementos cujo valor é diretamente apresentado. (Notar que constantes são identificadores que designam elementos cujo valor não pode sofrer alterações durante a execução do programa.)

2.5.1 - inteiros:

um literal inteiro é um número positivo. (Notar que os números negativos são construídos pela aplicação do operador simétrico (-) a um literal positivo.)
Um literal inteiro em decimal é constituído por uma sequência de 1 (um) ou mais dígitos decimais (dígitos de '0' a '9') em que o primeiro dígito não é um '0' (zero), excepto no caso do número 0 (zero) composto apenas pelo dígito '0' (zero), que é igual qualquer que seja a base de numeração. Um literal inteiro em octal começa sempre pelo dígito '0' (zero), sendo seguido de um ou mais digitos de '0' a '7'. 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 respetivamente. Um literal inteiro em binário começa sempre pela sequência '0b', sendo seguido de um ou mais digitos '0' ou '1'. Se não for possível representar o literal inteiro na máquina, devido a um *overflow*, deverá ser gerado um erro lexical.

2.5.2 - cadeia de carateres:

é consituída por dois ou mais iniciadores seguidos, sem separadores, mas constituindo elementos lexicais distintos, bem como uma só cadeia de texto. Os

iniciadores podem ser valores inteiros não negativos, carateres individuais ou cadeias de texto. Os valores inteiros, em qualquer base, representam um só caráter, devendo o valor ser truncado caso exceda 255. Os carateres individuais começam e terminam com o caráter plica (') podendo conter um só caráter ou uma sequência especial iniciada por (\). Uma sequência especial pode ser representada pelos carateres ASCII LF, CR e HT ('\n', '\r' e '\t', respetivamente), ou ainda a plica ('\'') ou a barra ('\\'). Qualquer outro caráter pode ser representado por 1 ou 2 digitos hexadecimais precedidos do caráter '\', por exemplo '\0a' ou apenas '\A'. Uma cadeia de texto, começa e termina com o caráter aspa ("), pode conter qualquer número de carateres, podendo estes ser carateres ASCII ou carateres UTF-8 para carateres portugueses, por exemplo. Os carateres utilizados para iniciar ou terminar comentários ('$') têm o seu valor normal ASCII não iniciando ou terminando qualquer comentário. As mesmas sequências especiais dos carateres individuais podem ser utilizados nas cadeia de texto, excepto o (\') que é substituído pela aspa (\") O caráter aspa (") pode ser utilizado desde que precedido de (\).

2.6 - Operadores de expressões:

são considerados operadores da linguagem **minor** os seguintes elementos lexicais,

- + * / % ^ := < > = >= <= ~= | & ~ ?

2.7 - Delimitadores e separadores:

Os elementos lexicais seguintes são considerados delimitadores da linguagem **minor**: '#', '[', ']', '(', ')', ';', '!' e ',' (vírgula).

2.8 - Zona de código:

Um programa em **minor** é representado pela parte de um ficheiro desde as palavras reservadas **program** ou **module** à palavra reservada **end**. Estas três palavras reservadas têm de aparecer no início da linha, caso contrario não têm o significado descrito, podendo ser usadas como identificadores. Assim, todo o conteúdo do ficheiro anterior às palavras reservadas **program** ou **module** e após a palavra reservada **end** deverá ser ignorado (quando no início da linha).

3 - Gramática

3.1 - Gramática:

A gramática da linguagem **minor** pode ser resumida pelas regras abaixo. Considerou-se que os elementos a negrito são literais, que os parênteses curvos agrupam elementos, que elementos alternativos são separados por uma barra vertical, que elemento opcionais estão entre parênteses retos, que os elementos que se repetem zero ou mais vezes estão entre chavetas.

Cuidado que a barra vertical, os parênteses e as chavetas são elementos lexicais da linguagem quando representados a negrito.

ficheiro	=	programa | módulo
programa	=	**program** [declaração { **;** declaração }] **start** corpo **end**
módulo	=	**module** [declaração { **;** declaração }] **end**
declaração	=	função | [qualificador] [**const**] variável [**:=** literais]
função	=	**function** [qualificador] (tipo | **void**) ident [variável { **;** variável }] (**done** | **do** corpo)
variável	=	tipo ident [**[** inteiro **]**]
tipo	=	**number** | **string** | **array**
qualificador	=	**public** | **forward**
corpo	=	{ variável **;** } { instrução }
literal	=	inteiro | caráter | cadeia
literais	=	literal { literal } | literal { **,** literal }
instrução	=	**if** expressão **then** { instrução } { **elif** expressão **then** { instrução } } [**else** { instrução }] **fi** | **for** expressão **until** expressão **step** expressão **do** { instrução } **done** | expressão (**;** | **!**) | **repeat** | **stop** | **return** [expressão] | left-value **#** expressão **;**

3.1.1 - elementos lexicais:

foram omitidos da gramática, por já terem sido definidos acima, os seguintes elementos: ident: definido em 2.3, inteiro: definido em 2.5.1, caráter: definido em 2.5.2, cadeia: definido em 2.5.2.

3.1.2 - left-value:

os elementos de uma expressão que podem ser utilizados como left-values encontram-se individualmente identificados na seção 6. Todos os left-value representam posições de memória passíveis de ser modificadas, podendo igualmente ser usadas como right-values em expressões através da leitura da respetiva posição de memória.

3.1.2 - expressão:

foi delibradamente omitida da gramática e será tratada em na seção 6.

3.2 - Ficheiros:

Um ficheiro descrito em **minor** pode ser de dois tipos:

3.2.1 - programas:

identificados no início do ficheiro pela palavra reservada **program**, na 1ª coluna. Os programas têm, obrigatoriamente um corpo de instruções que representa o ponto de entrada no programa, gerando uma função designada por _main.

3.2.2 - módulos:

identificados no início do ficheiro pela palavra reservada **module**, na 1ª coluna. Os módulos não contêm um ponto de entrada, pelo que devem ser ligados com um (e um só) programa para gerar um ficheiro executável, podendo conter outros módulos.

3.3 - Declaração de variáveis e constantes:

cada declaração permite declarar uma única variável ou constante. Uma declaração inclui os seguintes componentes:

3.3.1 - constante:

designada pela palavra reservada **const**, que torna o identificador constante, ou seja, cujo valor representado não pode ser modificado. Tal não significa que os valores indiretamente referidos (apontados) pelo identidicador sejam constantes, por exemplo variáveis do tipo **string** ou **array**.
Notar que declarações de constantes não iniciadas só são possíveis se forem identificadores pré-declarados, pertencentes a outros ficheiros (módulos ou programa).

3.3.2 - Símbolos globais:

Para a utilização de compilação separada em **minor** existem dois qualificadores opcionais nas declarações:

3.3.2.1 - publico:

designado pela palavra reservada **public** torna o identificador globalmente visível fora deste módulo (ou programa).

3.3.2.2 - pré-declaração:

designado pela palavra reservada **forward** identifica a entidade (constante, variável ou função) não iniciada, podendo esta estar declarada mais à frente no ficheiro ou em outro ficheiro (possivelmente noutra linguagem).

3.3.3 - tipo de dados:

designado por um dos 3 tipos de dados (**number**, **string** ou **array**).

3.3.4 - identificador:

designada por um identificador que passa a nomear a entidade declarada. No caso declarações do tipo **array**, o identificador pode ser seguido de um inteiro positivo delimitado por parenteses retos, que corresponde à dimensão total do vetor.

3.3.5 - inicialização:

a existir, inicia-se com o operador de atribuição ':=' seguido de valores literais dependentes do tipo declarado:

3.3.5.1 - inteiros:

o valor a iniciar é um número inteiro, representado em binário, decimal, octal ou hexadecimal.

3.3.5.2 - cadeia de carateres:

uma lista não nula de valores separados por carateres brancos. Os valores podem ser inteiros (ver 3.3.5.1), carateres individuais ou cadeias de texto. Estes valores são sempre constantes, independentemente de o identificador que as designa ser constante ou não.

3.3.5.3 - vetores de inteiros:

uma lista não nula de valores inteiros separados por vírgulas.

4 - Funções

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

4.1 - Declaração :

As funções em **minor** são sempre designadas por identificadores constantes precedidas da palavra chave **function**, de um qualificador opcional, e do tipo de dados devolvido pela função. As funções, que recebam argumentos, devem indicá-los no cabeçalho, separados por vígulas, após o identificador da função. Cada argumento é descrito por um tipo seguido do identificador. Uma declaração de uma função não iniciada, termina com a palavra reservada **done** e, é utilizada para tipificar um identificador exterior ou para efetuar *forward declarations* (utilizada para pré-declarar uma função que seja usada antes de ser definida, por exemplo, entre duas funções mutuamente recursivas). Caso a declaração seja iniciada, através da palavra reservada **do** seguido de um corpo, define-se uma nova função.
O valor devolvido por uma função, através da instrução **return**, deve ser sempre do tipo declarado. Todas as funções terminam com a instrução **return**, exceto as que não devolvam um valor (tipo **void**). Por outro lado, a função principal de um programa nunca termina com uma

instrução **return**, caso a última instrução seja executada a função principal deve devolver o inteiro **0** (zero) ao sistema operativo. Contudo, a instrução **return** pode ser utilizada nos sub-blocos (em **if**s ou **for**s) da função principal.
A função não pode definir variáveis designadas por identificadores com o mesmo nome de nenhum dos argumentos formais da função.

4.2 - Invocação :

A função só pode ser invocada através de um identificador que refira uma função previamente declarada ou definida. Se a função não tiver argumentos não devem ser utilizados parenteses na sua invocação.
Caso existam argumentos, na invocação da função o seu identificador é seguido de uma lista de expressões delimitadas por parenteses curvos. A lista de expressões é uma sequência, não nula, de expressões separadas por vírgulas. O valor resultante passado por cópia (passagem de argumentos por valor).
O número e tipo de parâmetros atuais deve ser igual ao número e tipo dos parâmetros formais da função invocada. A ordem dos parâmetros atuais deverá ser a mesma dos argumentos formais da função a ser invocada. Os parâmetros atuais 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.
Quem coloca os argumentos na pilha (a função chamadora) também é responsável pela sua remoção após o retorno da função chamada (chamada *à* C). Os valores de retorno são devolvidos no acumulador do processador.

4.3 - Corpo :

um sub-bloco de função (usada numa instrução condicional ou de iteração) não pode definir variáveis.

5 - Instruções

Excepto quando indicado as instruções são executadas em sequência, sendo os seus efeitos traduzidos, em geral, pela alteração do valor de variáveis.

5.1 - Instrução condicional:

a instrução é delimitada pela palavras reservadas **if** e **fi**. Se a expressão que segue a palavra reservada **if** for diferente de 0 (zero) então o corpo que lhe segue é executado.
Caso existam conjuntos **elif** as suas expressões serão sucessivamente executadas, caso todas as condições anteriores sejam nulas (iguais a 0). Assim que uma dessas condições seja diferente de 0 (zero), o seu corpo é executado e a instrução termina.
Caso exista um conjunto **else**, o seu corpo só é executado se nenhum dos anteriores corpos da instrção o tenha sido.

5.2 - Instrução de iteração:

a instrução é delimitada pela palavras reservadas **for** e **done**. A palavra reservada **for** é seguida de uma expressão de inicialização. As instruções do ciclo serão executadas enquanto a expressão que segue a palavra reservada **until** for igual a zero (falsa). Após as instruções do ciclo serem todas executadas, a expressão que segue a palavra reservada **step** é executada antes de testar a condição **until**. As instruções do ciclo aparecem após a palavra reservada **do**.

5.3 - Instrução de retorno:

iniciada pela palavra reservada **return**, a existir deverá ser a última instrução do bloco em que se insere. Esta instrução termina a execução da função, mesmo que se encontre num corpo que não seja o bloco principal. A instrução só é seguida de um expressão caso a função não seja do tipo **void**.

5.4 - Instrução de continuação:

constituída pela palavra reservada **repeat**, a existir deverá ser a última instrução do bloco em que se insere. Esta instrução reinicia o ciclo **for** mais interior em que a instrução **repeat** se encontra, tal como a instrução **continue** em **C**. Esta instrução só pode existir dentro de um ciclo.

5.5 - Instrução de terminação:

constituída pela palavra reservada **stop**, a existir deverá ser a última instrução do bloco em que se insere. Esta instrução termina a execução do ciclo **for** mais interior em que a instrução **stop** se encontra, tal como a instrução **break** em **C**. Esta instrução só pode existir dentro de um ciclo.

5.6 - Expressão como instrução:

qualquer expressão pode ser utilizada como instrução, mesmo que não produza qualquer efeito secundário. Caso a expressão seja terminada por **!** o seu valor deve ser impresso. O valor deve ser impresso em decimal para os valores do tipo **number** ou **array**, em ASCII e UTF-8 para valores do tipo **string**.

5.7 - Reserva de memória na pilha

A instrução de reserva de memória na pilha utiliza o operador **#** e permite atribuir ao ponteiro, *left-value* da esquerda, o início da zona de memória com comprimento igual ao número de unidades do *right-value* indicado na expressão da direita.

6 - Expressões

Uma expressão é uma representação algébrica de uma quantidade. Assim, todas as expressões devolvem um valor. As expressões na linguagem de programação **minor** utilizam operadores algébricos comuns: soma, subtração, multiplicação e divisão inteira e resto da divisão, além de outros operadores.

As expressões são sempre avaliadas da esquerda para a direita, independentemente da associatividade do operador. A precedência dos operadores é a mesma para operadores na mesma seção, sendo as seções seguintes de menor prioridade que as anteriores. O valor resultante da aplicação da expressão bem como a sua associatividade são descriminados em cada operador.
A tabela seguinte que resume os operadores da linguagem **minor**, por grupos de precedência decrescente:

designação	**operadores**	**associatividade**
primária	( ) [ ]	não associativos
unária	& - ?	não associativos
potência	^	da direita para a esquerda
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

Os operadores têm o mesmo significado que em C, com a excepção do operador de diferença (que usa ~= em vez de !=), o operador de igualdade (que usa = em vez de ==), o operador de atribuição (que usa := em vez de =), além do operador de leitura (que usa ?) e do operador de potência (que usa ^), como noutras linguagens de programação.

6.1 - Expressões primárias:

6.1.1 - identificadores:

Um identificador é uma expressão desde que tenha sido devidamente declarado. Um identificador pode denotar uma variável, uma constante ou uma função. Neste último caso o identificador representa a invocação de uma função sem argumentos.
Um identificador é o caso mais simples de um *left-value*, ou seja, uma entidade que pode ser utilizada no lado esquerdo (*left-value*) de uma atribuição.

6.1.2 - literais:

Os literais podem ser valores constantes inteiros não negativos, tal como definidos nas convenções lexicais, ou cadeias de carateres.

6.1.3 - parenteses curvos:

Uma expressão entre parenteses curvos tem o valor da expressão sem os parenteses e permite alterar a prioridade dos operadores. Uma expressão entre parenteses não pode ser utilizada como *left-value* (ver em indexação).

6.1.4 - indexação:

Uma expressão indexação devolve um valor inteiro que referencia um *left-value* através da sua localização em memória, somado ao deslocamento da expressão

entre parenteses retos que representa o número de unidades. O resultado de uma expressão de indexação é o valor existente na posição de memória indicada pelo identificador. Uma indexação também pode ser utilizada como *left-value*.

6.1.5 - invocação:

A função só pode ser invocada através de um identificador que refira uma função previamente declarada ou definida. A invocação de uma função sem argumentos refere apenas o seu identificador. Para funções com argumentos, estes são expressões separadas por vírgulas e delimitadas por parêntesis curvos.

6.1.6 - leitura:

A operação de leitura de um valor inteiro pode ser efetuado pela expressão **?**, que devolve o valor lido.

6.2 - Expressões unárias:

6.2.1 - localização:

A expressão localização **&** devolve um valor inteiro di tipo array que representa a posição de memória ocupada pelo *left-value* e utiliza o operador prefixado **&**.

6.2.2 - simétrico:

A expressão valor simétrico **-** devolve o simétrico do valor inteiro e utiliza o operador prefixado **-**.

6.3 - Expressão de potência:

a operação **^** é apenas aplicável a valores inteiros, devolvendo a multiplicação do primeiro argumento por ele próprio tantas vezes quantas o valor do segundo argumento.

6.4 - Expressões multiplicativas:

as operações *, / e **%** são apenas aplicáveis a valores inteiros, devolvendo o resultado da respetiva operação algébrica.

6.5 - Expressões aditivas:

as operações **+** e **-** são apenas aplicáveis a valores inteiros, ou quando um deles é inteiro e o outro é vetor (aritmética de ponteiros), devolvendo o resultado da respetiva operação algébrica. É possível fazer a diferença entre dois vetores, resultando num inteiro.

6.6 - Expressões de grandeza:

as operações **<**, **>**, **<=** e **>=** são aplicáveis quando ambos os valores são inteiros, ou ambos cadeias, devolvendo o valor inteiro 0 (zero) caso seja falsa e um valor diferente de zero caso contrário.

6.7 - Expressões de igualdade:

as operações **~=** e **=** são aplicáveis quando ambos os valores são inteiros, ou ambos cadeias, devolvendo o valor inteiro 0 (zero) caso seja falsa e um valor diferente de zero caso contrário.

6.8 - Expressões de negação lógica:

a operação **~** é aplicável a valores inteiros, devolvendo o valor inteiro 0 (zero) caso o argumento seja diferente de 0 (zero) e um valor diferente de zero caso contrário.

6.9 - Expressões de junção lógica:

a operação **&** é aplicável a valores inteiros, devolvendo um valor diferente de zero caso ambos os argumentos sejam diferente de 0 (zero) e um valor diferente de zero caso contrário. Caso o primeiro argumento seja 0 (zero) o segundo argumento não deve ser avaliado.

6.10 - Expressões de alternativa lógica:

a operação **|** é aplicável a valores inteiros, devolvendo o valor inteiro 0 (zero) caso ambos os argumentos sejam iguais a 0 (zero) e um valor diferente de zero caso contrário. Caso o primeiro argumento seja um valor diferente de zero o segundo argumento não deve ser avaliado.

6.11 - Expressões de atribuição:

O valor da expressão do lado direito do operador **:=** é guardado na posição indicada pelo *left-value* do lado direito do operador de atribuição. Só podem atribuídos valores a variáveis do mesmo tipo. O valor **0** (zero) pode ser atribuído a qualquer tipo de variável, correspondendo ao ponteiro nulo no caso de cadeias e vetores.

7 - Interface com o sistema operativo

7.1 - Função principal:

A execução de um programa em **minor** inicia-se com a invocação da função que realiza o programa, designado por program, e que no seu término deverá retornar 0 (zero).

7.2 - rotinas de biblioteca:

O ficheiro **linux.asm** contém a rotina de arranque (_start) que invoca a função principal para os programas desenvolvidos em **minor**, bem como a rotina de terminação (_exit). O ficheiro **lib.asm** contém um conjunto de rotinas de biblioteca que poderá utilizar, com nomes auto-explicativos e semelhantes aos da biblioteca de **C**:

- **void println**: imprime o caráter de mudança de linha ('\n');
- **void printsp(int n)**: imprime 'n' espaços brancos;
- **void prints(const char *s)**: imprime a cadeia de caracteres 's', terminada em NULL ('\0');
- **void printi(int i)**: imprime o valor inteiro 'i', em decimal;
- **char *readln(char *buf, int size)**: equivale a fgets(buf, size, stdin) em **C**;
- **int readb**: lê um byte do terminal;
- **int readi**: lê um inteiro decimal, isolado numa linha, do terminal;
- **int strlen(const char *s)**: equivalente à rotina homónima em **C**;
- **int atoi(const char *s)**: equivalente à rotina homónima em **C**;
- **char *itoa(int i)**: converte um valor inteiro para uma cadeia de caracteres ASCII, terminada em NULL ('\0'), situada num bloco de memória fixo (é reutilizado em chamadas subsequentes), em decimal.

Deverão igualmente ser adicionadas quaisquer rotinas necessárias ao correto funcionamento dos operadores existentes na linguagem.

7.3 - argumentos do programa:

Existem 3 funções que permitem obter os argumentos normalmente obtidos pela função main do **C**:

número de argumentos da linha de comando:

função designada por argc e que devolve um inteiro que representa o número de argumentos indicados na linha de comando, incluindo o nome do programa.

argumentos da linha de comando:

função designada por argv, recebe como argumento um inteiro, entre **0** e **argc**, que representa o número do argumento e devolve a respectiva cadeia de caracteres.

variáveis de ambiente:

função designada por envp, recebe como argumento um inteiro, de **0** até que **envp(N) == 0** (para N >= 0), que representa o número da variável de ambiente e devolve a respectiva cadeia de caracteres.

7.4 - chamadas ao sistema operativo:

O ficheiro **sys.asm** contém as chamadas ao sistema que pode realizar em programas escritos em **minor**. Uma explicação das chamadas ao sistema pode ser obtida através de:

```
prompt$ man 2 intro
prompt$ man 2 syscalls
```

Algumas destas chamadas não existem na biblioteca de **C**, outras, como o **brk**, têm um comportamento diferente da rotina homónima da biblioteca de **C**.

7.5 - rotinas em C:

Como a passagem de parâmetros e retorno de valores é igual ao **C**, podem-se invocar de **minor** rotinas em **C** e vice-versa.

8 - Exemplos

Os exemplos apresentados não são exaustivos, pelo que nem todas as construções da linguagem são utilizadas.

O já habitual hello world:

```
program start
        "olá pessoal!\n"!
end
```

O cálculo da função de Ackermann. (Esta função tem um crescimento muito elevado pelo que nos computadores atuais, mesmo utilizando **C**, os argumentos não deverão exceder m=3 e n=12 para executar em poucos segundos)

```
program
number cnt := 0;
function forward number atoi string s done;
function forward number argc done;
function forward string argv number n done;
function number ackermann number m ; number n do
        cnt := cnt + 1;
        if m = 0 then return n+1 fi
```

```
            if n = 0 then return ackermann(m-1, 1) fi

            return ackermann(m-1, ackermann(m, n-1))

start

            if argc > 2 then

                        ackermann(atoi(argv(1)), atoi(argv(2)))! " #"! cnt! "\n"!

            fi

end
```

Outros exemplos podem ser obtidos a partir da página do projeto no fénix.

9 - Avaliação

Este projeto pretende desenvolver um compilador para a linguagem de programação **minor**. Este compilador deverá efetuar a análise sintática e semântica da linguagem, produzindo como resultado código máquina (*assembly*) com seleção de instruções em formato **nasm** para **linux-elf32-i386**. O compilador deverá ser escrito em **C/C++** com auxílio das ferramentas de análise lexical, sintática e seleção de instruções, respetivamente, flex, byacc e pburg. Nenhuma destas ferramentas deve produzir erros ou avisos na sua invocação. A análise semântica da linguagem deverá garantir que o programa a processar se encontra corretamente escrito e que pode ser gerado código, devendo gerar mensagens de erro explícitas em caso contrário. Se forem encontrados erros (lexicais, sintáticos ou semânticos) não deve ser gerado código e compilador deve devolver um valor não zero. A seleção das instruções deve aproveitar da melhor forma possível as capacidades do processador, as macros **postfix**. Para o cálculo dos custos deve-se considerar que cada instrução **postfix** tem um custo pelo menos unitário (1), excluindo as diretivas *assembly*.

O ficheiro *assembly* gerado deve ser processado pelo *assembler* **nasm**, com o comando nasm -felf32 file.asm, para obter o ficheiro objeto file.o, assumindo que não são encontrados erros no seu processamento. O executável final deve ser construído pelo *loader* **ld** com todos os ficheiros objeto necessários e a biblioteca de *runtime*: ld -m elf_i386 file1.o file2.o -lminor.

O projeto é desenvolvido individualmente. Apenas se consideram para avaliação os projetos submetidos no Fenix, contendo todos os ficheiros fontes necessários à resolução do projeto, em formato .tgz (ou .zip), até à data limite. A submissão de um projeto pressupõe o compromisso de honra que o trabalho incluso foi realizado pelo aluno. A quebra deste compromisso, ou seja a tentativa de um aluno se apropriar de trabalho realizado por outros, tem como consequência a reprovação de todos os alunos envolvidos (incluindo os que possibilitaram a ocorrência) à disciplina neste ano letivo.

Parte da nota do projeto é obtida através de testes automáticos, pelos que estas instruções devem ser seguidas rigorosamente. Caso os testes falhem por causas imputáveis ao aluno a nota refletirá apenas os testes bem sucedidos. As restantes componentes da nota são obtidas pela análise do código entregue e pela avaliação do teste prático. O código é avaliado quanto à sua correção, simplicidade e legibilidade. Para tal, os comentários, nomeadamente, não devem ser excessivos nem óbvios, por forma a dificultar a legibilidade, nem muito escassos, por forma a impedir a compreensão das partes mais complexas do programa.

A entrega intermédia deve incluir os ficheiros de especificação **lex** (minor.l) e **yacc** (minor.y) que realizem a análise semântica e, caso o programa esteja correto, imprime a árvore sintática

abstrata (**AST**) no stdout. Deve incluir uma Makefile no diretório principal, que produza o executável minor no mesmo diretório, e todos os ficheiros auxiliares que forem necessários para gerar o executável. A execução do referido ficheiro executável deverá devolver 0 (zero) se o programa processado não tiver erros ou devolver 1 (um) e produzir mensagens de erro esclarecedoras no terminal (usando o stderr) caso o programa tenha erros lexicais, sintáticos ou semânticos.

O executável **minor** da entrega final não imprime a árvore sintática abstrata, gera código **linux-elf32-i386** no ficheiro de saída e deve incluir o ficheiro de especificação **burg** (minor.brg). A Makefile principal deve produzir, além do executável do compilador **minor**, a biblioteca de *runtime* libminor.a, no mesmo diretório, contendo suporte para todas as operações da linguagem.

10 - Desenvolvimento

A ferramentas, e material de apoio, contêm mecanismos de teste e *debug*. O compilador deve ser desenvolvido em passos de complexidade crescente, não devendo passar ao passo seguinte antes de concluir com sucesso o anterior:

•expressões regulares (definir tokens em minor.y e especificação lexical completa em minor.l):
./minor -initial ex.min
deve imprimir todos os *tokens* presentes no exemplo.

•gramática (definir apenas a gramática em minor.y):
YYDEBUG=1 ./minor ex.min
não deve dar erros nos programas corretos.

•árvore sintática (definir apenas a árvore nas ações em minor.y):
NODEDBUG=1 ./minor ex.min
deve imprimir a árvore nos programas corretos.

•tabela de símbolos (acrescentar manipulação de símbolos nas ações em minor.y):
IDDEBUG=1 ./minor ex.min
deve detetar identificadores não declarados.

•semântica (acrescentar verificações semânticas nas ações em minor.y):
./minor ex.min
deve detetar qualquer erro da linguagem.

•seleção (definir apenas a gramática em minor.brg):
./minor -trace ex.min
não deve dar erros nos programas corretos.
User **NODEDBUG=1 ./minor -trace ex.min** para obter também os custos.

•geração de código (acrescentar ações **postfix** em minor.brg):
./minor ex.min
nasm -felf32 ex.asm
não deve dar erros nos programas corretos e produzir *assembler* correto.

•runtime (introduzir biblioteca de *runtime* em run/):
./minor ex.min
nasm -felf32 ex.asm
ld -m elf_i386 ex.o -lminor
./a.out
programas compilados devem executar como esperado.

No caso da seleção e geração de código pode ser útil começar por processar a árvore de exemplos simples, tipo *hello world*, e depois destes executarem corretamente evoluir para programas sucessivamente mais complexos (árvores maiores e com mais tipos de nós).
Pedro Reis dos Santos
2020-03-19