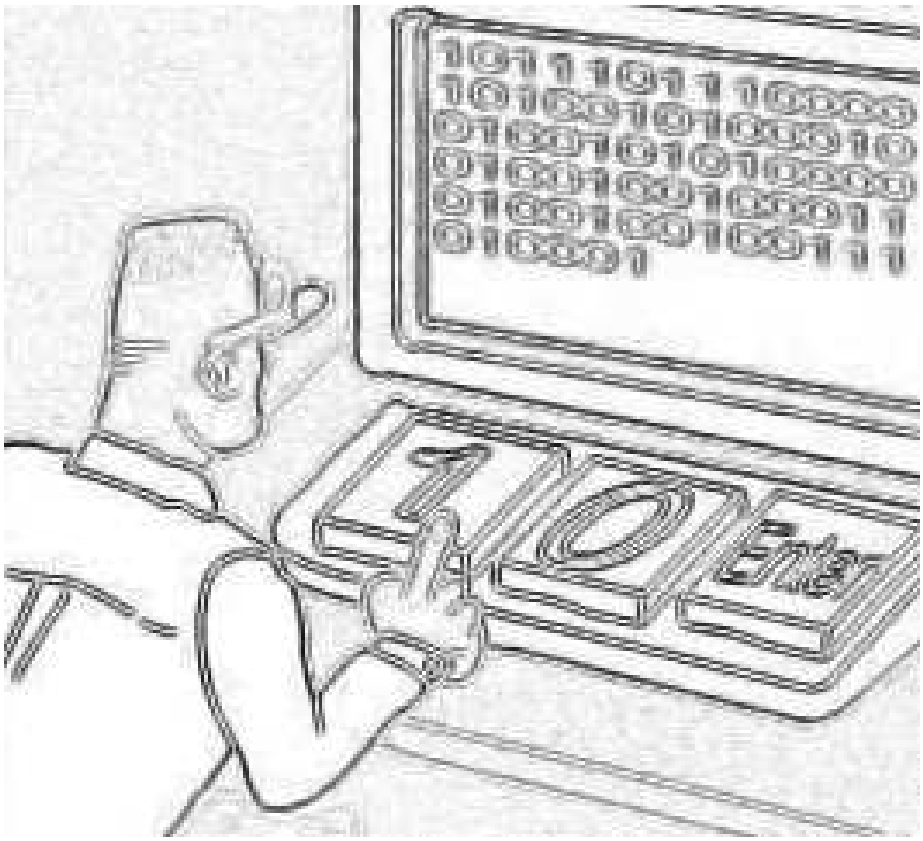
DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

COMPILADORES:
ferramentas de desenvolvimento



Pedro Reis dos Santos

Departamento de Engenharia Informática

5ª Edição
(Fevereiro, 2013)

Texto de apoio para a cadeira de Compiladores do IST-tagus.

Conteúdo

1 Introdução

Este documento pretende oferecer uma introdução às ferramentas de desenvolvimento utilizadas na disciplina de Compiladores. Esta introdução apresenta apenas uma descrição básica de cada ferramenta e não dispensa a consulta dos manuais específicos de cada aplicação.

O ambiente de desenvolvimento utilizado na disciplina de Compiladores é **linux-elf32-i386**, pelo que utiliza o sistema operativo **linux** em processadores **i386** (e derivados **Pentium**, **x86-64**, ...). A linguagem de desenvolvimento do compilador é **C**, embora linguagens como **Java** ou **C++** pudessem ser utilizadas sem grandes alterações uma vez que existem ferramentas equivalentes para estas linguagens. A análise lexical deve ser efectada com recurso à ferramenta **flex** e análise sintáctica com recurso à ferramentas **byacc**. A geração de código deve utilizar as macros **postfix** disponibilizadas, sendo a selecção das instruções a gerar efectuada com recurso à ferramenta **pburg**. A biblioteca de apoio à execução (*runtime library*) pode ser codificada em **C**, em *assembly* **nasm** ou pode ser escrita na linguagem de projecto, se as características desta forem suficientes.

1.1 Linux

Uma vez que o ambiente de avaliação dos projectos é **linux-elf32-i386** aconselha-se a efectuar o desenvolvimento no mesmo ambiente, seja nos computadores existentes no campus ou em computadores pessoais.

Caso ainda não disponha de um ambiente **linux** pode optar entre diversas soluções. Para garantir total compatibilidade com o ambiente de avaliação da disciplina aconselha a utilização de uma partição dedicada em *dual-boot* ou de um *live-CD*. Alternativamente, pode utilizar uma máquina virtual (por exemplo, **virtual-box**) ou um ambiente de emulação (como **cygwin** ou **minGW**), mas neste casos deverá garantir que o projecto funciona correctamente no ambiente de avaliação antes de efectuar a respectiva entrega.

1.2 Dual-boot

A utilização de *dual-boot* permite que dois ou mais sistemas operativos coexistam em zonas distintas do disco computador, podendo ser activados separadamente. Para tal cada sistema operativo ocupará uma ou mais partições de um mesmo disco, ou de vários discos, casos existam.

Assim, terá de sacrificar uma partição do disco (por exemplo a partição `D:`) ou parte desta. Se optar por redimensionar a partição, pode aproveitar o espaço libertado para

criar uma nova partição onde instala o sistema **linux**. A partição pode ser redimensionada com algumas versões da ferramenta **diskmgmt** do **windows**, ou por alguma ferramentas que se podem obter da *internet*. Esta solução permite não destruir o conteúdo da partição, devendo para tal existir espaço livre na partição. Se não for possível redimensionar a partição, esta terá de ser apagada, perdendo-se toda a informação nela contida. Uma vez apagada, pode optar por utilizar todo o espaço para o sistema **linux** ou criar uma partição mais pequena, deixando o espaço remanescente para o **linux**. A instalação do **linux** procurará algum espaço disponível para efectuar a sua instalação.

Uma solução alternativa consiste em utilizar um cartão de memória ou *pen* USB para instalar o sistema **linux**, desde que o computador permita efectuar o arranque a partir de um destes periféricos. A escolha do periférico de arranque pode ser efectuada premindo uma tecla, específica da BIOS de cada computador, no momento em que este é ligado e antes de começar a ser carregado o sistema operativo utilizado por omissão, em geral **windows**. O menu de escolha de dispositivo de arranque, tal como a BIOS do computador, permitem identificar as opções disponíveis em cada caso.

A instalação do sistema **linux** propriamente dito pode ser efectuado a partir de um *live-CD* ou, em alguns casos, directamente a partir do próprio **windows**.

1.3 Live-CD

Existem diversas opções de *live-CD*, como por exemplo o **knoppix** ou **ubunto**. Um *live-CD* consiste no sistema operativo e um conjunto de ferramentas gravadas num único CD-ROM (ou DVD/*blueray*) com a informação comprimida. Para salvaguardar os ficheiros da disciplina poderá criar uma área com cerca de 10MB numa partição de outro sistema operativo (em geral, windows).

Inicialmente terá de descarregar da rede um ficheiro **.iso** contendo o **live-CD**. Para basta pesquisar na rede utilizando palavras como `live linux`. Utilizando o gravador de CD/DVD grava o ficheiro no disco como uma única partição. Para utilizar o sistema basta iniciar o computador a partir do CD-ROM.

1.4 Máquina virtual

A máquina virtual permite a um sistema operativo executar um outro sistema operativo como se de um processo seu se tratasse. Para tal deverá instalar a aplicação de gestão da máquina virtual, por exemplo a **virtual-box**. As máquinas virtuais permitem, em geral, utilizar um ficheiro **.iso** de um *live-CD* ou criar um ficheiro onde guardam o sistema operativo, como se de uma partição se tratasse.

Notar que o ambiente subjacente é ainda aquele em que a máquina virtual executa. Assim, uma máquina virtual **linux** a executar em **windows** ainda exibe algumas particularidades do sistema operativo real (**windows**). Nomeadamente, quando são mapeadas novas páginas de memória em **linux**, estas são iniciadas com todos os bits a `0` (zero) tal como em **windows**, pois na realidade que efectua o mapeamento é o o **windows** e não o **linux**. O resultado é que quaisquer variáveis que apareçam nestas páginas ficam iniciadas a zero. No entanto, num sistema **linux** real as páginas são mapeadas com o *lixo* deixado pelo processo que anteriormente as utilizou. Desta forma, dificilmente a variável será iniciada a zero. Assim, se o programador se esquecer de iniciar uma variável local de uma rotina, em **windows** ou numa virtual de **linux** a executar em **windows** a variável é sempre iniciada em zero. Para ter o mesmo comportamento em **linux** real, o programador terá de alterar o código fonte e colocar o **= 0** na respectiva declaração (o que não poderá fazer após a entrega do projecto). Assim, o projecto deverá ser completamente testado num ambiente **linux** puro antes da respectiva entrega.

1.5 Emulação

A emulação consiste em utilizar ferramentas de **linux** compiladas especificamente para executar em **windows**. No caso do projecto da disciplina de Compiladores tal corresponde ao compilador de **C gcc** e respectivas ferramentas de apoio **gas**, **ld**, **ar**, *etc*. Para o projecto necessita ainda dos compiladores de compiladores **flex**, **byacc**, **pburg** e do *assembler* **nasm**.

Os principais emuladores disponíveis são o **cygwin** e o **minGW**. Notar que o sistema operativo que executa é apenas o **windows**, pelo que qualquer diferença entre este sistema e o sistema **linux** podem comprometer os resultados da avaliação do projecto. Tal como no caso da máquinas virtuais, o projecto deverá ser completamente testado num ambiente **linux** puro antes da respectiva entrega.

1.6 linux 64-bits (x86-64

A utilização de sistemas de 64-bits coloca dois problemas, a geração do compilador e a execução dos exemplos produzidos pelo compilador. Ao produzir o compilador em **linux 64-bits** com o **gcc**, os inteiros (**int**) continuam a ter 32-bits enquanto os ponteiros e os inteiros longos (**long**) passam a ter 64-bits. Notar que apenas os **long long**) têm sempre 64-bits em **gcc**. Assim, desde se tenha o cuidado de não misturar os dois tipos não se antecipam problemas, mas deve ter especial cuidado com os avisos do compilador.

O código gerado pelo compilador a realizar no projecto da disciplina de Compiladores é **i386**, ou seja código de 32-bits. No entanto, o ambiente **linux** de 64-bits (**x86-64**) permite

executar directamente código de 32-bits (**i386**). O compilador realizado no projecto, ao executar, gera código de 32-bits pelo que o **assembler** deverá gerar instruções de 32-bits, ou seja **nasm -elf32**. A principal diferença consiste na geração dos executáveis dos programas de teste, onde o carregador **ld** terá de ser instruído para gerar código de 32-bits, em geral **ld -m elf_i386**.

Contudo, aconselha-se que antes da entrega do projecto este seja completamente testado num sistema **linux** de 32-bits, por exemplo utilizando uma máquina virtual para executar um linux de 32-bits dentro de um linux de 64-bits.

1.7 Android

O sistema operativo **android** e outros sistemas operativo baseados no **linux** utilizam o mesmo núcleo do sistema **linux**, pelo que o comportamento do sistema operativo é idêntico. No entanto, é necessário garantir que as ferramentas a utilizar não foram adulteradas, ou seja são as mesmas versões que executam no **linux** normal. As principais diferenças que se podem notar ao nível do projecto residem essencialmente ao facto de o processador subjacente ser um processador **arm**. Este facto pode induzir comportamentos diferentes se o programa contiver erros (programas correctos executam de igual forma em todos os sistemas operativos e ambientes), nomeadamente na manipulação de ponteiros ou invocação de rotinas.

O facto de se tratar de um processador **arm** implica que não pode executar o código produzido pelo compilador a desenvolver, uma vez que os projectos deverão produzir código **i386**. Assim, pode-se desenvolver o compilador e executá-lo para produzir o código do exemplos, mas não se pode testar os exemplos propriamente ditos. O sistema **android** pode ser utilizado como base de trabalho, mas as fases finais de teste terão de ser efectuadas num ambiente **linux** puro, excepto se se utilizar **postfix** para **arm**. Caso utilize a opção de compilação **-DpfARM**, o código gerado pelo compilador passa a ser **arm** em vez **i386**. Neste caso, o código dos exemplos pode ser produzido com auxílio das ferramentas **as**, em vez do **nasm**, e do **ld**. O resultado pode agora ser executado pelo próprio processador **arm** do **android**. Notar que para a submissão do projecto o compilador deverá produzir código **i386**. No entanto, podem sempre existir diferenças de comportamento subtis entre os dois tipos de processadores (**i386** e **arm**) que não sejam completamente tratadas pelas macros **postfix**. Chama-se especial atenção à invocação de rotinas de bibliotecas gerais com mais de um argumento, uma vez que a regra em processadores **arm** consiste em passar até 4 argumentos nos registos, enquanto o **postfix** utiliza sempre a pilha.

1.8 Mac OS X

O sistema operativo **Mac OS X** é **unix** mas não é **linux**. Assim, embora o comportamento seja mais próximo do ambiente de avaliação que no caso do **windows** existem diferenças significativas. A mais importante reside no facto de os ficheiro objectos terem um formato próprio, designados por **fat binaries**. Assim, mesmo quando o processador é um **i386**, tal como no caso do **android**, não é possível testar os exemplos produzidos pelo compilador do projecto. Consequentemente, as fases finais de teste terão de ser efectuadas num ambiente **linux** puro.

1.9 Windows

O ambiente de avaliação utilizado na disciplina de Compiladores é **linux**, no entanto as linguagens de programação e ferramentas utilizadas (flex, byacc, nasm, ld/link, gcc/lcc/...) também existem no ambiente **windows**. Assim, numa fase inicial de desenvolvimento, as ferramentas de desenvolvimento podem ser utilizadas no ambiente **windows** em processadores **i386**.

Para o desenvolvimento do compilador, que constitui o projecto, aconselha-se a utilização de um compilador de **C** que gere ficheiros objecto em formato **PE**, como por exemplo o **Visual C++** ou **lcc**. O **lcc** tem a vantagem de ter distribuição livre (GPL), embora seja menos sofisticado que o **Visual C++**.

No desenvolvimento dos programas de teste produzidos pelo compilador do projecto basta utilizar o formato **win32** (em vez do **elf** do UNIX) e efectuar a ligação com o **link** (em vez do **ld**), por exemplo:

```
C:\comp> edit test.cpt
C:\comp> compact test.cpt
C:\comp> nasm -fwin32 test.asm
C:\comp> link test.obj win.obj lib.obj /subsystem:console kernel32.lib
C:\comp> test
```

onde **compact** é o compilador de exemplo distribuído, tal como os ficheiros: nasmw.exe, link.exe, win.asm, lib.asm e kernel32.lib.

A utilização do **MinGW** permite efectuar o desenvolvimento com as ferramentas de **unix** (**gcc**, **make**, **flex**, **byacc**, **pburg**, **nasm**) mas o executáveis são gerador no formato **PE** (com extensão `.exe`). Notar que os exemplos têm de ser gerados com **nasm -fwin32** e **link**.

1.9.1 Diferenças entre os formatos win32 e elf

Embora o formato dos ficheiros objecto **win32** e **elf** seja significativamente distinto, as directivas de assembly utilizadas pela ferramenta **nasm** são quase iguais.

A primeira diferença consiste no nome do segmento de dados constantes, sendo designado por **rdata** em **win32** e por **rodata** em **elf**, mas a ferramenta **nasm** faz a conversão de forma transparente podendo utilizar-se apenas **rodata** em ambos os formatos.

A directiva **global** em **win32** não permite a indicação do tipo de símbolo global – **função** (`pfFUNC`) ou **dados** (`pfOBJ`) – devendo usar-se apenas `pfNONE`.

2 KNOPPIX: ambiente linux por CD-ROM

O **knoppix** é um CD-ROM que contém um ambiente **linux** bastante completo, sem necessitar de instalação no disco rígido. Assim, ao ligar o computador, coloca-se o CD-ROM no leitor de CD-ROM de arranque e fica-se com um ambiente **linux**. O **knoppix** permite criar um directório no disco rígido (ou amovível) para guardar os ficheiros **linux** de uma sessão para outra, e um directório de ficheiros de configuração quando necessário. Estes directórios podem ser criados em qualquer dos sistemas de ficheiros suportados pelo **linux**, incluindo **fat** (MS-DOS) ou **vfat** (*windows*). A informação no CD-ROM **knoppix** encontra-se comprimida, pelo que o disco contém quase 1.5Gb de aplicações e um vasto suporte para todo o tipo de periféricos.

Nesta secção trataremos da preparação do **knoppix** para a utilização na disciplina de **Compiladores**. Assume-se que o computador contém um outro sistema operativo instalado (provavelmente *windows*) e cerca de 10Mb de espaço em disco livre. Considera-se que o computador possiu periféricos comuns que não carecem de configuração especial, ou seja, caso possua um computador muito recente pode não haver ainda suporte em **knoppix** ou se tiver algum periférico pouco usual pode necessitar de opções de configuração específicas. (Não tratado neste documento.)

2.1 Criação de uma área knoppix no disco rígido

A criação de uma área **knoppix** no disco rígido é efectuada da primeira vez que se usa o **knoppix** num dado computador. Tal operação só é necessária caso pretenda salvaguardar ficheiros entre utilizações distintas. Para tal é necessário colocar o CD-ROM no leitor de CD-ROM de arranque e iniciar o computador. Notar que a configuração da BIOS deverá aceder ao CD-ROM antes do disco rígido, e caso tal não esteja definido deverá alterar a ordem de arranque na configuração da BIOS.

Ao aparecer, no fundo do ecrã a mensagem **boot:** deverá premir apenas a tecla de ENTER. Das vezes seguintes, quando a área **knoppix** já estiver criada no disco rígido deverá digitar `home=scan` antes de premir a tecla de ENTER.

Aguardar o arranque do sistema operativo e verificar se todos os periféricos a usar são reconhecidos pelo **linux**, até obter um ecrã semelhante ao **windows** com uma barra de menus inferior. Nesta barra existe um icone com um **pinguim**, devendo seleccionar sucessivamente as opções: **configure** e **create a persistence KNOPPIX home directory**.

Os discos IDE são designados por `/dev/hda`, `/dev/hdb`, `/dev/hdc` e `/dev/hdd`, podendo ter até 4 partições primárias: `/dev/hda1` a `/dev/hda4` no disco principal (de arranque), podendo cada uma ser dividida em partições extendidas. Os periféricos SCSI

podem conter até 7 discos por controlador (`/dev/sda` a `/dev/sdg`), e cada disco pode conter até 7 partições: por exemplo, de `/dev/sda1` a `/dev/sda7` no disco principal.

Ao escolher a partição do disco onde colocar a área **knoppix**, usar de preferência o disco principal (para facilitar a busca automática em utilizações futuras), ou seja, `/dev/hda1` (disco IDE) ou `/dev/sda1` (disco SCSI).

O **knoppix** permite criar a área codificada com uma palavra passe, escolhendo a opção `AES256`, ou sem codificação escolhendo `NO`. Cuidado que caso esqueça a palavra passe **nunca** mais pode recuperar os ficheiros, apenas poderá apagar a área e criar outra. Assim, a menos que existam razões de segurança que o justifiquem, aconselha-se a que se escolha a opção sem palavra passe.

2.2 Preparação do ambiente para a disciplina de Compiladores

Uma vez criada a área **knoppix** podem-se criar os ficheiros e directórios necessários. Este processo só necessita ser efectuado uma vez após a criação da área **knoppix**, ou sempre que se arranca o computador, se tal área não for criada, tendo o cuidado de salvaguardar os ficheiros modificados antes de desligar o computador.

Para a disciplina de Compiladores sugere-se a criação de, pelo menos, três directórios:

wrk: a *sandbox* para os projectos do CVS.

vazio: um directório mantido sempre vazio para criar novos projectos CVS, que não resultem de importações de projectos já existentes.

bin: para a colocação de aplicações, por exemplo, o executável do compilador a desenvolver.

Notar que o projecto da disciplina de Compiladores está colocado num repositório remoto, no servidor da disciplina e indicado através da variável de ambiente **CVSROOT** (ver capítulo de CVS).

Para configurar a área de trabalho aconselha-se a utilização do terminal com a opção `create terminal` (representado pelo icone de terminal com uma prompt), criar os directórios e editar o ficheiro `~/.bashrc` (substituir por `~/.cshrc` caso use *csh* ou *tcsh*):

```
$ mkdir bin wrk vazio
$ emacs .bashrc
```

Definir as variáveis de ambiente no ficheiro `~/.bashrc`:

CVSROOT: local do repositório CVS que, inicialmente e para experiências, pode ser local `CVSROOT=$HOME/cvs`, mas durante a execução do projecto deverá indicar o respositório atribuído na inscrição para o projecto.

EDITOR: o editor de texto a utilizar e que será automaticamente invocado pelo CVS quando salvaguardar alterações ao projecto, excepto se indicar a opção **-m**. Aconselha-se a escolha de um editor de texto (não confundir com processador de texto, como o **word** do **windows**) evoluído como o **emacs** ou o **vi**, evitando editores mais básicos como o **pico**, **joe**, *etc*.

PATH: indicar o local onde devem ser procurados os programas a executar. Acrescentar à lista existente o directório local `$HOME/bin`, onde devem ser colocados os executáveis dos programas desenvolvidos e as aplicações necessárias, que não constem da distribuição do **knoppix**.

PROJ: o nome do projecto da disciplina, que varia de ano para ano e que designaremos por `proj`, mas deverá ser substituído pelo nome correspondente.

O aspecto do ficheiro `~/.bashrc` deverá ser semelhante a:

```
export CVSROOT=$HOME/cvs
export EDITOR=emacs
export PATH=${PATH}:$HOME/bin
export PROJ=proj
```

Executar o ficheiro acima, para definir as variáveis: `$ . ~/.bashrc` (ou `$ source ~/.cshrc` se se tratar de *csh* ou *tcsh*). Das vezes seguintes, o ficheiro acima é executado automaticamente não sendo necessário executá-lo.

Seguidamente basta iniciar o repositório (cvs init) e criar o projecto inicial (cvs import), obter uma primeira cópia de trabalho (cvs co), introduzir novos ficheiros nesse projecto (cvs add) e salvaguardar os novos ficheiros e alterações aos já existentes (cvs commit):

```
$ cvs init
$ cd vazio
$ cvs import -m ‘‘versão inicial’’ $PROJ compiladores inicial
$ cd ../wrk
$ cvs co $PROJ
$ cd $PROJ
$ emacs Makefile
$ cvs add Makefile
```

```
$ cvs commit
$ emacs $PROJ.l
$ cvs add $PROJ.l
$ emacs Makefile
$ cvs commit
```

Desligar o computador através da opção **shutdown** do menu inicial, tal como em **windows**.

Cuidado: ao aceder aos mesmos ficheiros a partir de **windows**, as ferramentas de **windows** acrescentam no fim das linhas o carácter **CR** ASCII (código 0x0D) que pode baralhar algumas ferramentas de **linux**, nomeadamente o **flex**. Se tal acontecer, os caracteres podem ser removidos com o comando: `tr -d '\015' < ficheiroComCR > ficheiroSemCR`

Não esquecer que nos arranques seguintes do **knoppix** indicar na *prompt* de arranque (boot:) a opção `home=scan` (ou indicar o periférico específico `home=/dev/hda1`) para poder voltar a aceder à área de trabalho criada. Depois basta criar o terminal e continuar o trabalho:

```
$ cd wrk/$PROJ
$ emacs $PROJ.l
$ emacs $PROJ.y
$ cvs add $PROJ.y
$ emacs Makefile
$ cvs commit
```

3 O sistema UNIX

Antes de poder compreender as funcionalidades das ferramentas é necessário introduzir, de uma forma breve, o sistema operativo com o qual interagem. O *unix* é um sistema operativo multi-utilizador onde podem coexistir aplicações em execução pertencentes a diversos utilizadores, sem que haja interferências indesejadas entre elas.

Por razões de eficiência o núcleo do sistema operativo *unix* representa as entidades que manipula por números, como por exemplo aplicações em execução (processos), ficheiros ou utilizadores. As mais importantes são o utilizador, designado por **UID** (*user identifier*), grupo (**GID**), processo (**PID**), ficheiro (**INODE**) e dispositivo (representado por dois números: **MAJOR** e **MINOR**).

3.1 Utilizadores

Os utilizadores de uma máquina *unix*, excepção feita a sistemas que usem utilizadores de rede como as páginas amarelas, estão listados no ficheiro `/etc/passwd`. Este ficheiro pode ser lido por qualquer utilizador, correspondendo uma linha do ficheiro a cada utilizador, com os campos separados por ':' e contém a seguinte informação:

login: nome do utilizador no sistema, em geral constituído apenas por siglas.

passwd: palavra passe cifrada que permite ao utilizador registar-se (fazer o *login*) no sistema. Por razões de segurança, nos sistemas mais recentes, a palavra passe cifrada é substituída por asteriscos, encontrando-se esta no ficheiro */etc/shadow* que apenas pode ser lida pelo administrador do sistema. Notar que a palavra passe é codificada por um algoritmo sem inversa, pelo que nem o administrador pode saber qual a palavra passe digitada pelo utilizador. O acesso ao sistema é permitido se a palavra passe depois de cifrada for igual à existente no ficheiro. O acesso indevido ao sistema é apenas conseguido através da sucessiva tentativa de palavras frequentemente utilizadas como nomes de pessoas ou animais, ou elementos pessoais como números de documentos de identificação ou datas de nascimento, *etc*.

uid: número de utilizador, é com este número que o sistema operativo *unix* identifica o utilizador, bem como os seus ficheiros ou processos. Notar que pode haver mais de um par *login-passwd* a que corresponde o mesmo **UID**, representado esses *login* a mesma entidade para o *unix* (tal como dois cartões multibanco sobre a mesma conta bancária). O *unix* irá apresentar o primeiro nome a que corresponde o **UID** como o dono da entidade. De referir que o administrador do sistema é aquele

que tem **UID=0** (em geral designado por **root**) e não deverá corresponder a um utilizador físico (pessoa) mas representa uma identidade a que qualquer utilizador pode ganhar acesso (cuidado, todas as tentativas ficam registadas com a indicação de quem tentou e quando). Da mesma forma, qualquer utilizador pode ganhar a identidade de outro utilizador introduzindo a respectiva palavra passe, incluíndo a **root**, através do comando `su`. Em geral, a **root** não pode entrar directamente do sistema, excepto através da consola (terminal principal), devendo primeiro entrar como outro utilizador.

gid: número do grupo primário a que o utilizador pertence. O sistema *unix* permite que os ficheiros possam ser partilhados entre grupos (ver 3.2.1), podendo um utilizador pertencer a mais de um grupo (`/etc/group`). No entanto, quando os ficheiros são criados, o grupo utilizado é o grupo primário, podendo ser alterado posteriormente para qualquer dos grupos a que o utilizador pertence (`chgrp`).

gecos: informação facultativa sobre o utilizador físico, tal como o seu nome completo, sala de trabalho, telefone, *etc*. (gecos significa "General Electric Comprehensive Operating System")

home: directório de trabalho. Cada utilizador, quando ganha acesso ao sistema, é colocado num directório base, supostamente distinto para cada utilizador. Deverá ser dentro deste directório que o utilizador cria o seus ficheiros e aplicações.

shell: nome do interpretador de comandos inicial, que deverá fazer parte dos indicados em `/etc/shells` (apenas algumas ferramentas efectuam tal verificação).

O ficheiro de grupos (`/etc/group`) é semelhante ao de utilizadores contendo, separado por ':', o nome do grupo, a palavra passe (em geral vazia, indicando que não é necessária a sua indicação), o número de grupo (**GID**) e uma lista de utilizadores pertencentes ao grupos e separados por vírgulas.

3.2 Ficheiros e protecções

Como já foi referido, do ponto de vista do núcleo sistema operativo *unix*, todas as entidades são referidas por número e não por nome. No caso dos ficheiros estes são referidos pelo número do seu **INODE**, dentro de cada sistema de ficheiros. Assim, em sistemas de ficheiros para *unix* ao contrário do formato *fat* usado pelo **dos/windows**, um ficheiro pode ter vários nomes desde que cada nome refira o mesmo **INODE**. Por cada nome do ficheiro, também designado por *hard-link* (por oposição aos *soft-links* que veremos mais adiante) o sistema incrementa um contador de ligações (*links*). O ficheiro só é apagado do disco quando o seu número de ligações é zero, ou seja quando todos os

nomes foram apagados. Como o número do **INODE** só é único dentro de um sistema de ficheiros, não é possível criar *hard-links* entre discos ou partições de discos distintas (ver *soft-links*). Os nomes alternativos, não o primeiro, são criados com o comando `ln` (com a opção **-s** no caso dos *soft-links*) e apagados com o comando `rm`, como qualquer ficheiro. Notar que não existe maneira de distinguir dois nomes de um mesmo ficheiro, nem de saber qual foi criado primeiro e qual foi criado depois. Da mesma forma, a protecção, dimensão e datas de acesso e modificação são as mesmas para os dois nomes já que referem o mesmo ficheiro. Na prática, os ficheiros em si não têm nome apenas os directórios lhes dão nome, estabelecendo uma associação entre o nome e o **INODE**.

O sistema *unix* permite 6 tipos de ficheiros, designado pela letra entre parênteses:

directório (d): ficheiro que inclui referências a outros ficheiros.

character (c): dispositivo cujo a informação é transferida carácter a carácter, como por exemplo terminais, ratos, placas de audio. Estes dispositivos são identificados pelo sistema por um número que indica o tipo de dispositivo (**MAJOR**) e um que indica o dispositivo dentro desse controlador. Por exemplo, todos os terminais série têm o mesmo **MAJOR** sendo distinguidos pelo seu **MINOR** e não pelo seu nome.

block (b): dispositivo cujo a informação é transferida em blocos, como por exemplo discos. Tal como os outros dispositivos, todos os discos IDE têm um **MAJOR** diferente dos disco SCSI, tendo cada disco um **MINOR** distinto.

socket (s): ponto de ligação entre aplicações locais. A sua abertura permite entrar em contacto com a aplicação que criou o ficheiro.

link (l): nome simbólico alternativo para um ficheiro (*soft-link*). Na realidade trata-se de uma espécie de um ponteiro para outro ficheiro, uma vez que as operações não são efectudas sobre o ficheiro mas sobre o ficheiro por ele indicado.

texto (-): qualquer tipo de ficheiro que não seja um dos casos acima. Notar que embora se designe por texto pode designar programas executáveis, bases de dados, folhas de cálculo e não apenas texto no seu sentido restrito.

Os nomes ficheiros em *unix* podem conter qualquer carácter, incluíndo acentos, excepto o separador de directórios '/' (barra para a frente) e o terminador '\0' (carácter cujo código é zero, que não deve ser confundido com o carácter '0' cujo código é 48 decimal). Um ficheiro é designado univocamente pelo seu caminho (*pathname*) completo (ou absoluto) que começa na origem do sistema de ficheiros, por exemplo `/home/user/dir/file`. Para não ser necessário indicar sempre o nome completo,

cada processo mantém um directório corrente (mostrado com `pwd` e alterado com `cd`), por exemplo `/home/user`, bastando indicar o caminho relativo, neste caso `dir/file`. Além disso, todos os directórios incluem os nomes '.' e '..' para designar o próprio directório e o directório anterior na hierarquia, por exemplo `./dir/./../../file` ou `../user/dir/file`. O comprimento máximo do nome está limitado a 255 caracteres e o seu caminho completo a 1023 caracteres.

Como os ficheiros em *unix* não necessitam de extensão, embora algumas aplicações o possam exigir, existe o comando `file` que permite determinar o possível tipo de informação contida no ficheiro com base na análise do seu conteúdo.

3.2.1 Protecções

Todos os ficheiros, incluindo os ficheiros de texto, os directórios e dispositivos em *unix* são considerados como ficheiros, têm três conjuntos de permissões (utilizador, grupo e mundo) e conjunto de previlégios (*sticky bits*). Cada conjunto de permissões inclui autorizações de leitura (r), escrita (w) e execução (x), sendo cada operação não autorizada traçada (-). Por exemplo, **rw-** indica permissão de leitura e escrita mas não de execução e **--x** indica permissão apenas de execução. As permissões são atribuídas ao utilizador (dono do ficheiro) e ao grupo (não inclui o dono do ficheiro, mesmo que pertença ao grupo) e ao mundo (todos os restantes utilizadores do sistema que não pertençam ao grupo nem sejam o próprio dono). Assim, as permissões completas de um ficheiro podem ser **rwxr-x--x** onde o utilizador pode fazer tudo, o grupo não pode escrever e o mundo apenas pode executar (notar que a permissão de executar não permite copiar o ficheiro para o qual é necessário poder ler). Da mesma forma, **r--w---x** permite ao utilizador ler mas não escrever nem executar (mesmo que pertença ao grupo) e aos restantes membros do grupo escrever mas não ler nem executar e ao mundo apenas executar.

No caso dos directórios entende-se ler como a capacidade para mostrar o seu conteúdo, escrever como a capacidade de modificar (acrescentar ou apagar) ficheiros a um directório e executar como a capacidade de procurar um ficheiro num directório. Notar que num directório apenas com permissão de execução só estão acessíveis os ficheiros dos quais se sabe o seu nome, facilitando a partilha restrita de ficheiros, embora o conteúdo do ficheiro possa eventualemente ser acedido por sucessivas tentativas de possíveis nomes.

Os previlégios são representados no lugar da autorização de execução e modificam essa permissão. Um ficheiro pode ser executado com os previlégios do dono (`suid`) e/ou grupo (`sgid`) do ficheiro, em vez dos previlégios de quem o manda executar. Assim, a aplicação `su`, para poder ganhar previlégios de administrador, tem de pertencer ao utilizador **root** ser `suid`, ou seja, **rws--x--x**. No entanto, a maioria dos dispositivos de rede e amovíveis é montada (ligada ao sistema através do comando `mount`) com a opção **nosuid** para evitar "cavalos de Troia".

No caso dos directórios a utilização de previlégios no mundo, designado por 't', por exemplo **rwxrwxrwt**, permite que qualquer utilizador possa criar e apagar e ficheiros no directório apenas se lhe pertencerem, tal é o caso dos directórios de ficheiros temporários `/tmp` e `/usr/tmp`. Notar que em **rwxrwxrwx** qualquer utilizador pode apagar os ficheiros de outros utilizadores, incluíndo os do administrador.

3.2.2 O Comando ls

O comando `ls` (*list source*) permite mostrar o conteúdo de um directório por ordem alfabética. Sem opções, o comando indica apenas os nomes dos ficheiros do directório corrente. A opção **-i** (`ls -i`) permite mostra o conteúdo completo do directório, ou seja, apenas o número do **INODE** e o nome a ele atribuído neste directório.

De maior utilidade são as opções que permitem mostrar não só o conteúdo do directório como a informação de controlo do ficheiro. A mais utilizada é a opção **-l** (*long*), por exemplo:

```
total 192
drwxr-x--x    2 root     users        4096 Sep 29 14:18 dir
-r-xr-x--x    1 root     users      184565 Sep 29 14:18 file
```

onde se indica:

tipo e permissões: uma letra para o tipo, seguido dos três grupos de permissões.

número de *links*: número de nomes alternativos de um ficheiro. Os directórios são criados com 2 *links*: o seu nome no directório corrente e o nome '.' dentro de ele próprio. Da mesma forma um directório com 8 *links* tem 6 sub-directórios já que cada um destes 6 tem um nome '..' para este directório.

nome do utilizador: primeiro nome de `/etc/passwd` que tem o **UID** existente no ficheiro.

nome do grupo: primeiro nome de `/etc/group` que tem o **GID** existente no ficheiro.

dimensão em bytes: esta dimensão não corresponde necessariamente ao espaço ocupado em disco pois a reserva é feita por blocos (em geral de 512 bytes, 1Kb, 2Kb ou 4Kb). Para mais, se depois de criar um ficheiro apenas se escrever um único byte na posição 1000000 este apresenta uma dimensão de 1000000 e ocupa apenas um bloco em disco (o necessário para guardar esse byte). Se for um dispositivo este número é substituído pelo **MAJOR** e **MINOR** separados por uma vírgula (ver ficheiros no directório `/dev`).

data da última modificação: a data inclui também a hora e minuto se a alteração ocorreu à menos de 6 meses e o ano em caso contrário. Notar que a data da modificação efectiva do ficheiro inclui inclusivamente o segundo em que a alteração foi efectuada e pode ser obtida com o comando `stat`, que indica igualmente a data da criação e do último acesso aos dados, incluíndo o fuso horário da máquina que efectuou a alteração, bem como o número de blocos ocupados, o **INODE** e, o dispositivo do **INODE**.

nome: o nome utilizado para aceder ao ficheiro.

Existem muitas outras opções que podem ser consultadas na página de manual: `man ls`.

3.3 Sistemas de ficheiros

Os sistemas de ficheiros representam entidades autónomas para a salvaguarda de informação pelos sistemas operativos. Um sistema de ficheiros pode residir num disco rígido ou amovível, ou em parte deste (partição). Em casos especiais, é possível permitir que um só sistema de ficheiros resida em mais de um disco ou partição, quer por limitações de espaço como por razões de fiabilidade e tolerância a falhas.

O mais comum, no caso de discos amovíveis ou de pequena capacidade, é o sistema de ficheiros ocupar todo o disco. Em discos de maior capacidade é possível definir partições independentes, evitando que a corrupção dos dados de uma partição afecte as restantes e permitindo diferentes utilizações para cada partição (por exemplo, a coexistência de dois sistemas operativos).

Um disco SCSI pode ter até 7 partições e um disco IDE pode ter até 4 partições primárias e estas serem subdivididas em partições extendidas, geridas por `fdisk` (ou `sfdisk` ou ainda `parted`, entre outros). Um disco encontra-se dividido em `cabeças` (não necessariamente igual ao número de cabeças de leitura fisicamente existentes no disco), `cilíndros` (aneis concêntricos que, em geral, constituem a unidade física de leitura e escrita) e `sectores` (que constituem a unidade de formatação). Uma partição, e por consequência um sistema de ficheiros, deve sempre conter um número inteiro (nunca fraccionário) de cilíndros. A formatação consiste em definir blocos de dados (`sectores`) e respectivos blocos de controlo de erro, por exemplo, a cada bloco de 512 bytes de dados está associado um bloco de controlo de erro de normalmente 32 bytes. Uma operação de escrita regista não só os dados como o controlo de erro associado, calculado a partir desses mesmos dados. Na operação de leitura recalcula-se a informação de controlo de erro, com base nos dados lidos, e verifica-se se coincide com a informação de controlo guardada anteriormente. Em caso de discrepância, a leitura é repetida várias vezes, em geral cerca de 16, até ser dado um erro. A informação de controlo de erro (ECC, mas não CRC) pode ainda ser utilizada para reconstruir parte dos dados danificados, já que existe redundância de informação.

Um sistema de ficheiros é construído numa partição, definindo a forma como a informação está organizada e é acedida. O comando `mkfs` permite criar um novo sistema de ficheiros (de entre os vários possíveis em *unix*) e o comando `tunefs` permite alterar dinamicamente muito dos seus parâmetros.

3.3.1 Acesso aos sistemas de ficheiros

Inicialmente, o sistema operativo *unix*, cria o directório base '/' em memória. Depois estabelece uma ligação entre este directório imaginário e um primeiro sistema de ficheiros, designado por *root file system*, que contém a informação necessária ao arranque do sistema. Com o sistema já a funcionar pode agora ligar-se a directórios deste *root file system* outros sistemas de ficheiros, e assim sucessivamente.

O comando `mount` permite ligar a origem de um sistema de ficheiros a um directório de outro sistema de ficheiros já acessível, funcionando o referido directório como uma porta para o sistema de ficheiros ligado. A lista de ligações predefinidas pelo sistema operativo encontra-se no ficheiro */etc/fstab* e pode ser modificado por forma a incluir novos pontos de ligação como ZIPs, PENs, *floppy*s, CDROMs, *etc*. O ficheiro, à semelhança do comando `mount` determina o periférico que contém o sistema de ficheiros, o directório onde deve ser efectuada a ligação (montagem), o tipo de sistema de ficheiros (dos, vfat, ext2, ext3, minix, hpfs, ufs, *etc*.) e um conjunto de opções. Destas opções salienta-se:

noauto	não deve automaticamente montado no arranque (discos amovíveis)
user	os ficheiros pertencem a quem executa o comando `mount`
owner	os ficheiro pertencem ao dono do directório onde é montado
ro	os ficheiros são apenas para leitura
nosuid	o *sticky bit* dos ficheiros é ignorado
uid=XXX	todos os ficheiros têm o UID indicado
gid=XXX	todos os ficheiros têm o GID indicado
mode=XXX	todos os ficheiros têm a protecção indicada (por exemplo, mode=620)

Por exemplo,

```
/dev/hda1      /win            vfat    uid=100,gid=100         0 0
/dev/cdrom     /mnt/cdrom      iso9660 noauto,owner,ro         0 0
/dev/sda1      /mnt/usb1       auto    noauto,user             0 0
/dev/sdb1      /mnt/usb2       auto    noauto,user             0 0
/dev/sda4      /mnt/zip        auto    noauto,user             0 0
```

O comando `mount` só pode ser utilizado pela **root**, excepto se existir uma entrada no ficheiro `/etc/fstab` e essa entrada incluir `user` ou `owner` (e o utilizador que pretende fazer o `mount` for o mesmo que o `owner` do ponto de ligação).

Os sistemas de ficheiros podem ser desligados com `umount`, indicando os pontos de ligação (os directórios que servem de porta).

3.4 Processos

Para poder executar uma aplicação o sistema operativo cria um processo. O pedido de execução pode ser efectuado pelo interpretador de comandos ou por qualquer outra aplicação do sistema. Esse processo terá um número de execução **PID** e os mesmos previlégios do processo que fez o pedido, excepção feita quando o programa a executar é um ficheiro com previlégios especiais, como o **suid** ou o **sgid**. Cada processo mantém a indicação do comando que lhe deu origem e do processo que efectuou o pedido (pai) **PPID** (*parent process identifier*), entre outra informação. Quando um processo termina, o processo pai é avisado da sua terminação e seu estado (sucesso, tipo de erro, gerou **core**, *etc*.), e caso este já tenha terminado o processo *init* (**PID=1**) é notificado.

O número de processo é um inteiro positivo entre 1 e 65535 (por enquanto), o núcleo é o processo 0, sendo reaproveitados os números de processos entretanto terminados. Assim, o número máximo de processos simultaneamente em execução está limitado a 65535 embora seja imposto um limite por utilizador, incluíndo a **root**, para evitar erros devido a ciclos ou a apropriação de todos os processos por um só utilizador.

Um processo pode estar em diversos estdos, dos quais os mais comuns são execucação (R - *running*), suspenso (S - *suspended*), parado (T - *stopped*), bloqueado (D - *device I/O bloqued*) ou moribundo (Z - *zombie*), entre outros. Um processo fica `suspenso` a seu pedido (acorde-me daqui a 1 minuto) ou quando o periférico pelo qual espera está inactivo (pediu para ler uma tecla mas ninguém carregou em nenhuma). Se o periférico estiver activo a servir esse processo ou um pedido anterior de outro processo, o processo fica `bloqueado` à espera que o periférico consiga satisfazer o seu pedido. Um processo fica `parado` através do envio de uma interrupção (STOP ou TSTP), normalmente associada à tecla (^Z) do interpretador de comandos, sendo reactivado com uma interrupção (CONT), normalmente associada ao comando `fg` ou `%` do interpretador de comandos, ficando congelado em memória no entretanto, e recomeçando exactamente no mesmo ponto onde foi suspenso. Finalmente, um processo fica `moribundo` quando já terminou a sua execução mas o processo pai ainda não reconheceu a sua terminação. Este facto indicia um mau funcionamento do processo pai e pode levar à exaustão do processos disponíveis de um utilizador (erro comum na aprendizagem de sistemas operativos).

O comando `ps` permite mostrar os processos em execução, sendo em geral indispensável para identificar o número dos processos. A informação disponibilizada, através da activação de diferentes opções inclui o número do processo e do pai, a utilização de CPU e memória, o terminal a que está associado, a data de início do processo e o tempo de CPU já gasto, entre outros. As opções mais comuns deste comando são **x** para todos os processos do utililidor e **ax** para todos os processos do sistema. Informação adicional

pode ser obtida com a opção **u**. Existem diversas versões deste comando, para as diversas variantes de *unix* e para diferentes versões destas variantes, onde a mesma letra pode ter significados distintos, pelo que a leitura da página de manual é indispensável: `man ps`.

3.4.1 Inicialização

O arranque do sistema operativo é efectuado carregando para memória uma ficheiro que contém o código executável do núcleo do sistema operativo. Este primeiro programa é carregado em modo priveligeado e, depois de reconhecer os dispositivos e se configurar, lança em modo utilizador (não priveligeado) o programa `init` em modo administrador (**root**). É responsabilidade deste programa iniciar as restantes aplicações, nomeadamente os programas que permitem aos utilizadores ganharem acesso à máquina (*login*), com os respectivos previlégios. Quando o processo `init` termina o núcleo do sistema operativo (o processo pai) é avisado e desliga os dispositivos, parando o computador (*shutdown*). Por exemplo, ao premir CTRL-ALT-DEL o núcleo envia uma interrupção SIGINT ao processo `init`.

3.4.2 Prioridade

Os processos em *unix* têm associada uma prioridade, de tal forma que um processo com maior prioridade é sempre escolhido para executar sobre outros de menor prioridade. Notar que se um processo estiver à espera de um periférico, por exemplo, caracteres de um terminal ou blocos de um disco, os processos de menor prioridade podem executar, enquando o pedido que originou a espera não for satisfeito.

As prioridades dos processos em modo utilizador variam entre **-20** (a maior prioridade) e **19** a menor prioridade. Em geral, um processo executa com prioridade **0** (zero) podendo o dono (processo com o mesmo **UID**) diminuir a prioridade (na realidade, aumentar o número que a representa). Qualquer aplicação pode ser iniciada em baixa prioridade, por omissão prioridade **10**, precedendo o comando habitual por `nice`, por exemplo `nice ls -l`. Da mesma forma, qualquer utilizador pode diminuir a prioridade de qualquer processo em execução com o seu **UID** com o comando `renice` indicando a nova prioridade (ou incrementos desta) e os números dos processos a serem afectados. O administrador do sistema, utilizador **root** (**UID=0**), pode aumentar ou diminuir as prioridades de qualquer processo, excepto o núcleo do sistema operativo.

3.4.3 Interrupções

Os processos podem comunicar entre si através de interrupções. De momento existem 32 interrupções da quais 30 estão predefinidas e 2 são de utilização livre (USR1 e USR2). O envio de uma interrupção de um processo para outro está restringída a processos do mesmo utilizador (**UID**), excepto a **root** que pode enviar a todos os processos em modo utilizador. Notar que algumas interrupções podem ser enviadas pelo núcleo, como por exemplo o habitual "*segmentation fault*" (SEG), a falta de energia (PWR). Do ponto de vista do processo todas as interrupções, com a excepção do CONT e KILL, podem ser interceptadas ou ignoradas. Por interceptar entende-se invocar uma rotina que corrige o problema e permite a continuação da execução da aplicação. Claro que muitas interrupções, como o SEG ou o ILL (*illegal instruction*), são de difícil solução com o programa em execução. No entanto, muitas interrupções, como os relógio ALRM (alarme) e VTALRM (alarme em tempo virtual), HUP (*hanghup* - usado por muitas aplicações para reler os ficheiros de configuração), INT (*interrupt*, o ^C do terminal), PIPE (fecho de ligação entre processos), CHLD (processo filho terminou), WINCH (*window change*, dimensão da janela foi alterada) podem e devem ser interceptadas por aplicações que utilizem esses recursos.

As interrupções podem ser enviadas a partir do interpretador de comandos através do comando `kill`, sendo especialmente úteis as de terminação, a saber por ordem crescente de 'poder destructivo': INT (*interrupt*), QUIT (o ^\ do terminal), TERM (*terminate*) e KILL (que não pode ser interceptado ou ignorado pela aplicação e que consequentemente garante a sua terminação, mas impede a salvaguarda de alguma informação por parte da aplicação).

3.4.4 Memória

Em *unix* devido ao modo de partilha de memória utilizado, dois comandos iguais aproveitam o código comum, apenas não partilhando a informação distinta (*copy on write*). Além disso, como a memória física é reduzida, todos os processos vão sendo copiados para disco, para uma área designada por *swap*, sempre que partes do seu código (páginas) não estejam a ser utilizadas. Quando essas partes voltarem a ser necessárias terão de ser trocadas com partes de outros processos que não estejam agora em uso, daí o termo *swap* (troca). A memória só fica esgotada quando os processos não couberem na memória física mais o espaço de *swap*, o que origina a mensagem *out of swap space*. No entanto, um sistema *unix*, pode começar a ficar muito lento se tiver pouca memória física, pois o sistema perde mais tempo a trocar as páginas que a executar o seu conteúdo (*trashing*), considere-se o caso limite em que a memória física comporta uma única página.

Estas situações podem ser acompanhadas através do comando `ps`, embora a solução passe por executar menos processos ou adquirir mais memória física.

4 shell: interpretador de comandos

O interpretador de comandos, ou *shell*, constitui a interface entre o utilizador e o sistema operativo *unix*. O *shell* permite aceder ao recursos do sistema operativo através de chamadas especiais (*system calls*) e lançar e controlar a execução das aplicações. Contudo, o *shell* é também uma linguagem de programação completa, embora mais dedicada à manipulação de ficheiros e texto que ao cálculo numérico.

Na realidade o *shell* é uma aplicação como qualquer outra, sem previlégios especiais, podendo cada pessoa escrever, usar e distribuir o seu próprio interpretador de comandos. No entanto, para evitar problemas de segurança, a maioria dos sistemas *unix* permite que apenas as aplicações indicadas no ficheiro `/etc/shells` possam ser utilizadas como interpretador de comandos no *login*. Claro que o administrador do sistema, o utilizador designado por **root**, pode alterar esta lista e cada utilizador pode usar temporariamente qualquer outro interpretador de comandos.

Os interpretadores de comandos distribuídos com o sistema operativo *unix* agrupam-se em dois grupos com características muito semelhantes e cujas diferenças serão indicadas ao longo do texto sempre que se justificar. O grupo de interpretadores que usa a sintaxe do interpretador original do *unix*, o `/bin/sh` (*Bourne Shell*), inclui também os interpretadores `/bin/bash`, `/bin/ash` e `/bin/ksh`. Um segundo grupo que procura apresentar uma sintaxe mais próxima da linguagem **C** inclui o `/bin/csh` (*C Shell*) e os interpretadores `/bin/tcsh` e `/bin/zsh`.

4.1 Comandos simples

Um comando consiste em uma ou mais palavras separadas por espaços brancos, onde a primeira palavra designa o comando a ser executado e as restantes os argumentos que parametrizam o comportamento do comando.

Quando o interpretador de comandos está apto a receber mais pedidos apresenta uma etiqueta ou *prompt*. Nos exemplos, a *prompt* será representada apenas pelo carácter $, embora o seu conteúdo possa ser modificado atribuíndo um novo texto à variável **PS1**.

No exemplo,

```
$ ls -l /bin
```

o comando é `ls` (*list source*) e tem dois argumentos. A maioria dos comandos corresponde a ficheiros executáveis acessíveis ao interpretador de comandos, neste caso o ficheiro situa-se em `/bin/ls`, mas podem também ser comandos internos do interpretador. O primeiro dos dois argumentos funciona como um modificador, alterando o

comportamento normal do comando, neste caso apresenta mais informação sobre os ficheiros (**-l**, long list). Em *unix*, os modificadores, também designados por opções, são precedidos pelo híphen (sinal de '-') e entecedem, em geral, os restantes argumentos. Contudo, os comandos que não fazem parte do sistema operativo podem utilizar outras regras.

4.2 Edição de comandos

A maioria dos *shell*s suporta alguma forma de reaproveitar comandos anteriores para construir novos comandos. Tal expediente é especialmente útil quando o *shell* é utilizado em modo interactivo, pois o utilizador escreve menos e reduz os erros de digitação.

4.2.1 Edição de linha

Interpretadores mais recentes como *bash*, *tcsh* e *zsh* possuem um editor de linha, que tal como um editor de texto, que permite operações básicas, com uma sintaxe próxima da usada no editor *emacs*. Por exemplo,

^A colocar o cursor no início da linha;

^E colocar o cursor no fim da linha;

^K apagar todo o texto do cursor até ao fim da linha;

^F avançar o cursor de um carácter;

^B recuar o cursor de um carácter;

^L redesenhar o ecrã colocando a linha no topo do terminal;

^W apagar a palavra antes do cursor;

^P subir um comando na história de comandos executados;

^N descer um comando na história de comandos executados;

entre muitos outros.

Existem outros, como o *zsh*, que usa uma sintaxe próxima do editor de texto *vi*.

4.2.2 Substituição

Quando dois comandos apresentam grandes semelhanças, diferindo apenas em parte, pode-se utilizar a substituição, onde se indica a parte do texto a substituir e a sua substituição. A limitação da substituição reside no facto de apenas a primeira ocorrência ser substituída, podendo haver uma só substituição de cada vez. Esta capacidade é especialmente útil em pequenas substituições em comandos extensos, por exemplo,

```
$ find . -type f -name "*.log" -print
...
$ ^log^aux
find . -type f -name "*.aux" -print
...
```

onde os '**...**' representam o resultado omitido do comando.

4.2.3 Historial

A maioria dos *shell*s pode manter um historial dos últimos **n** comandos digitados numa sessão, podendo inclusivamente manter esse historial entre sessões distintas (salvaguardando os comandos executados num ficheiro). Para tal recorre a uma variável que controla o número de comandos a guardar (**HISTSIZE** em *bash* ou **history** em *tcsh*, por exemplo). Para guardar os comandos entre sessões usa-se o ficheiro designado pela variável **HISTFILE** em *bash* (ou **histfile** em *tcsh*, que por omissão é o ficheiro **.history**) e a variável **HISTFILESIZE** (ou **savehist**, em *tcsh*) designa o número de comandos a guardar.

O comando **history** permite listar os últimos comandos digitados e o seu número de ordem (em alguns casos inclui também a hora em que foram executados). A repetição de comandos completos pode ser efectuada com recurso às substituições:

!! para repetir o último comando digitado,

!nnn para repetir o comando número **nnn** (o número de ordem indicado pelo comando **history**),

!-nnn para repetir o último comando número **nnn** em relativamente ao comando actual.

!ccc para repetir o último comando começado por **ccc** (não é necessário repetir o nome do comando completo, basta a(s) primeira(s) letra(s)).

!?xxx? para repetir o último comando que inclui a sequência **xxx** em qualquer local do comando (nome do comando, opções ou argumentos). O segundo '?' pode ser omitido se for imediatamente seguido de mudança de linha.

!# para repetir o comando corrente. Deve ser usado com muito cuidado para evitar recursão.

Além da repetição de comandos completos é possível seleccionar apenas parte do comando, ou seja, alguns dos seus argumentos:

!* representa todos os argumentos do comando anterior,

!^ representa o primeiro argumento do comando anterior,

!$ representa todos o último argumento do comando anterior,

!:nnn-mmm representa todos os argumentos entre o de ordem **nnn** e o de ordem **mmm** do comando anterior. Admite as variantes **!:nnn** para **nnn**-ésimo, **!:nnn-** para todos os argumentos desde o **nnn**-ésimo, **!:-mmm** para todos desde o início até ao **mmm**-ésimo (excepto o nome do comando, o **!:0**), entre outros.

4.3 Expressões regulares: *wildcards*

A maioria dos argumentos do interpretador de comandos designa ficheiros. Para facilitar a identificação de conjuntos de ficheiros, sem necessidade de os enumerar exaustivamente, os interpretadores de comandos permitem a utilização de expressões regulares (*wildcards*).

Por exemplo,

```
$ ls -l *.c
```

mostra todos os ficheiros com a extensão **.c**. As expressões regulares podem inclui:

***** representa zero ou mais caracteres.

? representa exactamente um só carácter.

[...] representa qualquer um dos caracteres entre parênteses rectos, podendo estes ser enumerados (por exemplo, [aeiou]) ou representados por conjuntos (por exemplo, [a-z]). Contudo, representa apenas um carácter do ficheiro, logo [a-z][0-9] designa todos os ficheiros cujo nome tem dois caracteres sendo o primeiro uma letra alfabética minúscula e o segundo um digito decimal. Da mesma forma, [a-z]* não designa ficheiro constituídos apenas por minúsculas, mas ficheiros em que o primeiro carácter é uma letra minúscula, podendo os restantes (a existir) ser qualquer um dos 254 caracteres possíveis.

4.4 Quoting e escape

Para permitir a utilização dos caracteres especiais utilizados nas expressões regulares, ou noutras funções do interpretador, com o seu valor real (em *unix* um ficheiro pode-se chamar *) todos esses caracteres necessitam de ser precedidos por \. O próprio carácter \ necessita de ser precedido por outro \ para representar um único \.

Para evitar ter de fazer o *quote* a todos os caracteres especiais pode-se incluir um argumento entre plicas (’). Neste caso, todos os caracteres entre plicas têm o seu valor normal, devendo a utilização de uma plica no seu interior ser precedida de \. Cuidado que o próprio separador de argumentos, o carácter espaço branco, passa a ter o seu valor normal e deixa de funcionar como separador. Por exemplo,

```
$ ls -l ’/windows/Os meus documentos’
```

tem apenas dois argumentos, enquanto na ausência das plicas seriam quatro argumentos. Assumindo que no directório principal não existem mais sub-directórios começados por **w**, o comando seguinte é idêntico ao anterior:

```
$ ls -l /w*/’Os me’us\ documentos
```

A utilização de caracteres que não podem ser impressos, como os caracteres de contolo (por exemplo, ^A ou ^Z), devem ser precedidos de ^V. Da mesma forma para introduzir um só ^V é necessário digitar dois ^V seguidos.

Os caracteres \ e ^V são designados como caracteres de escape, pois permitem escapar ou evitar a interpretação de qualquer carácter ou sequência que se lhes siga. As plicas, e como veremos mais adiante também as aspas, são delimitadores que fazem com que o texto contido entre eles tenha o seu valor literal (o valor escrito), sendo designados por caracteres de *quoting*.

4.5 Controlo de fluxo

Na sua forma mais simples, os comandos vão sendo sucessivamente digitados pelo utilizador e executados pelo interpretador de comandos. Contudo, em muitas situações, tem interesse alterar a forma como os diversos comandos interagem entre si sem necessidade de alterar esses mesmos comandos. O interpretador de comandos *unix* permite ligar comandos e ficheiros entre si por forma a produzir aplicações mais específicas.

4.5.1 Serialização

A serialização permite executar dois ou mais comandos em sequência, dependendo do separador utilizado:

`;` os comandos, separados por ';', são executados em sequência. Quando o primeiro termina inicia-se o segundo e assim sucessivamente.

`||` se o comando anterior ao separador '||' terminar com êxito então o comando seguinte não é executado.

&& se o comando anterior ao separador '&&' terminar com êxito então o comando seguinte é executado, caso contrário é ignorado.

Listas Notar que é possível a utilização de parênteses curvos ou chavetas para agrupar conjuntos de comandos e que todos estes caracteres são caracteres especiais, necessitando de ser *quoted* ou *escaped* para poderem ser utilizados em argumentos de comandos com o seu valor normal. A diferença entre os parênteses curvos e as chavetas reside no facto de no primeiro caso ser criado um sub-*shell* para executar a lista de comandos, enquanto nas chavetas é o próprio *shell* que os executa.

Por exemplo,

```
$ (cd /etc; grep main *.conf) && { cd /home; ls; }
```

o comando `grep` é executado depois de a directoria corrente ter sido alterada para `/etc` enquanto a lista de comandos `cd /home` e `ls` só é executado se o comando `grep` (o último do conjunto anterior) tiver tido sucesso (no caso de sucesso a directoria a corrente passa a ser `/home`, caso contrário não é alterada). Notar que os comandos entre chavetas são executados a partir da directoria corrente actual, não em `/etc`, já que os comandos entre parênteses curvos são executado num sub-*shell*, que tem a sua própria directoria corrente herdada do *shell* que lhe deu origem.

4.5.2 Redirecção

Um programa em *unix* inicia-se com 3 ficheiros abertos:

0 - stdin: o ficheiro de leitura, notar que a leitura do terminal é feita linha a linha (o programa só recebe o primeiro carácter depois de premida a tecla de `return` ou `enter`) e a leitura do disco é feita bloco a bloco (os blocos têm, em geral, entre 512 e 8192 bytes).

1 - stdout: o ficheiro de escrita, também linha a linha para terminais e bloco a bloco para disco.

2 - stderr: o ficheiro de erros, também um ficheiro de escrita mas para onde é suposto as aplicações reportarem os erros, sendo os dados trasnferidos carácter a carácter e não em linhas ou blocos (por forma a tornar os erros imediatamente visíveis).

Inicialmente, todos os 3 ficheiros referidos encontram-se associados ao terminal, sendo a leitura efectuada do terminal e a escrita e os erros enviados para este. A redirecção permite que sejam utilizados ficheiros em disco em substituição do terminal.

A redirecção da leitura permite enviar o texto contido num ficheiro para um comando, sem que este se aperceba que não é o utilizador a digitá-lo. Para tal basta incluir o carácter '<' seguido do nome do ficheiro no comando, por exemplo qualquer dos seguintes comandos é equivalente,

```
$ grep main < file.c
$ < file.c grep main
$ grep < file.c main
```

embora considere a primeira opção mais legível.

Em programas escritos em *shell* (será abordado mais adiante) poderá ser útil escrever o texto no próprio programa em vez de ter um ficheiro separado ou obrigar o utilizador do programa a fazê-lo. Tal é possível através dos chamados *here documents* que usam dois '<' e são seguidos do texto terminador, por exemplo

```
$ grep linha << fim
primeira linha
segunda linha
mais texto
fim
```

O ficheiro de saída pode ser redireccionado do terminal para um ficheiro em disco através da utilização de um ou dois '>' seguidos do nome do ficheiro. A diferença reside no facto de caso se utilize apenas um '>' o anterior conteúdo do ficheiro será primeiro apagado, enquanto a utilização de '>>' acrescenta ao texto já existente. Em ambos os casos o ficheiro é criado vazio, caso não exista.

```
$ grep main *.c > busca.txt
$ grep main *.h >> busca.txt
```

O redireccionamento da saída de erro (*stderr*) baseia-se na duplicação de descriptores, onde '2>&1' significa que o *stderr* é duplicado como *stdout*. Por exemplo,

```
$ ls > outerr 2>&1
$ ls 2>&1 > out
```

o primeiro caso redirecciona ambos os descritores (*stdout* e *stderr*) para o ficheiro designado por `outerr`, enquanto no segundo exemplo apenas o *stdout* é redirigido pois a duplicação é efectuada antes da redirecção.

Alguns interpretadores, como os do tipo *C shell*, permitem a notação '>&' ou '&>' para redireccionar ambos os descritores (1 - stdout e 2 - stderr), por exemplo

```
$ ls >& outerr
$ ls &> outerr
```

4.5.3 Background

A sequenciação permite que os comandos sejam executados uns após os outros, mas também é possível que os comandos corram em paralelo. Notar que mesmo em máquinas com um só processador, onde não é possível paralelismo real, consegue-se aproveitar os tempos mortos em que um processo aguarda a resposta de um periférico para executar outros processos. Por outro lado, quanto mais processos estiverem em execução simultânea maior é a quantidade de memória (física e virtual) necessária, logo maior o risco de se entrar em *trashing*.

Ao pôr um processo a executar em *background* o controlo é imediatamente devolvido ao interpretador de comandos, que pode iniciar novos comandos em paralelo com o já em execução. Para enviar um processo para *background* basta terminar o comando com '&', por exemplo

```
$ sleep 20 &
```

Para poder esperar por processos que executam em *background* (paralelo), o interpretador de comandos utiliza o comando `wait`. Este comando espera por todos os processos em *background* ou por um específico se for indicado o seu PID como argumento. A variável de ambiente (ver variáveis) `$!` contém o PID do último processo enviado para *background*.

4.5.4 Pipeline

Quando se pretende que os resultados de um programa sejam a entrada de outro pode-se recorrer à utilização de um ficheiro intermédio por redirecção. O *pipelining* entre processos permite eliminar o ficheiro intermédio ligando os dois programas por um 'tubo' (*pipe*) que transfere os dados de um para outro. Notar que os programas correm em simultâneo, ficando o processo que lê do *pipe* à espera do outro quando o *pipe* fica vazio. Por outro lado, o processo que escreve fica bloqueado quando o *pipe* fica cheio (os *pipes* têm uma capacidade máxima, frequentemente 8kb) por falta de leitura do outro processo.

A criação de um *pipeline* utiliza o carácter `'|'` como separador dos comandos, por exemplo

```
$ grep '^X' texto | sort | uniq | wc -l
```

procura as linhas iniciadas por 'X' no ficheiro 'texto' (grep), ordena alfabeticamente as referidas linhas (sort), retira as linhas duplicadas consecutivas (uniq) e conta o número de linhas resultantes (wc -l).

O *pipeline* pode transferir o *stderr* além do *stdout* utilizando a duplicação dos descriptores, ou no caso do *C shell* utilizando `'|&'`.

4.5.5 Substituição de comandos

A substituição de comandos permite utilizar o resultado de um comando como argumento de outro comando. Para tal basta delimitar o comando por plicas para trás. Por exemplo, se

```
$ grep -l main *.c
```

lista todos os ficheiros da linguagem **C** que contêm a palavra `main`, então

```
$ vi `grep -l main *.c`
```

permite editar esses mesmos ficheiros, tal como se os seus nomes tivessem sido enumerados na linha de comando.

4.6 Controlo de processos

O sistema **unix** permite não só enviar processos para *background*, que vão executando em paralelo, como suspender processos e resumir a sua execução sob o controlo do interpretador de comandos. A forma mais simples de suspender um programa consiste em enviar uma interrupção **TSTP** (*terminal stop*), em geral associada à tecla ^Z do terminal. Notar que ao voltar ao interpretador de comandos, o programa que estava em execução não terminou nem salvaguardou quaisquer ficheiros (excepção feita a alguns, poucos, programas que interceptam essa interrupção), ficando o programa suspenso.

O comando **jobs** permite saber quais os processos sob o controlo deste interpretador de comandos (notar que todos os processos são acessíveis através do comando **ps**). Por exemplo,

```
$ sleep 200 &
[1] 1133
$ cc main.c
^Z
[2]+  Stopped(SIGTSTP)        cc main.c
$ jobs
[1]-  Running                 sleep 200 &
[2]+  Stopped(SIGTSTP)        cc main.c
$
```

Para recomeçar a execução de um processo suspendo, ou ficar associado a um processo em execução, basta usar o comando **fg** (que pode ser abreviado para % na maioria dos *shell*s) seguido do número do sub-processo (não o seu PID, mas um número atribuído pelo *shell*). Caso este número seja omitido, será utilizado o processo com a indicação '**+**' na lista, que corresponde ao último processo suspenso ou, no caso de estarem todos em execução, ao último processo acedido.

Notar que é possível enviar para *background* processos suspensos, acrescentando um '**&**' no fim do comando **fg**. Por exemplo,

```
$ %2 &
$ fg 1
```

reinicia a execução do comando **cc** em *background* (ficando a executar em paralelo) e depois associa-se ao comando **sleep**, só voltando a ter a *prompt* disponível quando expirarem os 200 segundos (a contar da invocação do comando inicial), mesmo que o comando de compilação termine entretanto.

4.7 Variáveis

O interpretador de comandos possui variáveis, tal como uma linguagem de programação, mas todas do tipo cadeia de caracteres. Algumas variáveis são necessárias ao funcionamento do *shell* e encontram-se predefinidas logo de início. O comando `set`, sem argumentos permite visualizar as variáveis definidas em dado instante. (Em *C-shell* também existe o comando `setenv`, com uma sintaxe diferente.)

Definição de variáveis Em *shell* não existe distinção entre criar uma nova variável ou modificar o valor de uma variável já existente, bastanto para tal fazer uma normal atribuição, por exemplo

```
$ x=12
$ file=main.c
```

Notar que a variável `x` é uma cadeia de dois caracteres e não um inteiro, embora possam ser efecuadas operações aritméticas através do comando `expr`.

Valor de variáveis Numa atribuição, todos os valores que surjem do lado direito de uma atribuição (sinal de '=') são calculados e atribuídos à variável do lado esquerdo, tal como em qualquer linguagem de programação. Para obter o valor de uma variável, mesmo sem ser uma atribuição, é necessário precedê-la do carácter '$', por exemplo

```
$ old_x=$x
$ x=‘expr $x + 1‘
```

Para evitar ambiguidades, pode ser necessário incluir o nome da variável entre chavetas, em especial se o nome for seguido de caracteres especiais. Se for o caso, usar **${x}** em vez de **$x**.

Substituição condicional de valores Caso uma variável não esteja definida então o seu valor é a cadeia de caracteres vazia. Quando se pretende determinar se a variável já existe ou utilizar um valor por omissão, caso ainda não exista, utiliza-se a substituição condicional. Assim, **${x-12}** usa o valor **12** sempre que a variável **x** não exista (continuando **x** a não existir). No caso de se usar **${x=12}**, a variável, caso não exista, é criada com o valor indicado, que é devolvido.

Por último, **${x?nada feito}** testa a existência da variável e caso não exista o programa termina com a impressão da mensagem indicada. O comando **:**, que não faz nada (comando nulo), pode ser útil para testar a existência de variáveis necessárias à execução do programa. Por exemplo,

```
$ : ${CVSROOT?defina o repositório} ${EDITOR?defina o editor}
```

Variáveis de ambiente Algumas destas variáveis podem ser tornadas globais, através do comando `export`, podendo ser acedidas dentro das aplicações (por exemplo, a função `getenv()` da linguagem **C**). Estas variáveis são designadas por variáveis de ambiente. Em *csh* estas variáveis são criadas com: `$ setenv nome valor`.

Apenas em *shell*, uma variável pode ser passada como variável de ambiente num comando, sem necessidade alterar o seu valor anterior nem de a tornar global a todas as aplicações, por exemplo

```
$ x=12
$ x=20 use_x
$ echo $x
12
```

o comando `use_x` recebe a variavel de ambiente `x` com o valor 20, mas no interpretador de comandos ela mantém o seu valor e o seu estado (de ambiente ou não).

Quoting de variáveis Os caracteres de *quoting*, atrás descritos, apresentam comportamentos distintos quando se trata de obter o valor de variáveis: as aspas permitem a determinação do valor das variáveis, enquanto as plicas impedem essa determinação. Por exemplo,

```
$ echo x="$x" ou '$x'
x=12 ou $x
```

4.8 Programação

O interpretador de comandos funciona como uma linguagem de programação comum, mas a maioria dos comandos disponíveis resulta da invocação de aplicações em processos autónomos. Tal facto torna a programação em *shell* muito flexível (todas as aplicações disponíveis são potenciais instruções da linguagem) mas lenta, já que a criação de processos é dispendiosa.

4.8.1 Condições lógicas

A determinação de condições lógicas é efectuada com base no valor de retorno do último processo executado, sendo considerado verdadeiro se a aplicação retornou 0 (zero) e falso para qualquer outra situação. Notar que quando uma aplicação termina através de uma interrupção, com ou sem geração do ficheiro *core*, o código de terminação é igual ou superior a 128.

Literais booleanos Os literais booleanos, `verdadeiro` e `falso`, podem ser obtidos a partir de processos que retornem, respectivamente, 0 (zero) e diferente de zero (por exemplo, 1). Tal é o caso dos programas `/bin/true` e `/bin/false`, que são eles próprios escritos em *shell*.

Comando *test* O comando `test`, que pode ser abreviado para [e terminado por], permite efectuar testes a ficheiros e comparações entre cadeias de caracteres e valores inteiros. Tal como nos outros comandos, a leitura do respectivo manual é essencial, pois este documento apenas apresenta o comando e alguns exemplos.

A conjunção, disjunção e negação de condições lógicas é efectuada com as opções **-a**, **-o** e **!**, respectivamente. Por exemplo,

```
$ test -f main.c -o -x a.out -a -d exemplos
$ test -z $x -o $x != "string"
$ [ $n -gt 0 -a $n -le 12 ]
```

onde o primeiro exemplo verifica se: o ficheiro `main.c` existe (e é regular) ou se o ficheiro `a.out` é executável e a directoria `exemplos` existe. No segundo exemplo, verifica-se se a cadeia de caracteres, indicada pela variável **x**, está vazia ou se a cadeia de caracteres é diferente da constante literal `string`. No último exemplo, verifica-se se o valor da variável **n** está compreendido entre 1 e 12, inclusivé.

Claro que é possível misturar na mesma condição verificações de ficheiros, cadeia de caracteres e valores inteiros, bem como usar muitos outros tipos de testes não exemplificados.

Finalmente, o valor devolvido por um comando, seja uma condição lógica ou não, pode ser acedido através da variácvel **$?** (ou **$status** em *csh*). Por exemplo,

```
$ n=12
$ [ $n -lt 0 ]
$ echo $?
1
$ echo $?
0
$
```

o resultado da comparação é falso, logo **$?** é diferente de zero, mas o comando `echo` (que imprime esse valor) termina com sucesso, logo **$?** já vale zero no comando seguinte.

4.8.2 Construções sintácticas

O *shell*, como linguagem de programação permite as construções habituais:

if *lista* **then** *lista* { **elif** *lista* **then** *lista* } [**else** *lista*] **fi**
while *lista* **do** *lista* **done**
for *lista* [**in** *palavra* { *palavra* }] **do** *lista* **done**
case *palavra* **in** { *palavra* { **|** *palavra* }) *lista* **;;** } **esac**
nome **() {** *lista* **; }**

onde as palavras reservadas são representadas a negrito, os elementos opcionais entre parênteses rectos e as repetições (zero ou mais vezes) entre chavetas.

Na utilização das construções sintácticas pode-se escrever tudo na mesma linha, com os elementos separados por **;**

```
$ x=1; while [ $x -lt 15 ]; do echo $x; x=‘expr $x + 1‘; done
```

ou substituir os separadores **;** por mudanças de linha

```
$ x=1
$ while [ $x -lt 15 ]
>    do echo $x
>       x=‘expr $x + 1‘
>    done
```

a indentação utilizada não é necessária, mas ajuda a legibilidade do código.

4.8.3 Programas

Para construir programas executáveis, como em outras linguagens, basta incluir a sequência de comandos num ficheiro, que na primeira linha indica o nome do interpretador de comandos a utilizar, antecedido de **#!**. Notar que os programas são interpretados a partir da descrição textual, pelo interpretador de comandos, não havendo nenhum tipo de compilação ou geração de ficheiros binários.

Como não existem restrições, em *unix*, em relação ao nome ou localização do interpretadores de comandos (excepção aos utilizados no *login*), qualquer programa pode funcionar como interpretador de comandos. Assim, qualquer interpretador de uma linguagem, nova ou existente, pode ser invocado pelo mesmo mecanismo. Por exemplo, um interpretador para a linguagem **C** residente em `/home/user/bin/ic` (interpretador de **C**) é automaticamente invocado sempre que é executado um ficheiro (`main.c`)que inclua na primeira linha '`#! /home/user/bin/ic`'. Neste caso, o ficheiro `main.c` deverá ter previlégios de leitura e execução (`$ chmod a+rx main.c`), podendo ser invocado como qualquer outro comando: `$ main.c`. (Existe um compilador *load-and-go*, designado por **tcc** que faz mais ou menos o mesmo.)

Comentários Embora podendo ser utilizados no interpretador de comandos em modo interactivo, os comentários são especialmente úteis na documentação de programas. Um comentário inicia-se no carácter **#** e termina no fim da linha. Por exemplo,

```
#! /bin/sh
# programa que imprime os valores de 1 a 12
x=1 # define o valor inicial
while [ $x -lt 12 ]; do # ciclo
   echo $x # imprime o valor de x
   x=‘expr $x + 1‘# incrementa x de uma unidade
done
```

Parâmetros A utilização de parâmetros na invocação dos comandos é possível a partir de variáveis predefinidas. Por exemplo **$1** representa o primeiro parâmetro, **$9** o nono, e **$0** o nome do programa invocado (tal como os valores do vector **argv** na linguagem **C**). Da mesma forma, a variável **$#** permite saber quantos argumentos existem (tal como a variável **argc** na linguagem **C**). Finalmente, a variável **$*** representa todos os parâmetros excepto o **$0**, separados por espaços.

Para aceder aos sucessivos argumentos existe o comando **shift** que desloca todos os parâmetros de menos uma unidade. Assim, o nono parâmetro passa agora a ser **$8** e perde-se o **$1**, ficando o **$0** sempre inalterado.

Por exemplo,

```
#! /bin/sh
# imprime os valores entre 2 argumentos (de 1 a 12 se omitidos)
if [ $# -le 1 ]; then x=1; else x=$1; shift; fi # valor inicial
if [ $# -le 1 ]; then fim=12; else fim=$1; fi # valor final
while [ $x -le $fim ]; do # ciclo
   echo $x # imprime o valor de x
   x=‘expr $x + 1‘# incrementa x de uma unidade
done
```

Rotinas A utilização de rotinas segue a descrição sintáctica acima (ver 4.8.2), não sendo indicados parâmetros. Tal facto deve-se, em primeiro a todos os parâmetros serem do tipo cadeia de caracteres e, em segundo, o acesso aos parâmetros ser feito por posição (como nos programas).

```
#!/bin/sh
find_and_count()
{
  if ‘grep -q main $1‘
    then
      wc -l $1
    fi
}

for i in ‘find . -name "*.c" -print‘
  do
    find_and_count $i
  done
```

Comando *read* Para ler valores (cadeias de caracteres) do terminal existe o comando **read**, que lê uma linha completa. Para processamento de ficheiros de texto, conjuntos de linhas, aconselha-se o uso de programas mais elaborados como: **awk**, **sed**, *etc.*

Comando *eval* O comando `eval` permite interpretar o conteúdo de uma cadeia de caracteres como se de um programa se tratasse. Por exemplo,

```
#!/bin/sh
read x && eval $x
echo ${y-não existe}
```

só executa o comando `eval` caso tenha conseguido ler uma linha de texto para a variável **x**. Na última linha é impresso o valor da variável **y**, que não é referida no programa. Assim, só será impresso o valor de **y** se na linha digitada pelo utilizador, durante a execução do programa, existir uma atribuição à variável **y**. Por exemplo, se o programa acima se chamar **xy**:

```
$ ./xy
x=12
não existe
$ ./xy
y=12
12
$
```

4.8.4 Detecção de erros

Não existe um depurador (*debugger*) em *shell* pelo que existem opções que provocam a escrita de mensagens à medida que o programa é executado. Estas opções podem ser incluídas no programa através do comando **set** (por exemplo, `set -v`) ou ser directamente passadas ao *shell* no momento da invocação do programa (por exemplo, `sh -v prog.sh`).

As opções mais importantes incluem **-v** (*verbose*) que imprime os comandos que vão sendo executados, sendo particularmente útil na procura de erros sintácticos, e **-x** que imprime os valores que vão sendo calculados, logo mais orientada para erros semânticos.

A detecção de erros pode ser desligada através de **set -** e as opções activas são indicadas pela variável **$-**.

4.8.5 Tratamento de excepções

Tal como nos programas escritos em linguagens tradicionais, é possível enviar e receber interrupções, não só entre partes de um programa escrito em *shell*, como de ou para programas compilados tradicionais. O envio de uma interrupção usa o comando **kill**, enquanto a recepção de uma interrupção usa o comando **trap**. As interrupções disponíveis dependem do sistema operativo, mas são em geral 32 e podem ser listadas com **kill -l**.

Para o envio de uma interrupção indica-se a interrupção, utilizando o seu nome ou número, e uma lista de PIDs. Notar que **$$** é o número do processo corrente e **$!** o número do último processo enviado para execução paralela (*background*).

Na recepção de interrupções indica-se o comando a executar, usar plicas se for mais de um, e uma lista das interrupções a receber. Quando a interrupção é enviada ao processo, quer por *hardware* (por exemplo, *bus error*, *segmentation fault*, *illegal instruction*, *floating point exception*) quer por *software* (por exemplo, qualquer das interrupções de *hardware* bem como *interrupt*, *hangup*, *quit*, *abort*, *software termination*, *etc.*) o comando é executado. Se o comando for vazio a interrupção é ignorada. Existem algumas interrupções, em especial o *kill* (número 9), que não pode ser apanhada ou ignorada, permitindo retirar de memória qualquer processo. Por exemplo,

```
trap '' 2 # ignorar ^C
trap 'rm tmpfile; echo saindo!' 11 15
kill -HUP $!
```

4.9 Ambiente

O ambiente de execução do interpretador de comandos é constituído pelo conjunto de variáveis definidas e de comandos acessíveis, em determinado instante.

Algumas variáveis são predefinidas:

$? ($status em *csh*)	valor de terminação do comando anterior
$$	PID deste *shell*
$!	PID do último comando ainda em execução num sub-*shell*
$-	opções de invocação do *shell* e alteradas pelo comando **set**
$#	número de parâmetros posicionais
$*	todos os parâmetros posicionais
$4 ou $12	**n**-ésimo parâmetro posicional

4.9.1 Variáveis mais comuns

A variável de ambiente **PATH** contém uma lista de directórios, separados por **:**, onde são procurados os programas a executar. Notar que muitas vezes o directório corrente não é incluído na variável **PATH**, obrigando a preceder os comandos no directório corrente do prefixo **./**. Para evitar tal procedimento basta incluir o directório **.** na variável **PATH**: no fim da lista para não encobrir comandos de sistema (`$ PATH=${PATH}:.`); no início da lista para evitar o comportamento inverso (`$ PATH=.:$PATH`). Notar que a **root** nunca deve incluir *pathname*s relativos na variável **PATH**, por forma a impedir a execução de programas de utilizadores com previlégios de administrador.

A variável de ambiente **HOME** determina o directório de *login* do utilizador, face ao qual inúmeros ficheiros são acedidos (por exemplo, `${HOME}/.cshrc`). Em muitos dos interpretadores de comandos, o directório indicado por `$HOME` pode ser abrevidado para `~/`. Da mesma forma, o directório de *login* de outro utilizador por ser referido intrepondo o nome de *login* do utilizador entre o `'~'` e `'/'`, por exemplo `~root/.bashrc`

A variável de ambiente **PS1** contém o valor da *prompt* primária e **PS2** contém o valor da *prompt* secundária (utilizada para indicar construções incompletas). Por exemplo,

```
$ PS="olá "; PS2="mais "
olá
```

A variável de ambiente **IFS** contém a lista de separadores, que inclui o carácter de mudança de linha (não necessariamente o *CR*), o carácter espaço branco ou o tabulador horizontal. A alteração dos valores desta variável permite processar ficheiros com outros delimitadores. Contudo, a sua alteração inadvertida pode tornar o interpretador de comandos inutilizável.

4.9.2 Ficheiros de iniciação

Antes de iniciar a execução, o interpretador de comandos executa um conjunto de ficheiros de configuração. O primeiro a ser executado é global ao sistema operativo e encontra-se, em geral, em `/etc`.

Sob o controlo do utilizador, na sua directoria de *login* (**HOME**), existe o ficheiro de iniciação `.profile`, ou no caso dos *shell* mais recentes: o nome do *shell* seguido de **rc** (`.bashrc`, `.cshrc`, `.tcshrc`, `.zshrc`, *etc.*) O ficheiro de iniciação inclui a declaração de variáveis de ambiente necessárias às aplicações utilizadas e definição de macros, por utilizador.

Além deste ficheiro, no caso de se tratar de um *shell* de *login* (*shell* criado após a introdução da palavra passe, *password*) é processado um ficheiro de *login*, designado por `~/.bash_profile` (ou `~/.login` em *csh*). Este ficheiro é executado após o ficheiro de iniciação atrás falado, e permite definir variáveis de ambiente que subsistem para os sub-*shell*s depois criados ou executar programas de *login*.

Também existe um ficheiro, que a existir, é executado quando o interpretador de comandos termina: `.logout` ou `.bash_logout`.

Execução de comandos no ambiente Uma vez que os programas escritos em *shell* executam em sub-processos, as variáveis criadas ou modificadas são destruídas na terminação do processo. Para fazer o *shell* actual executar os comandos e reter os valores é necessário invocar o programa precedido do comando **.** (ou **source** no caso do *csh*).

4.10 Comandos comuns

Nesta secção abordaremos muito brevemente alguns dos comandos mais comuns, por ordem alfabética, para que o utilizador tenha conhecimento da sua existência. Estes comandos dispoem de inúmeras opções que podem ser consultadas nas respectivas páginas de manual, disponíveis *online*. Só com um conhecimento específico dessas opções é possível tirar partido destas aplicações.

ar	cria e gere bibliotecas de rotinas compiladas
as	montador (assembler): gera objectos a partir de descrições assembly
awk	procura formatada com base em expressões regulares
basename	devolve o nome do ficheiro(*pathname*) sem directoria (ou extensão)
bc	calculadora básica infixada (notação matemática)
cal	calendário
cat	mostra o conteúdo de um ficheiro
cc	compilador para a linguagem **C**
cd	muda a directoria corrente
chgrp	altera o grupo de um ficheiro
chmod	altera as protecções de um ficheiro
chown	altera o dono (e o grupo) de um ficheiro
cmp	compara 2 ficheiros binários (não de texto)
cvs	programa de controlo de versões
date	obtém e modifica a data de sistema em diversos formatos
dc	calculadora postfixada
df	verifica o espaço disponível nos diversos periféricos montados
diff/diff3	compara 2 ou 3 ficheiros de texto

dirname	devolve a directoria de um ficheiro(*pathname*)
echo	imprime os seus argumentos
ed/sed	editor de linha original do unix
emacs	editor visual completo e complexo
exit	termina o processo corrente
expr	efectua operações aritméticas sobre números inteiros
false	devolve sempre a condição falsa
find	procura ficheiros por nome, data, tipo, *etc*.
gdb/dbx	depurador (debugger) simbólico
grep	procura cadeias de caracteres em ficheiros de texto
gzip	compressor de ficheiros (existem outros)
head	devolve as primeira(s) linhas de um ficheiro
jobs	mostra os processos associados ao terminal
kill	envia interrupções a processos
ld	carregador (loader): cria executáveis a partir de objectos e bibliotecas
less	visualizador do conteúdo de um ficheiro (permite buscas e saltos)
ln	cria nomes alternativos para os ficheiros
lpr	envia ficheiros para a impressora (ver também lpq, lprm, lpc)
ls	mostra o conteúdo de um directório ou ficheiros específicos
make	programa de controlo de configurações
man	(help) acesso *online* aos manuais (começar com **$ man man**)
(u)mount	(des)liga sistemas de ficheiros uns aos outros
newgrp	altera o grupo corrente do utilizador
nice/renice	altera a prioridade de execução dos processos
nm	mostra símbolos, seu tipo e posição em ficheiros objecto binários
od	imprime o conteúdo do ficheiro em inteiros de diferente base e tipo
ps	lista os processos em execução
pwd	devolve a directoria corrente
read	lê uma linha de texto do terminal
sleep	espera o número de segundos indicados
sort	ordena alfabeticamente ou numeriucamente linha de texto
stat	mostra informação de controlo de um ficheiro
sync	escreve a informação em *cache* e memória no disco
tail	devolve as última(s) linhas de um ficheiro
tar	arquivador de ficheiros (existem outros)
tee	envia para um ficheiro os dados transferidos numa *pipe*
test/[	calcula expressões lógicas
time	imprime estatísticas de execução de um processo
trace/strace	imprime as chamadas aos sistema efectuadas por um processo

trap	recepção de uma interrupção
tr	substitui ou elimina certos caracteres de um fichero
true	devolve sempre a condição verdadeira
ulimit	impõe limites aos processos do utilizador
umask	defina as protecções com que os ficheiros são criados
uniq	elimina linhas iguais consecutivas
vi	editor visual completo original do unix
wait	aguarda pela terminação de processos
wc	conta caracteres, palavras e linhas

5 CVS: gestão de versões

Um sistema de gestão de versões mantém o registo das alterações actualizadas (commited) de cada ficheiro. Assim, é possível recuperar uma versão anterior, devido a um erro cometido entretanto, ou caso pretenda explorar outras alternativas. As diversas versões são guardadas incrementalmente, ou seja, apenas as diferenças entre cada duas versões consecutivas, poupando espaço em disco. O CVS mantém o controlo de projectos constituídos por diversos ficheiros em várias directorias. Para tal regista as versões de cada ficheiro do projecto sempre que são declaradas alterações. Posteriormente, é possível repor os diversos ficheiros de um projecto nas versões respectivas em certa data. Para facilitar a identificação das alterações é registada, além da data da alteração, o utilizador e uma descrição, podendo atribuir-se um nome lógico sempre que se justifique. No CVS cada utilizador trabalha numa área independente, geralmente designada por *sandbox*, só se actualizando a base de dados comum quando atinge uma versão estável. Se ao actualizar a base de dados surgirem conflitos, por outros utilizadores terem executado alterações na mesma zona do ficheiro, é possível resolver o conflito com a ajuda dos algoritmos disponibilizados.

5.1 CVSROOT

O sistema de gestão de versões é acedido através de um único comando, designado por **cvs**, cujo primeiro argumento indica a operação, seguido dos argumentos dessa operação, caso existam. Antes de utilizar o CVS torna-se necessário definir a variável de ambiente CVSROOT, que designa a localização da base de dados, para não ter de indicar a sua localização em cada comando **cvs**, através da opção '-d'.

A utilização do CVS necessita que o *shell* tenha definida a variável de ambiente CVSROOT, cujo valor deve indicar o caminho absoluto (a partir da raiz do sistema de ficheiros) da base de dados comum, por exemplo o subdirectório *cvs* no directório de *login*:

- em *sh* (bash, ksh, ...):

  ```
  $ export CVSROOT=$HOME/cvs
  ```

- em *csh* (tcsh, zsh, ...):

  ```
  $ setenv CVSROOT ~/cvs
  ```

Para que a variável fique definida em cada novo terminal, a instrução acima deve ser introduzida no ficheiro de iniciação do *shell* respectivo na directoria de *login*

($HOME/.bashrc , por exemplo). Em windows a variável de ambiente pode ser definida com o comando `set`, em vez de `export`, e guardado em *Control Panel*, *System*, *Advanced*, *Environment Variables*.

5.2 Criação da base de dados comum

Esta operação só necessita ser efectuada uma vez. Notar que é possível aceder a uma base de dados comum por *nfs*, *rsh* ou *ssh*, além de um protocolo proprietário por *internet*. Além disso, uma base de dados comum pode manter o controlo de diversos projecto distintos, por exemplo, trabalhos de disciplinas distintas.

```
$ cvs init
```

5.3 Criação de um projecto

Esta operação é efectuada quando se inicia um novo projecto, pela primeira vez. Para tal é necessário um conjunto inicial de ficheiros, ou no mínimo um directório vazio. A criação do projecto é executada no directório principal do projecto, sendo colocados no CVS todos os ficheiros contidos no directório corrente, bem como subdirectórios e respectivos ficheiros.

No caso mais simples, sem um único ficheiro inicial:

```
$ mkdir comp
$ cd comp
$ cvs import -m ‘‘versão inicial’’ projecto disciplina inicial
```

onde ‘projecto’ é o nome do projecto, ‘disciplina’ é a identificação do vendedor (*vendor-tag* e ‘inicial’ o nome da distribuição a introduzir no CVS.

Caso existam ficheiros ou uma estrutura de directórios inicial, estes devem ser colocados sob um directório comum de onde o comando é executado:

```
$ mkdir aplic
$ cd aplic
$ tar zxvf ../compact-src-1.2.tgz
$ cvs import -m ‘‘distribuição v1r2’’ compact comp05  distrib
$ cd ..
$ rm -rf aplic
```

no fim os ficheiros introduzidos no CVS devem ser apagados, por forma a não serem confundidos com as cópias de trabalho posteriormente obtidas do CVS. Contudo, é conveniente guardar esses ficheiros iniciais numa *pen*, *cd-rom*, ou agrupadas e comprimidas num directório específico.

5.4 Evolução de um projecto

Para inspeccionar o projecto cria-se uma *sandbox* local:

```
$ cvs checkout compact
$ cd compact
$ less compact.y
$ cd ..
$ cvs release -d compact
```

o comando 'checkout' cria um subdirectório com o nome do projecto e preenche-o com os ficheiros e directórios que o constituem. O comando 'release' pretende cancelar a operação do comando 'checkout'. Embora este comando não seja estritamente necessário, pois o CVS não bloqueia (*lock*) os ficheiros logo estes podem ser apagados convencionalmente, evita que sejam esquecidas alterações entretanto efectuadas. A opção '-d' no comando 'release' apaga o directório do disco, permanecendo a cópia original no repositório.

Para introduzir modificações nos ficheiros basta criar a *sandbox* e editar os ficheiros. Quando se atingir uma versão estável e se pretende registá-la, por forma a que os restantes utilizadores ou posteriormente nós próprios tenhamos acesso, basta executar uma actualização (*cvs commit*). O comando admite a opção **-m "descrição das alterações"** , ou na sua ausência invoca o editor de texto para que as alterações sejam descritas. Estas mensagens permitirão posteriormente identificar a versão, caso a data não seja suficiente para avivar a memória.

Quando se criam novos ficheiros e se pretende que estes façam parte do sistema de gestão de versões deve ser executada uma adição (*cvs add*). Da mesma forma, se um ficheiro deixa de ser utilizado deve ser retirado (*cvs remove*), embora continue a existir em versões anteriores. A alteração do nome de um ficheiro, ou directório, deve ser declarada com uma adição do novo nome e uma remoção do nome anterior. Notar que as alterações só fazem efeito quando se realiza uma actualização, aliás como os respectivos comandos de 'add' e 'remove' informam.

Caso exista mais de um utilizador a aceder ao projecto em simultâneo, pode ser necessário incluir na *sandbox* as alterações entretanto actualizadas na base de dados pelos

restantes utilizadores, por forma a incorporar essas modificações na versão em mãos. Para tal, o comando 'update' compara a base de dados com os ficheiros existentes na *sandbox* e introduz na *sandbox* os ficheiros entretanto modificados na base de dados. Caso existam conflitos, por exemplo a mesma linha de código do mesmo ficheiro foi alterada para conteúdos distintos na *sandbox* e na base de dados, o utilizador é informado do facto e o ficheiro é modificado por forma a conter ambas as alterações. Neste caso, o utilizador deverá decidir qual o resultado pretendido através da edição do ficheiro em causa.

```
$ cvs checkout compact
$ cd compact
$ vi compact.y
$ mv tri.cpt triangulo.cpt
$ rm ex5.cpt
$ cvs add triangulo.cpt
$ cvs remove tri.cpt ex5.cpt
$ cvs update
$ vi compact.l
$ cvs commit
$ vi ex.cpt
$ cvs release .
```

5.5 Identificação dos ficheiros

O CVS controla projectos, ou seja, conjuntos de ficheiros que se podem espalhar por diversos directórios e sub-directórios. No entanto, o controlo das diferentes versões de cada ficheiro individualmente é efectuado pela ferramenta RCS. O RCS é também um sistema de controlo de versões, mas onde o utilizador necessita controlar quais as versões de cada ficheiro que constituem determinada versão do projecto. Por esta razão, a sua utilização directa torna-se difícil de gerir quando o projecto é composto por diversos ficheiros.

O RCS mantém apenas a cópia integral da última versão do ficheiro, registando apenas as linhas modificadas para a versão anterior, e desta para a que a antecedeu, *etc.* Desta forma, o espaço ocupado é reduzido e o tempo de recuperação de uma versão aumenta com a sua antiguidade. Se atendermos a que as últimas versões serão, certamente, as mais frequentemente acedidas tal decisão não é limitativa. Por outro lado, todas as versões de um ficheiro são mantidas num mesmo ficheiro RCS, pelo que quando este é copiado ou transportado leva consigo todas as versões desde a sua criação.

O controlo individual de cada ficheiro é efectuado indirectamente pelo RCS, isto é, o utilizador nunca invoca nenhum comando RCS directamente quando está a utilizar o CVS. No entanto, como a informação de cada ficheiro é gerida pelo **rcs**, toda a informação de controlo de versões que pode ser colocada num ficheiro é definida pelo RCS. Esta informação permite incluir, por exemplo o número da versão actual do ficheiro, numa cadeia de caracteres ou numa directiva '#if' para controlar a compilação.

Esta informação pode ser utilizada para identificar os ficheiros, mesmo depois de compilados, permitindo identificar quais as versões dos ficheiros fonte que deram origem ao executável. Por exemplo,

```
static char id[] = "$Header\$";
```

irá criar uma cadeia de caracteres, designada pela variável **id**, que será substituída em cada `checkout` pela identificação do ficheiro. Esta identificação inclui a localização do ficheiro original do repositório (isto é, independentemente do local para onde o `checkout` foi efectuado), a data e hora da criação da versão em uso e o nome do utilizador que a criou. Notar que esta informação pode ser obtida através do comando **ident** (não é um comando CVS pelo que não tem o prefixo `cvs`), mesmo que o ficheiro a **ident**ificar seja um ficheiro objecto ou executável.

A informação disponível é identificada por macros, além de '$Header$', e que incluem:

$Author$: nome de 'login' do utilizador que criou a versão.

$Date$: dia e hora em que versão foi criada.

Id : o mesmo que '$Header$' mas o nome do ficheiro não inclui o directório onde reside, apenas o nome do ficheiro.

Log : a mensagem de registo da versão.

$Revision$: número da versão.

$Source$: nome completo do ficheiro, incluindo o directório onde reside.

5.6 Extrair informação do projecto

Para retirar cópias dos ficheiros sob o controlo de versões existe o comando 'export'. Este comando é semelhante ao 'checkout', mas não retira a informação de controlo do CVS sendo mais aconselhável para distribuição por clientes. Para que a versão distribuída possa ser devidamente identificada, para posteriores correcções ou melhoramentos, é necessário atribuir a essa versão um nome lógico (tag), antes de a extrair. A colocação do nome lógico é efectuado pelo comando 'rtag' para um projecto no sistema de versões ou 'tag' para um conjunto de ficheiros previamente retirados (checkout) do sistema de versões.

```
$ cvs rtag ok-1_0 compact
$ cvs export -r ok-1_0 compact
$ tar cf - compact | gzip -9 > compact-1_0.tgz
$ rm -rf compact
```

O comando 'rdiff' permite obter as diferenças entre todos os ficheiros de duas versões, indicada nos argumentos do comando. O ficheiro resultante tem o formato de um *patch file*, e contém apenas as diferenças entre as duas versões, cuja dimensão é, em geral, muito inferior à da totalidade dos ficheiros do projecto. Desta forma pode ser aplicado a ficheiros previamente extraídos, da primeira versão, para produzir a segunda versão. Tal como no comando 'tag', também existe um 'rdiff' para um projecto e um 'diff' para um conjunto de ficheiros previamente retirados (checkout) do sistema de versões. As versões podem ser identificadas por nomes simbólicos ou por datas, podendo estas ser absolutas (ano-mês-dia hora:minuto) ou relativas:

```
$ cvs rdiff -D ‘‘3 hours ago’’ -r ok-1_0 compact > patch3-1.0
$ mail -s ‘‘as últimas 3 horas’’ user@mega < patch3-1.0
```

5.7 Consulta dos registos

Existe um conjunto de comandos que permite analisar o funcionamento do CVS, consultar as operações realizadas, quem as executou, quando e a partir de onde. Estes comandos dispõem de uma extensa lista de opções pelo que a consulta do manual torna-se quase indispensável.

O comando 'history' permite obter listagem das operações de 'commit', 'checkout', etc., consoante as opções indicadas. Por exemplo, para obter as datas, utilizador e versões geradas do ficheiro 'compact.l':

```
$ cvs history -c compact.l
```

O comando 'log' permite obter informação sobre as operações efectuadas individualmente sobre cada ficheiro pelo RCS. O RCS é o subsistema do CVS que permite controlar as versões de cada ficheiro individualmente, a que o CVS acrescenta a possibilidade destes se poderem distribuir por directorias e controlar os conjuntos de versões de ficheiros individuais que constitui uma versão de projecto. Desta forma os comandos CVS actuam sob conjuntos de ficheiros coerentes, os projectos. Embora o RCS possa ser utilizado individualmente como aplicação independente, os ficheiros sob o controlo do CVS não podem ser directamente manipulados pelo RCS. O comando de 'log' transmite a informação guardada pelo RCS que, por ser privada de cada ficheiro, não necessita ser gerida pelo CVS. Por exemplo, para obter não apenas uma listagem de datas, utilizadores e versões, mas para saber quantas linhas foram modificadas e qual a mensagem associada ao 'commit' que afectou essa versão:

```
$ cvs log -b compact.l
```

O comando 'admin' permite efectuar operações de gestão muito específicas como manipulação de informação nos ficheiros RCS, definição do sistema de 'locking', etc. As aplicações deste comando saem fora do âmbito deste documento pelo que se dirige o leitor para o manual do CVS, quando este já tiver dominado o funcionamento do CVS.

5.8 Recuperação de versões anteriores

A recuperação de versões anteriores permite ter acesso a antigos estado do projecto, podendo inspeccionar código ou ficheiros completos entretanto apagados. As versões anteriores podem ser recuperadas por etiqueta (`tag`) ou por data, usando as opções **-r** ou **-D** respectivamente. Os formatos de data aceites pelo CVS são diversos, mas conselha-se a norma ISO 8601 (data CEE), ou seja, **ano-mes-dia hora:minuto:segundo** ou **ano-mes-dia** usando apenas números e representando o ano com os 4 digitos.

Na recuperação de versões anteriores pode-se criar um novo espaço de edição (*sand-box*) com o comando **cvs checkout**, ou actualizar o espaço de edição corrente com o comando **cvs update**. Da mesma forma, as mesmas opções podem ser aplicadas a muitos outros comandos CVS, por exemplo, **diff**, **export**, **history**, **commit**, *etc*.

Para recuperar uma versão específica deve-se, previamente, identificar essa versão mediante um comando de consulta de registos como **history** ou **log** (ou o comando **cvspast**).

5.9 Desenvolvimento concorrente

O CVS permite que sejam criados vários ramos sendo possível trabalhar em cada um deles concorrentemente. Esta capacidade permite que um utilizador possa efectuar estudos exploratórios independentemente dos restantes utilizadores do projecto, podendo numa fase posterior integrar as suas alterações no ramo principal, se assim o entender.

Para criar um novo ramo usa-se a opção **-b**, em geral nos comandos **tag** ou **rtag**. A versão a partr da qual se cria o ramo não necessita ser a última, devendo ser referenciada por etiqueta ou data.

```
$ cvs rtag -b -rAntiga Nova Projecto
```

Depois de criado o ramo, este pode ser acedido pelos comandos habituais, onde a opção **-rNova** designa o ramo recém criado.

```
$ cvs checkout -rNova Projecto
```

Todos as actualizações, através do comando **commit**, executadas nesta directoria re-fe–rem-se ao novo ramo.

Por outro lado, caso se pretenda que as alterações efectuadas não sejam introduzidas no ramo corrente mas num novo ramo, evitando *contaminar* a evolução do projecto corrente em uso por outros utilizadores, pode-se etiquetar as alterações directamente para um novo ramo.

```
$ cvs tag -b Nova2
$ cvs update -rNova2
```

Junção de ramos Caso se deseje que as modficações feitas num ramo sejam incluídas noutro, por exemplo o ramo principal, usa-se a opção **-jetiqueta** ou **-jetiqueta:data**. No último caso a **data** permite limitar a versão a usar na junção. Por exemplo,

```
$ cvs checkout -rAntiga Projecto
$ cd Projecto
$ cvs update -jNova2
```

junta as alterações do ramo `Nova2` no ramo `Antiga` do `Projecto`. O comando pode incluir uma lista de ficheiros, ficando a operação limitada aos ficheiros indicados.

5.10 Utilização remota

Uma vez que existe um só repositório este deve estar colocado numa máquina de fácil acesso. Na situação mais simples, o acesso aos ficheiros é feito fazendo *login* nessa máquina ou utilizando um protocolo de acesso a ficheiros em rede, por exemplo, **nfs**. Quando tal não é possível o CVS permite dois tipos de acesso alternativo, por *shell* e através de um protocolo próprio. Nesta situação, os utilizadores podem trabalhar cada um em sua máquina partilhando um repositório comum.

No acesso por *shell*, utiliza-se um programa que permita a execução remota de comandos. Por omissão o CVS usa o `rsh`, mas por razões de segurança aconselha-se o `ssh`. Para que o acesso ao servidor seja possível é necessário permitir o acesso na conta do utilizador no servidor. A autorização de acesso é colocada no servidor no ficheiro **~/.rhosts** no caso do `rsh` e no ficheiro **~/.shosts** no caso do `ssh`, contendo o nome da máquina de onde é efectuado o acesso seguido do nome do utilizador que efectua o pedido nessa máquina. Por exemplo,

```
lab1.ist.utl.pt guest
pc8lab5.ist.utl.pt public
```

No cliente a variável de ambiente **CVSROOT** deverá indicar, além do directório onde se localiza o repositório, a máquina onde este se situa, bem como o nome do utilizador da máquina e a forma de acesso:

```
:method:user@host:/path
```

onde o *method* representa a forma de acesso, *user* o nome do utilizador, *host* o nome da máquina que tem acesso ao repositório, e **/path** o directório que constitui a base do repositório.

Os métodos de acesso podem ser:

local: o acesso é directo, isto é, dentro da própria máquina, e a componente *user@host* não tem significado. Este método de acesso foi aquele utilizado até ao momento.

server: usa uma versão do `rsh` integrada no CVS, podendo não estar disponível em todas as versões do CVS.

ext: usa um programa exterior para efectuar a ligação remota. Por omissão usa o `rsh`, mas pode ser modificado através da variável de ambiente **CVS_RSH**. A localização do directório onde reside o comando **cvs** na máquina remota pode ser definido pela variável **CVS_SERVER**, caso não esteja directamente acessível.

pserver: usa um protocolo próprio do CVS em que os utilizadores são definidos por repositório e podem não corresponder aos utilizadores da máquina.

Assim, o acesso remoto pode ser efectuado através de:

```
$ setenv CVS_RSH ssh
$ setenv CVSROOT :ext:prs@mega.ist.utl.pt:/home/prs/cvs
$ cvs checkout compact
```

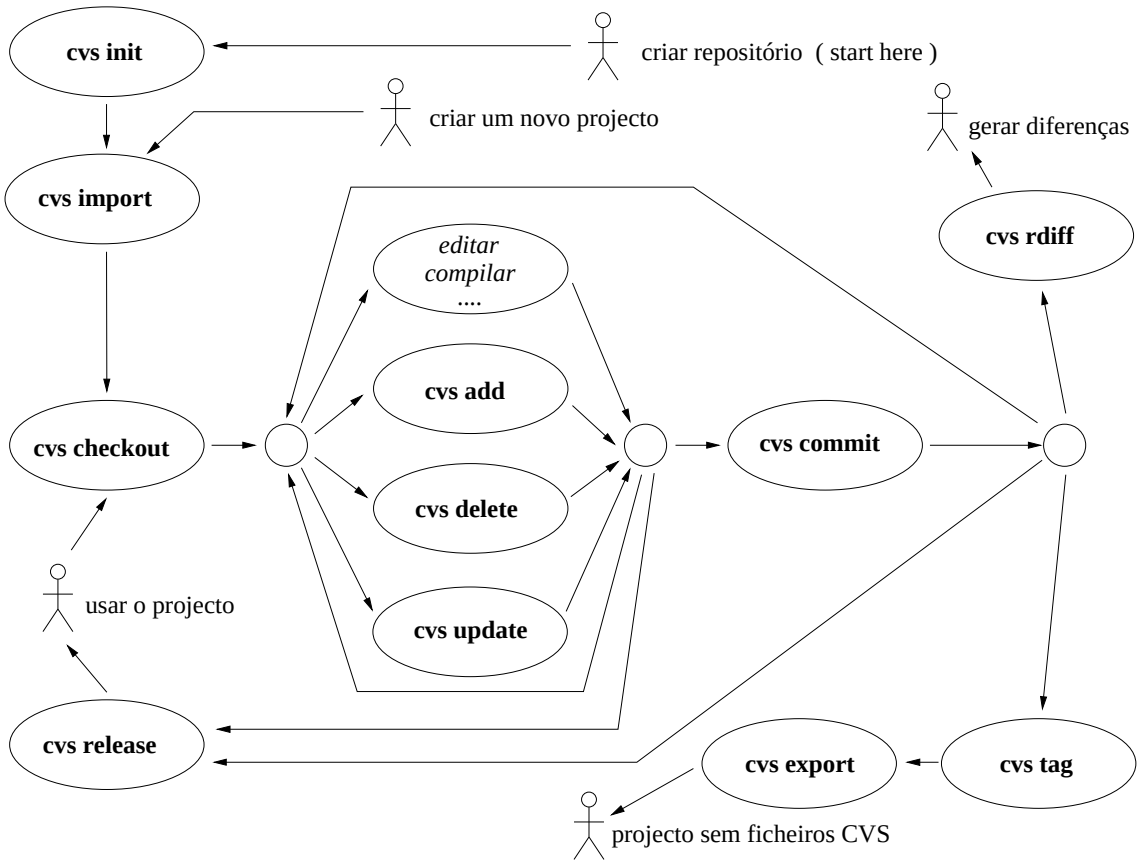
Modo pserver No modo `pserver` é necessaŕio utilizar uma palavra passe (*password*) para ganhar acesso ao repositório no servidor remoto. Para tal o utilizador deverá começar o acesso ao repositório com um comando `cvs login`, depois de ter definido o repositório e utilizador respectivo através do `CVSROOT`, indicando interactivamente a palavra passe previamente definida para esse utilizador.

Depois de ter efectuado os acessos ao repositório, através dos comandos habituais, é necessário efectuar o comando `cvs logout` para apagar as referências à palavra passe em `~/.cvspass`. Notar que se pode fazer `login` em vários servidores simultaneamente, sendo o servidor escolhido através do `CVSROOT` ou da opção **-d** dos comandos CVS.

Comando cvspass O comando **cvspass** não faz parte da distribuição do CVS, podendo ser obtido na página da disciplina. Este comando permite alterar a palavra passe que concede o acesso ao utilizador do repositório. Notar que este comando só produz o resultado desejado nos servidores que usem a extensão, como o servidor da disciplina.

5.11 Diagrama de utilização

O diagrama abaixo apresenta um fluxograma muito simplificado das principais operações do CVS num ambiente de desenvolvimento típico:


cvs init
criar repositório (start here)
criar um novo projecto
gerar diferenças
cvs import
cvs rdiff
editar
compilar
....
cvs add
cvs checkout
cvs commit
cvs delete
cvs update
usar o projecto
cvs release
cvs export
cvs tag
projecto sem ficheiros CVS

6 make: gestão de configurações

Durante a execução de um projecto muitos ficheiros são produzidos de forma automática a partir daqueles directamente editados pelo programador. O objectivo da aplicação **make** é determinar quais a partes do projecto que necessitam ser recompiladas, ou actualizadas, por ferramentas sempre que um ou mais ficheiros são modificados pelo programador.

O **make** é um programa que controla dependências entre ficheiros a partir de um conjunto de regras que lê de um ficheiro chamado **Makefile**. As regras estabelecem dependências entre a data da última modificação do **objectivo** face aos **dependentes**. Sempre que um dependente é alterado o objectivo necessita ser actualizado. Note-se que o **make** utiliza uma perspectiva pessimista pois a alteração no dependente pode não ter implicações no objectivo. Por exemplo, acrescentar um comentário não influencia o código gerado. No entanto, o mesmo comentário pode alterar a documentação, por exemplo, através do *javadoc*. Assim, o **make** não tira conclusões sobre o conteúdo dos ficheiros. Aliás, o **make** não abre sequer os ficheiros, limita-se a usar os seus nomes e respectivas datas de modificação.

O conteúdo de uma **Makefile** é, essencialmente, um conjunto de regras. Notar que o formato das regras é muito específico, devendo o nome do objectivo situar-se na primeira coluna e as operações são precedidas de um ou mais tabuladores horizontais (TABS). É necessário ter especial atenção na parte dos TABS pois é, por vezes, difícil distinguir entre um espaço e um TAB, mas antes da operação não podem existir espaços. Cuidado pois alguns editores de texto substituem espaços por TABS e vice-versa. Por outro lado, os separadores dos dependentes e dos argumentos das operações podem ser espaços ou TABS. Uma regra é da forma "**objectivo: [dependentes][; operações]**" ou numa apresentação mais legível e usual:

```
objectivo: dependente-1 ... dependente-N
< TAB > operacao-1
...
< TAB > operacao-M
```

Se a data do ficheiro **objectivo** for posterior (mais actual) que qualquer dos dependentes, o **make** não executa nada; se algum dos ficheiros **dependentes** tiver uma data mais recente que o **objectivo** então as operações são executadas sequencialmente até que alguma falhe, isto é, devolva um código de terminação diferente de 0 (zero). Se todas as operações devolverem 0 (zero), o **make** assume que o objectivo está actualizado em conformidade com os seus dependentes.

Os dependentes podem ser objectivos de outras regras. Desta forma, o resultado final pode ser especificado por regras que especificam cada um dos passos intermédios. Neste caso, apenas aqueles passos que necessitam ser repetidos são efectivamente executados. Quando o número de ficheiros envolvidos é grande, uma vez que as dependências não são de todos com todos, o número de operações executadas pode ser significativamente menor que numa abordagem tradicional.

Por exemplo, num ambiente de compilação separada em **C**, quando um ficheiro fonte é modificado, apenas esse ficheiro necessita ser compilado (gerando o respectivo **.o**) e junto aos restantes **.o** já compilados. No mesmo exemplo, caso se modifique um ficheiro de declaração (*header file*, ou **.h**), apenas os ficheiros fonte que o incluem necessitam ser recompilados. Caso seja o utilizador a manter mentalmente estas dependências, é certo que mais cedo ou mais tarde vai-se esquecer de uma e começar a gerar aplicações incoerentes, que começam a apresentar erros e falhas de funcionamento aparentemente inexplicáveis.

Se o **make** não abre os ficheiros, para verificar se existe um `#include`, terão de ser os programadores a especificar essas dependências. Aliás o **make** não sabe **C** nem nenhuma outra linguagem, de programação ou não. Manter uma listagem actualizada de todos os ficheiros, e das inclusões que cada um tem, é uma tarefa que dá mais trabalho que esperar mais uns segundos para que todos os ficheiros sejam compilados. Desta forma, não se corre o perigo de gerar uma aplicação incoerente. Por esta razão muitos programadores optam por continuar a não utilizar o **make**. De facto, se tal fosse necessário, e atendendo à velocidade de compilação das máquinas de hoje, poderia ser considerada uma solução aceitável mesmo para projectos com dezenas de ficheiros fonte.

Nesta perspectiva, uma **Makefile**, usando exclusivamente as regras enunciadas acima, seria:

```
programa: a.o b.o c.o
        cc -o programa a.o b.o c.o
a.o: a.c ab.h ac.h
        cc -c a.c
b.o: b.c ab.h
        cc -c b.c
c.o: c.c ac.h
        cc -c c.c
```

Embora não seja um esforço inicial exagerado, é mais um ficheiro que necessita ser actualizado sempre que se altera um `#include`. Notar que se omitiu as inclusões dos

ficheiros de sistema como o **stdio.h** pois assume-se que estes ficheiros nunca são alterados. No entanto, caso se pretenda passar a usar o **gcc** torna-se necessário alterar as 4 referências ao **cc**. Da mesma forma, se mais tarde se necessitar de informação de *debug*, alterações semelhantes necessitam ser feitas. Além disso, sempre que se adiciona um novo ficheiro fonte torna-se necessário acrescentar uma nova regra, onde o ficheiro objecto (**.o**) depende do ficheiro fonte e dos ficheiros que este inclui, devendo executar a compilação desse ficheiro fonte em caso de inconsistência.

Na realidade, o **make** é uma ferramenta complexa e que tem conhecimento prévio das operações mais comuns, podendo o utilizador extender esse conhecimento através de uma linguagem própria. Por exemplo, num caso simples de um programa em **C** constituído por um só ficheiro, por exemplo `main.c`, sem a inclusão de outros ficheiros além dos de sistema, basta fazer `make main` para se obter o executável. Notar que nem é necessário escrever uma **Makefile**, o directório pode conter apenas o ficheiro `main.c`. Aliás, o argumento do **make** designa aquilo que se pretende obter, logo `make main.s` ou `make main.o` permitem obter o ficheiro *assembly* e o ficheiro objecto, respectivamente. Para mais, além de saber os comandos necessários para compilar **C**, o **make** conhece os comandos de compilação da maioria das linguagens de programação mais comuns. De facto se o ficheiro tiver a extensão **.p** ou **.f77**, os exemplos acima produzem igualmente o executável, o *assembly* ou o objecto usando os compiladores de **Pascal** e **Fortran**, respectivamente.

Para o exemplo de compilação separada apresentado na **Makefile** acima, é necessário escrever uma **Makefile** pois não existe forma de saber quais são os ficheiros, desta directoria ou de outras, necessários para construir o executável. No entanto, a **Makefile** inicial pode resumir-se a

```
O = a.o b.o c.o
programa: $(O)
        $(LINK.c) $(O)
```

ou, usando o **gmake**, caso o executável tenha o mesmo nome do primeiro dependente, pode atingir a sua forma mais simples:

```
a: a.o b.o c.o
```

A definição `O` contém a lista dos módulos que constituem o objectivo (`programa`), e a definição `LINK.c` indica que a geração do objectivo deve ser feita com as ferramentas do **C**. Notar que a **Makefile** não inclui dependências dos ficheiros de inclusão pelo que

se torna necessário incluí-las, uma vez que o **make** não analisa o conteúdo dos ficheiros. Esta operação pode ser executada pela aplicação **makedepend**, indicando todos os ficheiros usados que utilizem o pré-processador de **C**, ou seja o **cpp**. Notar que pode-se indicar todos os ficheiros da directoria, mesmo aqueles que não fazem parte do executável, pois apenas os ficheiros que dependam, directa ou indirectamente, do `programa` serão considerados. Para mais, esta operação pode ser incluída como uma regra especial da **Makefile**, ficando

```
O = a.o b.o c.o
programa: $(O)
        $(LINK.c) $(O)
depend:
        makedepend *.c
.PHONY depend
```

Notar que a regra `depend` não tem dependentes pelo que é sempre executada, quando invocada. A última linha não é em geral necessária, mas informa que o **make** não deve tentar procurar ficheiros chamados `depend`, pois este é apenas um nome simbólico. A partir deste ponto é necessário criar as dependências e gerar o executável e utilizá-lo:

```
$ make depend
$ make
$ programa
```

Quando o **make** é invocado sem argumentos, este tenta gerar o objectivo da primeira regra da **Makefile**. Neste ponto, acescentar ou retirar um ficheiro fonte resume-se a actualizar a variável `O`, enquanto acescentar ou retirar a inclusão de um ficheiro se resume a fazer `make depend`.

6.1 Opções da linha de comando

A sintaxe do comando **make**:

```
make [opções] [objectivo ...]  [ definição=valor ...]
```

por exemplo,

```
make -k programa CC=gcc CFLAGS=-g
```

As opções disponíveis são:

-i ignorar os códigos de erro devolvidos pelas aplicações. Desta forma, mesmo que um dos passos intermédios falhe o **make** continua a executar os comandos necessários para gerar o objectivo final, embora o resultado possa ficar incoerente.
Esta opção pode ser activada na **Makefile** com a declaração `.IGNORE`

-k procurar executar todos os passos possíveis. Mesmo que alguns dos passos necessários para gerar o objectivo final falhem, todos os que for possível executar de uma forma coerente são executados. Notar que o **make** termina no primeiro erro que encontra, logo pode ter interesse verificar se outros passos também estão errados. Por exemplo, se no exemplo acima a primeira compilação falhar, já não é possível efectuar a edição de ligações (linkage) final, mas pode-se compilar os restantes ficheiros fonte.

-n indicar as operações que seriam executadas, mas não as executar. Até as linhas iniciadas com '@' são impressas.

-s modo silencioso, ou seja, não imprimir os comandos à medida que estes vão sendo executados. Esta opção pode ser activada na **Makefile** com a declaração `.SILENT`

-f ficheiro usar o ficheiro indicado em vez do ficheiro chamado **Makefile**.

-r ignorar as regras pré-definidas.

-t alterar a data (touch) dos objectivos em vez de executar os comandos que os actualizam. Este comando desvirtua a funcionalidade do **make**, podendo conduzir a resultados incoerentes.

-q verificar se o objecto está actualizado ou não, devolvendo um código de retorno zero ou diferente de zero respectivamente.

-p imprimir as definições e regras existentes e os respectivos valores.

-d imprimir os ficheiros analisados e as regras tentadas (debug).

Todas as opções passadas na linha de comando são guardadas na definição `MFLAGS`, podendo ser passada para outros comandos **make**, por exemplo em subdirectorias. Além disso, existem as definições `MAKE` que designa o comando executado, por exemplo, **make** ou **gmake**, e `MAKELEVEL` que contém o número de sub-makes em execução. Por exemplo, se o **make** foi chamado por outro **make**, `MAKELEVEL` tem o valor 1 e não 0 (zero), como acontece no **make** invocado pelo utilizador.

6.2 A Makefile

Uma **Makefile** é um ficheiro que contém regras e definições, servindo de receita ao programa **make** para manter actualizados todos os ficheiros de geração automática, isto é através da invocação de comandos, de um projecto.

O comando **make** procura um ficheiro designado por **Makefile** ou **makefile**, não devendo existir os dois simultaneamente. Caso não seja indicado um nome alternativo, através da opção '-f', e caso não exista uma **Makefile**, o **make** recorre às definições e regras implícitas pré-defindas (ver opção '-p').

Se o **make** for chamado sem um objectivo como argumento, o **make** procura uma regra chamada '**all:**' no ficheiro **Makefile** e, caso não exista, executa a primeira regra não implícita desse ficheiro.

Tal como já foi dito anteriormente, o **make** distingue espaços de tabulações horizontais (TABS). Assim, as definições e regras devem começar na primeira coluna do ficheiro, isto é sem serem precedidos de espaços ou TABS. Os comandos de uma regra, e as continuações de linha (que usam o carácter \ como em **C**), devem ter um ou mais TABS, mas não podem conter espaços até ao primeiro carácter não branco. No interior das regras, definições ou comandos as várias palavras são separadas por espaços ou TABS, indiferentemente. Os comentários são iniciados com o carácter # e prolongam-se até ao fim da linha, podendo ser continuados se a linha terminar no carácter de continuação.

Os comandos são executados num sub-*shell*, sendo cada comando executado num sub-*shell* distinto e o seu valor de retorno verificado. Caso se pretenda executar diversos comandos dentro do mesmo sub-*shell* deverá usar-se, igualmente, o carácter de continuação. No entanto, deve ter-se presente que apenas o valor de retorno do sub-*shell* é testado, em geral, o último programa a ser executado pelo sub-*shell*. Caso se pretenda ignorar o valor de retorno de um sub-*shell*, continuando a construção do objectivo independentemente desse valor, deve-se preceder a operação pelo sinal de '-'. Finalmente, para impedir que o comando a ser executado seja impresso no terminal deve-se preceder a operação pelo sinal '@', podendo usar-se a sequência '-@' para não imprimir e ignorar o valor de retorno de um comando.

```
CFLAGS=-g
CC=gcc -Wall -ansi \
      -pedantic
todos.o: todos.c a.h b.h c.h d.h e.h f.h \
      g.h h.h i.h j.h muitos.h outros.h \
                ficheiros.h podem.h seguir.h
      -/bin/false              # ignorar o codigo de erro
```

```
@echo "nao imprimir esta mensagem"
-@comando que nao imprime e \
                  pode dar erro
        $(COMPILE.c) todos.c # usa CC e CFLAGS, ver 'make -p'
```

6.3 Dependências

As dependências, ou seja, o objectivo e os seus dependentes podem surgir em regras, isto é, seguidos de comandos, ou isoladas. Caso surjam isoladas, as dependências são adicionadas umas às outras. Contudo, só uma das dependências pode ter a sequência de comandos a executar.

Por outro lado, as dependências isoladas podem referir-se a mais de um objectivo. Por exemplo, no caso dos ficheiros de inclusão pode-se indicar todos os ficheiros que dependem de um determinado ficheiro.

```
a.o b.o: ab.h
a.o: ac.h
a.o: a.c
        cc -c a.c
```

Notar que no caso dos ficheiros de inclusão (.h), quem depende do ficheiro não é o programa fonte (.c) que contém a directiva `#include`, mas o ficheiro objecto (.o) que deve ser gerado novamente quando o ficheiro de inclusão é modificado. Saliente-se que um ficheiro de inclusão alterado não gera um novo ficheiro fonte. Contudo, os ficheiros fonte ou ficheiros de inclusão podem ser gerados. Tal é o caso do gerador de analisadores sintácticos **yacc**, onde um ficheiro `sintaxe.y` gera os ficheiros `y.tab.h` e `y.tab.c` através da execução do comando `yacc -d sintaxe.y`.

6.4 Pseudo-objectivos

Os pseudo-objectivos definem comandos, em geral sob a forma de regras, que não correspondem a ficheiros existentes ou a serem gerados. Uma vez que os comandos das regras sem dependentes são sempre executados, quando a regra é invocada, é possível utilizar o comando **make** para realizar acções predefinidas.

O exemplo do 'depend:' acima é um caso comum, como por exemplo,

```
clean:
      rm -f a.out core *.o $(OBJ)
```

A regra acima pode ser incluída como dependente de outra regra, forçando os ficheiros indicados a serem apagados antes dos comandos dessa regra serem executados.

No entanto, caso exista um ficheiro chamado 'clean' o **make** irá tentar verificar, através da sua data de modificação, se a regra 'clean:' necessita ser executada. Para que o ficheiro 'clean' não seja confundido com o pseudo-objectivo 'clean:' deve-se incluir o pseudo-objectivo predefinido '.PHONY', tal como exemplificado no caso do 'depend:' acima.

Além do '.PHONY:' existem outros pseudo-objectivos predefinidos, de onde se salienta:

.SUFFIXES: permite alterar a ordem de selecção dos sufixos dos ficheiros em caso de ambiguidade. Ver regras implícitas.

.PRECIOUS: impede que os ficheiros intermédios, indicados como dependentes, sejam apagados quando o comando **make** é interrompido. Deve ser utilizado para ficheiros cuja geração é muito demorada, evitando que sejam apagados devido à geração do objectivo final ter falhado por qualquer outra razão que não involva directamente os referidos ficheiros.

.SILENT equivale à opção **-s** da linha de comando.

.IGNORE equivale à opção **-i** da linha de comando.

6.5 Definições

As definições são macros, ou variáveis, do **make**. As definições podem ser atribuídas entre si ou concatenadas, por exemplo 'X = $(LINK,c) $O'. Se o nome da definição for constituida por uma só letra, por exemplo a lista de objectos do exemplo, o seu valor pode ser obtido por '$O'. Caso seja constituída por mais de uma letra deve-se usar parentesis '$(LINK.c)' ou chavetas '${LINK.c}'. O mecanismo de substituição de sufixos '$(var:str1=str2)' consiste em substituir todas as ocorrências de 'str1' por 'str2' em 'var'. Por exemplo, '$(O:.o=.c)' terá o valor 'a.c b.c c.c', podendo ser usado como argumento do 'makedepend'.

Além das variáveis definidas dentro da `Makefile` ou predefinidas pelo **make**, podem ser acedidas todas as variáveis exportadas pelo editor de comandos (**shell**), como por exemplo **$HOME**, **$USER**, **$EDITOR**, **$MAIL**, **$TERM**, **$LD_LIBRARY_PATH** ou **$CVSROOT**.

Definições dinâmicas As definições dinâmicas têm um valor que depende da regra onde se inserem, só podendo ser usadas dentro de uma regra:

$< designa o primeiro dos dependentes

$? designa todos os dependentes mais recentes que o objectivo, separados por um espaço branco.

$@ designa o objectivo

$* usado apenas em regras implícitas designa o nome do ficheiro sem extensão

acrescenta-se um conjunto de outras definições dinâmicas exclusivas do gmake:

$^ designa todos os dependentes, separados por um espaço branco.

$+ designa todos os dependentes mantendo os nomes repetidos, pela ordem que surjem, separados por um espaço branco.

$% designa o nome do membro do arquivo, se o objectivo for um membro do arquivo. Por exemplo, se o objectivo for 'foo.a(bar.o)' então $% é 'bar.o' e $@ é 'foo.a'. $% é vazio se o objectivo não for membro do arquivo.

$(@D) e $(@F) se o objectivo for um nome composto 'dir/foo.o' então '$(@D)' vale o caminho para o ficheiro 'dir' e '$(@F)' o nome do ficheiro sem o caminho 'foo.o'. Definições idênticas existem substituindo '@' por '*', '%', '<', '?' e '^'.

Por exemplo,

```
prog: a.o libx.a
	cc $< -o $@ -lx

# apenas os x?.o modificados sao adicionados 'a biblioteca
libx.a: x1.o x2.o x3.o x4.o x5.o
	ar crl $@ $?
	ranlib $@

# todos os dependentes sao usados (gmake)
prog2: a.o x1.o x2.o x3.o x4.o x5.o
	 cc $^ -o $@
```

Definições com padrões As definições com padrões existem exclusivamente para a ferramenta **gmake** e permitem generalizar os mecanismos de substituição de sufixos. A substituição de padrões permite que em '$(var:base1%suf1=base2%suf2)' todas as ocorrências do padrão 'base1%suf1' sejam substituídas por 'base2%suf2' em 'var', considerando que '%' representa zero ou mais caracteres (o equivalente ao '*' no *shell*).

Por exemplo, se uma variável 'XX' tiver o valor 'xx_a.c xx_b.c yy_c.c', então a expressão '$(XX:xx_%.c=%.o)' terá o valor 'a.o b.o yy_c.c'.

6.6 Regras implícitas

Uma vez que os processos de compilação são idênticos, só mudando o nome dos ficheiros alvo, o **make** permite construir regras que não dependem do nome do ficheiro mas apenas do seu sufixo. Assim, podemos dizer que um ficheiro fonte '.c' gera um ficheiro *assembly* '.s' com o comando 'gcc -S':

```
.c.s:
 gcc -S $<
```

Notar que as referências aos nomes do objectivo e seus dependentes tem de ser feita por definições dinâmicas, pois não existem nomes de ficheiros.

Para que as regras implícitas sejam correctamente executadas, o **make** necessita de conhecer os prefixos em uso. A definição 'SUFFIXES' contém os sufixos conhecidos (ver make -p), podendo ser alterada através da directiva '.SUFFIXES:'. Esta directiva permite, se utilizada sem argumentos, eliminar o conhecimento prévio que o **make** tem dos sufixos mais comuns. Se for incluída uma lista de sufixos, como argumento da directiva '.SUFFIXES:', estes serão procurados pela ordem indicada. Desta forma, a directiva '.SUFFIXES:' pode ser utilizada para acrescentar novos sufixos ou, apenas, para alterar a ordem de procura. Por exemplo, se existirem na directoria dois ficheiros com o mesmo nome `main`, sendo um escrito na linguagem **C** '.c' e outro na linguagem **Pascal** '.p', e se pretender obter o ficheiro objecto '.o' através de `make main.o` qualquer dos dois ficheiros pode ser utilizado, sendo a escolha feita através da ordem dos respectivos prefixos na definição 'SUFFIXES'.

Por exemplo, em

```
.SUFFIXES: .cpt .asm .c .p
.cpt.asm:
        compact $< > $@
```

```
.asm.o:
        nasm -felf $<
```

o **make** fica habilitado a gerar ficheiros objecto '.o' a partir de ficheiros '.asm' directamente, e a partir de '.cpt' indirectamente. Além disso, garante-se que em caso de conflito, por exemplo para gerar um '.o' a escolha recai primeiro no '.cpt' (mais prioritário que o '.asm', '.c' ou '.p'), só sendo usado um '.p' se nenhum dos outros três existir. Notar que as regras implícitas '.c.o:' e '.p.o:' continuam a existir, apenas se acrescentou mais duas regras.

Regras implícitas usando padrões O **gmake** também permite utilizar padrões em regras implícitas que ficam com a forma

```
op%os: [dependentes ...] dp%ds [dependentes]
        [comandos]
```

onde 'op' e 'os' são o prefixo e o sufixo do objectivo, respectivamente, e 'dp' e 'ds' são o prefixo e o sufixo do dependente, respectivamente. O carácter '%' representa o nome base constituído por zero ou mais caracteres que é encontrado num objectivo e usado para construir o nome das dependências.

Por exemplo, para gerar um executável, sem extensão, a partir de um '.cpt' com o mesmo nome, pode-se fazer

```
%: %.cpt
compact $< > $*.asm
        nasm -felf $*.asm
ld $*.o lib.o -o $@
```

7 gcc: compilador da linguagem C

Esta secção debruça-se sobre o compilador da linguagem **C gcc** e o processo de compilação. Comecemos por referir que o comando **gcc**, tal como o **cc** e muitos compiladores de outros compiladores, são de facto programas de interface que invocam diversos programas. Na realidade, apenas um desses programas é de facto o compilador propriamente dito, em geral designado por **cc1** ou **ccom**.

O **gcc** permite produzir um executável, no ambiente definido pela linguagem **C**, a partir de um ficheiro fonte (.c) na linguagem **C** e respectivas declarações (.h) ou de qualquer dos formatos intermédios gerados durante o processo de compilação:

pré-processado: resulta da inclusão dos ficheiros de declaração, usados num ficheiro fonte, através do processamento das directivas `#include`, além das restantes directivas do pré-processador que são iniciadas por # no início de uma linha. Por exemplo, definição de constantes com `#define` ou compilação condicional com `#ifdef`. Notar que é este ficheiro que é lido pelo compilador e que, nomeadamente, os números de linha em que os erros são reportados referem-se a este ficheiro, diferindo por vezes do ficheiro fonte original. Quando um erro parecer não ter relação com o código presente na linha indicada, pode ser útil obter este ficheiro intermédio com a opção **-E**, para verificar se alguma macro (em geral desconhecida do programador) não efectuou subtituições imprevistas.

assembly: resulta da invocação do compilador, propriamente dito, sobre o ficheiro pré-processado. O resultado é ainda um ficheiro de texto, que pode ser lido e modificado com editor de texto normal, mas cujas instruções são específicas do processador. Estas instruções são também, em geral, dependentes do sistema operativo, caso as rotinas compiladas realizem chamadas ao sistema (*system calls*). Algumas directivas são dependentes do tipo de ficheiro objecto utilizado e até do *assembler* a utilizar. Em UNIX existem diversos formatos de ficheiro objecto, `a.out`, `coff` e `elf` são os mais comuns. Nesta cadeira também iremos utilizar dois *assembler*s muito diferentes: o **gas** para ficheiros gerados com o **gcc** e o **nasm** para ficheiros gerados pelo compilador desenvolvido na cadeira.

objecto: mais rigorosamente **ficheiro objecto relocatável**, resulta da invocação do *assembler* (em português, programa `montador`), que lê instruções em *assembly* e gera um ficheiro em formato binário, designado por ficheiro objecto. Em primeiro lugar, é de referir que o nome não tem nada a ver com linguagens de programação orientadas por objectos, já existindo estes tipos de ficheiros muitos anos antes de tais linguagens terem surgido. Por formato binário entende-se um formato que já

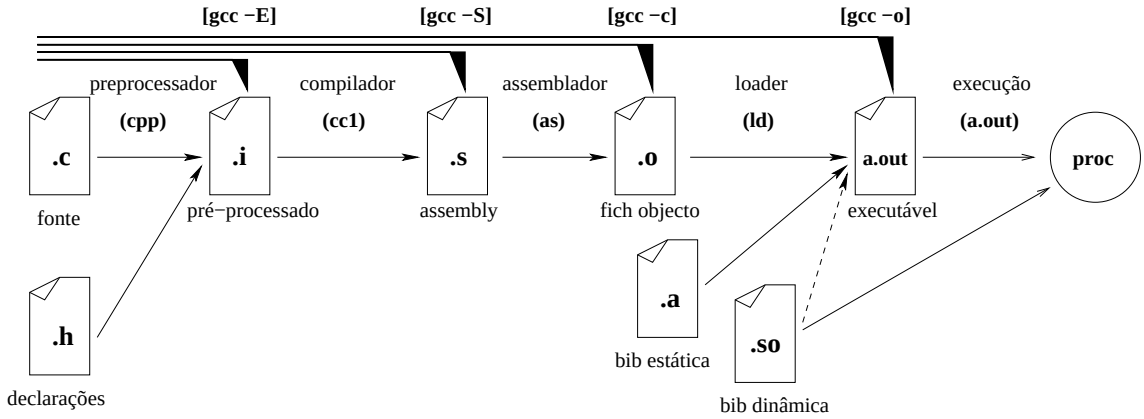
não pode ser lido com um editor de texto. De facto, as menemónicas (nomes simbólicos das instruções do processador, por exemplo `mov`, `call` ou `push`) foram substituídas pelos respectivos valores numéricos utilizados pelo processador (por exemplo um `call` tem o código `E9` em **i386**, seguido de um endereço da função, por exemplo `080489e0`).

biblioteca estática: uma biblioteca estática não é gerada pelo comando **gcc**, mas pelo comando **ar**. Contudo, pode ser utilizada pelo **gcc** para gerar um executável final ou outro ficheiro relocatável. Notar que, quer um ficheiro relocatável, quer um executável final são ficheiros objecto, apenas o executável final contém endereços fixos, logo não reposicionáveis ou relocatáveis. Estas bibliotecas serão tratadas mais adiante, mas são colecções de ficheiros objecto relocatáveis, podendo estes ser incluídos ou retirados da biblioteca interactivamente.

biblioteca dinâmica: também uma biblioteca dinâmica não pode ser gerada pelo **gcc**, mas pelo *loader* (`carregador` em português) também designado por *linker*. Tal como as bibliotecas estáticas, as bibliotecas dinâmicas podem ser utilizadas pelo **gcc** para gerar o executável final. Ao contrário das anteriores, as rotinas das bibliotecas dinâmicas não são incluídas no ficheiro executável final, ficando este apenas com uma referência para a biblioteca e com os endereços das suas rotinas que utiliza. Desta forma, embora poupando espaço no ficheiro executável, a biblioteca dinâmica necessita estar presente no momento da execução do processo. Além disso, necessita estar posicionada no directório onde estava no momento da compilação, ou a sua localização ser referida pela variável de ambiente `LD_LIBRARY_PATH`.

7.1 O compilador gcc

O processo de compilação controlado pelo **gcc** pode ser resumido pela figura seguinte:



7.2 A linguagem C em ambiente UNIX

Embora a linguagem **C** exista desde o início da década de 1970, tem sofrido diversas actualizações e normalizações. Na realidade, o único *software* que não necessita ser actualizado é aquele que já não se utiliza. Neste sentido, e como existem diferenças que, embora pouco significativas, podem compromoter o teste do projecto da cadeira em **linux-elf-i386** quando o desenvolvimento for efectuado noutro ambiente, torna-se necessário especificar as normas que devem ser cumpridas. Estas incluem, a linguagem e respectivas regras de codificação, bem como as funções predefinidas que podem ser utilizadas, sejam estas funções de biblioteca ou de interface com o sistema operativo.

As funções predefinidas da linguagem **C** a utilizar devem ser normalizadas, ou seja, na respectiva página de manual deve constar "CONFORMING TO":

linguagem C: ANSI C (ANSI X3.159-1989).

funções de biblioteca: ISO C99 (ISO/IEC 9899:1999).

interface com o sistema operativo: POSIX (POSIX 1003.1).

Para além disso, é necessário respeitar os limites impostos pela linguagem. Embora alguns dos compiladores superem largamente muitos destes limites, muito existe ainda que mantêm muitos desses limites. Notar que alguns limites estão indirectamente limitados pelo sistema operativo ou o formato do ficheiro objecto utilizado. Desta forma, convém ter presente que:

- as directivas `#include` estão limitadas a 8 níveis de profundidade, ou seja, no processamento do ficheiro fonte apenas podem estar simultaneamente abertos 8 ficheiros de declaração. É pois muito importante evitar inclusões mútuas e desnecessárias.
- as directivas `#if` e suas variantes, como `#ifdef`, também estão limitadas a 8 níveis de profundidade.
- o nível de profundidade dos parênteses curvos '(' e ')' está limitado a 32 para expressões completas. Especial cuidado deve ser tido com as macros que, em geral, utilizam muito parênteses para evitar ambiguidades na sua substituição.
- o número máximo de caracteres por linha lógica está limitado a 509. Notar que, em **C**, é possível continuar uma instrução ou cadeia de caracteres na linha seguinte, terminando a linha anterior com um barra para trás '\', o que constitui uma única linha lógica. Mais uma vez as macros recorrem frequentemente a tais caracteres de continuação, produzindo na sua substituição linhas muito longas.

- apenas os primeiros 6 caracteres de um identificador exterior são significativos, variável global ou função global. Notar que a maioria dos formatos de ficheiro objecto actuais já permite pelo menos 32 caracteres significativos, nomeadamento o formato **elf** que utilizaremos na cadeira, mas é sempre bom não assumir tais extensões.

- um máximo de 127 membros por estrutura '`struct`' ou '`union`'.

- um máximo de 31 parâmetros por chamada a função. Neste caso, o cuidado reside nas funções de argumentos variáveis (`varargs`) como o `printf` ou o `scanf`.

7.3 Regras de codificação

A linguagem **C**, por ser muito permissiva, conduz frequentemente a que o programador escreva instruções, que sendo correctas em **C** e podendo ser gerado código para elas, não reflectem o pensamento do programador. Por forma a obviar muitos destes problemas, coleccionados por programadores zelosos, dedica-se esta secção a algumas regras de codificação que impediram muitos dos erros mais vulgares.

Note que estas regras podem salvar-lhe a ...*disciplina*.

- As regras de indentação, se seguidas rigorosamente, podem impedir muitos erros desagradáveis.

- Não poupe nos caracteres do ficheiro pois vai pagar esse espaço em tempo gasto a encontrar erros.

- Não escreva código tão compacto que nem você consegue explicar. (Existem concursos de *obfuscated C* para esses prodígios)

7.3.1 Regras base

- As directivas do pré-processador devem ter o '#' na primeira coluna. Embora apenas alguns pré-processadores exijam esta regra, a sua utilização facilita a leitura.

- Utilize sempre nomes com todas as letras maiúsculas para constantes e macros definidas pelo pré-processador. Assim, está sempre alertado para possíveis substituições de código. Infelizmente, muitas das macros dos ficheiros de declaração não seguem esta regra.

- Todos os ficheiros de declaração (.h) devem ser protegidos contra dupla inclusão. A certa altura os ficheiros já se incluem todos uns aos outros sem que tenha controlo sobre isso. Para tal basta delimitar esses ficheiros com:

```
#ifdef _FILE_H_
#define _FILE_H_
...
#endif /* _FILE_H_ */
```

  onde `FILE` deve ser substituído pelo nome do ficheiro em questão.

- Aconselha-se um indentação de 2 a 4 colunas, por forma a facilitar a leitura sem forçar a mudanças de linha em instruções de dimensão média.

- Evite utilizar nomes ingleses para funções e variáveis, mesmo que locais. Estes nomes podem colidir com nomes de funções de biblioteca ou variáveis globais. Notar que se definir uma função chamada **open** deixa de poder abrir ficheiros, pois o próprio **fopen** utilizada o **open** para efecutar a abertura.

- Utilize sempre nomes com apenas a primeira letra maiúscula para definir novos tipos de dados **typedef**.

- Nunca junte o ponteiro com tipo '**char* p;**', mas sempre com a variável '**char *p;**', pois se juntar mais variáveis apenas o tipo é distribuitivo '**char *p, *s, t;**' (**t** é um único carácter e não um ponteiro).

- Utilize um espaço branco de cada lado dos operadores, ou seja, **if (a + b < c) {** em vez de **if(a+b<=c){**.

- Declare todas as funções locais a um ficheiro como **static**, excepto se necessitar de as invocar de outros módulos. Além de poder colidir com outras funções com o mesmo nomes, atrasa o processo de edição de ligações. Assim, apenas as funções e variáveis acessíveis de outros módulos não devem ter o prefixo **static**. Por outro lado, estas funções e variáveis devem estar todas declaradas no ficheiro de declarações respectivo, para poderem ser correctamente utilizadas pelos outros módulos.

7.3.2 Condições e ciclos

- Não utilize expressões como condições, por exemplo **if (a + b)**, mas inclua sempre uma condição **if (a + b != 0)**.

- Utilize, sempre que possível a constante no início da condição, ou seja, **if (0 == a)** é preferível a **if (a == 0)**. Caso digite mal (na melhor das hipóteses), e fique com **if (a = 0)** tem uma condição correcta mas sempre falsa e que lhe coloca a variável a 0 (zero). Acontece com maior frequência que seria desejável, acredite.

- Utilize sempre um bloco, mesmo que pretenda incluir apenas uma instrução. Se mais tarde acrescentar outras instruções ou esta for substituída por uma macro que gera mais de uma instrução, apenas a primeira pertence à condição ou ao ciclo. Ver exemplo de blocos na indentação.

- Os blocos dos ciclos e dos **if**s começam no fim da linha que inicia a instrução e terminam na mesma coluna, por exemplo,

```
if (a > b) {
  c = a - b;
} else {
  c = b -a;
}
```

- Se uma condição ocupar mais de uma linha, coloque as condições lógicas alinhadas com início da condição,

```
if (   a >= b + c
    || b >= a + c
    || c >= a + b) {
```

- Caso o bloco do ciclo seja vazio, coloque o ';' numa linha vazia. Até ajuda incluir um comentário do tipo **/* empty */** .

7.4 Principais ficheiros de declarações

As páginas de manual de cada função utilizada, acessíveis através de `man função`, indicam no início quais os ficheiros de declaração que necessitam ser incluídos. Tais ficheiros de declaração permitem definir estruturas de dados utilizadas pelas rotinas e incluem a declaração do tipo dos seus parâmetros e valor de retorno, por forma a que seja possível ao compilador fazer as verificações exigidas pela linguagem.

Notar que, tais declarações influenciam directamente a geração de código. Por exemplo,

```
int main()
{
  printf("%g\n", pow(3,4));
}
```

ao ser compilado

prompt$ gcc main.c -lm

produz o resultado **-1.99778**, que é claramente diferente de $3^4 = 81$. Notar que a biblioteca das funções matemáticas **-lm**, necessita ser incluída na geração do executável final. No entanto, se se incluir, antes da primeira invocação da função `pow` a sua declaração:

```
double pow(double,double);
int main()
{
  printf("%g\n", pow(3,4));
}
```

o resultado da execução já é o esperado. Tal facto deve-se, neste caso, à dimensão do tipo `double` ser diferente do tipo dos literais inteiros 3 e 4. Assim, os valores colocados na pilha de dados são inteiros em complemento para dois com 32 bits e não números em vírgula flutuante de precisão dupla no formato **IEEE** com 64 bits.

Os ficheiros de declaração agrupam estas declarações por temas, permitindo ao incluir um único ficheiro declarar dezenas de funções relacionadas, e que provavelmente várias delas serão utilizadas no mesmo programa. No exemplo anterior poderia substituir-se a declaração da função `pow` pela inclusão do ficheiro `math.h` que declara, além da função `pow`, as funções trigonométricas, raiz quadrada e muitas outras funções predefindas que recorrem a números em vírgula flutuante. Notar que a inclusão do ficheiro `math.h` não permite eliminar a indicação da biblioteca **-lm**, na realidade o ficheiro `/usr/lib/lib`**`m`**`.a` ou `/lib/lib`**`m`**`.so.6` (onde 6 é o número da versão) consoante se use a biblioteca estática ou dinâmica. Os ficheiros de declaração apenas informam o compilador da forma como o código deve ser gerado e auxiliam-no nas verificações sintácticas e semânticas.

Os ficheiros de declarações mais usados são:

stdio.h inclui as funções de uso mais comum, sendo regra geral incluir sempre este ficheiro. Tais funções incluem os habituais `printf` e `scanf` e outras rotinas de interacção com ficheiros através da estrutura `FILE`.

math.h conforme referido atrás, inclui as funções matemáticas predefinidas em **C**. Estas funções não se encontram na biblioteca geral da linguagem **C**, ou seja `libc.a` ou `libc.so.6`, sendo necessário incluir a biblioteca de matemática `libm.a` ou `libm.so.6`, através da opção **-lm** após a indicação dos ficheiros que dela necessitam.

string.h inclui as funções de manipulação de cadeias de caracteres (*NULL terminated*), como `strcpy` ou `strcmp`.

stdlib.h inclui funções predefindas de uso geral `atoi`, `malloc`, `alloca`, `strtol` ou `getenv`. Este ficheiro, tal como o `stdio`, é quase sempre incluído.

ctype.h inclui funções e macros para a manipulação de caracteres, tais como `isupper`, `isdigit` ou `toupper`.

unistd.h inclui as declarações das funções e estruturas de dados necessárias para efectuar as chamadas ao sistema. Algumas chamadas ao sistema, como o acesso ao sistema de ficheiros, exigem a inclusão de outros ficheiros além deste.

fcntl.h inclui as declarações de constantes e funções que são necessárias para aceder ao sistema de ficheiros sem o uso de tampões (a estrutura `FILE`). Por exemplo, as funções `open` ou `fcntl`.

signal.h inclui as declarações das funções e códigos das interrupções de sistema. Por exemplo, `signal(SIGINT, control_C);` determina que caso seja gerada uma interrupção `SIGINT`, em geral através de um *CTRL-C* do terminal, seja chamada a função definida pelo utilizador `control_C`.

errno.h inclui os códigos de erro das chamada ao sistema operativo. Desta forma é possível saber porque uma chamada ao sistema falhou. Por exemplo, quando uma tentativa de abertura de um ficheiro falha, devolvendo a função `open` o código **-1** ou a função `fopen` o ponteiro nulo, a variável global **errno** tem o valor **EPERM** no caso de erro de permissão ou o valor **ENOENT** no caso do ficheiro pedido não existir, entre mais duas dezenas de erros possíveis. Notar que, para imprimir no terminal o erro basta utilizar a função **perror** definida no ficheiro `stdio.h`, por exemplo,

```
...
if ((fp = fopen(filename, "r")) == NULL) {
```

```
        perror(filename);
        exit(2);
    }
    ...
```

dirent.h inclui as declarações das funções utilizadas pela ler o conteúdo dos directórios.

setjmp.h inclui as declarações das funções utilizadas para efectuar **goto**s entre funções, especialmente úteis no tratamento de interrupções. Notar que o **C** não tem um mecanismo de tratamento de interrupções tão evoluído como o **Java**, mas oferece as primitivas necessárias à sua realização, através das funções `setjmp` e `longjmp`.

varags.h inclui as declarações utilizadas para definir funções com número variável de argumentos, como o `printf`. Notar que este ficheiro não é necessário para invocar tais funções, como deve ser óbvio da utilização do `printf`, mas apenas para definir novas funções com a mesma capacidade.

7.5 O compilador gcc

O compilador de **C**, ou como vimos a interface com este através do comando **gcc**, permite controlar todo o processo de geração de uma forma simples (embora não tão simples como o `CTRL-F5` das ferramentas de WINDOWS).

Para controlar o ponto de paragem do processo de geração do executável final poderá existir uma das seguintes opções:

-E gera apenas o ficheiro pré-processado, executando o pré-processador **cpp** para incluir os ficheiros de declaração, seleccionar as instruções de compilação condicional e substituir as macros.

-S gera o ficheiro *assembly*, com a extensão **.s**, contendo o resultado da invocação do compilador propriamente dito.

-c gera o ficheiro objecto relocatável, com a extensão **.o** que resulta da invocação do *assembler*. Pode receber como argumentos ficheiros fonte **.c** (pré-processados ou não) e ficheiros *assembly* **.s**.

Caso não seja indicada nenhuma opção de paragem, o compilador tentará gerar um executável final efectuando a ligação de todos os ficheiros compilados e das bibliotecas indicadas. Por exemplo, o comando **gcc main.c intr.s aux.o libx.a -lm** tentará compilar o ficheiro **main.c**, depois invocará o *assembler* para os ficheiros **main.s** e **intr.s**, finalmente

efectuará a edição de ligações por forma a produzir o executável final **a.out** dos três ficheiros objecto **main.o**, **intr.o** e **aux.o**, indo buscar às bibliotecas **libx** e **libm** os símbolos (variáveis e funções) utilizados mas não definidos pelos três ficheiros objecto.

Outras opções úteis incluem:

-o `resultado` permite dar um nome específico ao ficheiro resultante. Notar que se o resultado for um executável, deixará de ser **a.out** para ter o nome indicado, mas se for um ficheiro objecto relocatável terá igualmente o nome indicado, em vez do nome do ficheiro original com extensão **.o**.

-I procura os ficheiros de declaração nos directórios pela ordem que são indicados, por exemplo, **-I.l/include** ou **-I/usr/local/include**. Se indicado sem argumentos impede a procura nos directórios usados por omissão como **/usr/include**.

-L igual ao **-I** mas para bibliotecas.

-l*biblioteca* procura os símbolos indefinidos, até ao momento, na biblioteca **abc**. A biblioteca é um ficheiro `lib`**abc**`.a` ou `lib`**abc**`.so.N` (onde `N` é número da versão da biblioteca dinâmica).

-g gera informação simbólica de análise para ser usado por um *debugger* simbólico como o **gdb**.

-p gera código para contabilização de desempenho (*profiling*) para ser utilizado por ferramentas de análise de desempenho como o **gprof**.

-O, -O2, -O3 liga o optimizador de código. Quanto maior for o número indicado mais serão as optimizações realizadas. Em geral, apenas a opção **-O** pode ser usada em conjunção com **-g**, pois as outras alteram a ordem das instruções impedindo o *debugging* simbólico, mas podendo ainda ser depurado. Cada tipo de optimização pode ser escolhido individualmente, em vez de utilizar as opções **-O**, através das opções **-f**. Por exemplo, **-funroll-loops**, **-fstrength-reduce**, **-fno-peephole**, **-fcaller-saves**, **-fomit-frame-pointer**, **-fno-defer-pop**, *etc.*

-s remove todos os símbolos do executável final.

-Wall liga o verificador de código fonte. Este verificador faz um grande número de verificações adicionais que, sendo permitidas pela linguagem, são, em geral, resultado de erros ou possíveis fontes de erro para a maioria dos programadores. É **altamente** aconselhada a utilização desta opção. As verificações podem ser seleccionadas individualmente com as opções **-W**.

-ansi liga o verificador da nor-ma **ansi**. Notar que ainda são permitidas construções não **ansi**.

-pedantic rejeita tudo o que não pertencer rigorosamente à nor-ma definida. Desta forma, **-ansi -pedantic** garante que os programas obedecem rigorosamente à norma **ansi**, em geral designados por **strict-ansi**. Esta opção é também **altamente** aconselhada, sob risco de o programa não funcionar sob as condições de teste.

-Dmacro=valor define a macro com o valor indicado, ou com o valor unitário caso a atribuição não seja indicada.

-Umacro elimina uma macro predefinda.

-static -shared -symbolic permitem controlar o tipo de ficheiros objecto relocatáveis e executáveis a gerar. Ver **ld**, mais adiante.

-time imprime os tempos de execução de cada uma das fases do processo: pré-pro-ces-sa-men-to, compilação, assemblagem, edição de ligações.

8 gdb: depurador de código (*debugger*)

O objectivo de um *debugger*, ou depurador de código em português, consiste em permitir observar o que acontece nos passos intermédios da execução de um programa. Um *debugger* permite suspender a execução do programa em certos pontos para permitir a análise e modificação dos valores das variáveis nesses pontos, sem ter de alterar e recompilar o programa. Um *debugger* permite, ainda, analisar o resultado de uma execução que terminou de forma abrupta, gerado um *core*.

Um *debugger* substitui, com vantagem, as tradicionais instruções de impressão (*printf*'s) utilizados nas formas mais simples de depuração. De facto, um *debugger* permite analisar variáveis de outras funções activas que não aquela onde se encontra o programa além de estabelecer condições de paragem dinamicamente. Tudo isto pode ser executado sem necessidade de recompilar o programa, o que se pode tornar significativo em programas de grande dimensão.

A análise de situações *post-mortem*, em que o programa gera o ficheiro *core*, permite detectar erros que poderiam não se verificar caso existam *printf*'s ou outras formas de intrusão no programa original, incluindo a utilização do próprio *debugger* durante a execução. A rotina `abort()` permite terminar o programa, tal como a rotina `exit()`, gerando o ficheiro *core* para posterior análise. Desta forma, podem-se detectar certo tipos de erros, os mais difíceis de detectar, pois desaparecem ao interagir com o programa.

8.1 Utilização básica

Embora o *debugger* **gdb**, proporcione uma grande gama de comandos, existe um número reduzido de comandos cuja utilização é mais frequente. Devido às necessidades mais comuns do programador e permitem:

- descobrir o local onde ocorreu um erro,
- ver e alterar os valores das variáveis,
- colocar pontos de paragem (*break-points*),
- executar e controlar o funcionamento do programa.

8.1.1 Informação simbólica

Os programas para serem analisados por *debugger* simbólico como o **gdb** necessitam ser compilados com informação simbólica. A informação simbólica está dividida em informação de ligação e informação de análise. Caso esta informação não exista, o programa pode ainda ser analisado. No entanto existem apenas referências a posições de memória e instruções máquina sem relação com o ficheiro fonte original.

A informação de ligação, que tem de existir em todos os ficheiros objecto para que estes possam ser usados para gerar executáveis, inclui essencialmente a localização das funções e variáveis globais e, eventualmente locais. Esta informação fica presente no executável final, excepto se for explicitamente indicado através da opção **-s** do *loader* (editor de ligações) ou de alguns compiladores, ou retirado do executável já produzido através do comando **strip**.

A informação de análise, por outro lado, necessita ser explicitamente pedida ao compilador através da opção **-g**, na maioria dos compiladores, pelo menos em UNIX. Esta informação inclui as instruções que correspondem a cada linha no código fonte, a constituição das estruturas e tipos de dados das variáveis. Desta forma o utilizador pode seguir a execução do programa em função das instruções que codificou sem se preocupar como são codificadas ou executadas pela máquina que usa.

Notar que toda esta informação, que aumenta significativamente a dimensão do executável, pode ser retirada com o comando **strip**, quando deixar de ser necessária. Contudo, não é possível obter um código optimizado e simultaneamente preparado para *debug*. Embora algumas optimizações sejam possíveis, com alguns compiladores, muitas optimizações modificam a ordem das instruções, impedindo um *debugger* simbólico de coordenar o código do ficheiro fonte com as instruções máquina que o realizam.

Assim, caso pretenda utilizar um *debugger*, deverá instruir o compilador para gerar informação simbólica de análise, em geral utilizando a opção **-g**. Uma vez gerado o executável, o programa pode ser executado sob o controlo de um *debugger*, no caso o **gdb**. O **gdb** admite dois argumentos, o primeiro é o nome do ficheiro que contém o executável a analisar, ou **a.out** por omissão, enquanto o segundo argumento é o nome do *post-mortem* que deve corresponder ao executável que o gerou, em geral designado por **core**. Para o utilizador mais experimentado, o **gdb** também permite analisar um programa já em execução, quer através da indicação do seu **pid** na linha de comando, quer através dos comandos **attach** e **detach**.

8.1.2 Operações básicas

Devido ao elevado número de comandos que dispõe, o **gdb** tem um comando de ajuda interactivo designado por `help`. Este comando admite como argumentos os nomes de outros comandos ou classes de comandos, por exemplo `breakpoints`, `running`, `stack`, `status` ou `files`.

Para abandonar o **gdb** existe o comando **quit** ou pode-se digitar o `fim-de-ficheiro` (^**D**, em UNIX).

Ao iniciar o **gdb** apenas o processo do *debugger* é iniciado, embora seja possível ver o código fonte não é possível inspeccionar variáveis ou qualquer entidade dinâmica. Para iniciar o programa a analisar é necessário executar o comando **run**, com os mesmos argumentos que utilizaria numa execução normal, incluindo redirecções de e para ficheiros.

Assumindo que o programa tem um erro que obriga a terminar a execução abruptamente, por exemplo uma falta de segmento (*Segmentation fault*) para referir um dos mais comuns, voltará a obter a linha de comando do **gdb**. Neste ponto poderá usar um dos comandos:

backtrace: pode ser abreviado para **bt**, e permite determinar não só a função e linha em que o problema foi detectado, como todas as funções invocadas e respectivos argumentos.

list: pode ser abreviado para **l**, e permite mostrar as linhas de código fonte em torno do ponto de paragem. Se for indicado um argumento pode mostrar uma função (`list main`) ou linha de um ficheiro (`list 23` ou `list fich.c:23`).

print: pode ser abreviado para **p**, e permite imprimir o valor de variáveis do programa, visíveis na linha em que a execução do programa se encontra, ou expressões contendo essas variáveis.

Determinar um ponto de paragem Para inspeccionar o programa, é necessário colocar pontos de paragem em locais específicos do código fonte. Os pontos de paragem mantêm-se entre diferentes execuções do programa em análise, desde que não se abandone o *debugger*. Os pontos de paragem podem ser identificados com o comando `info break` (ou apenas `i b`) e retirados com `delete` (ou apenas `d`) seguido dos números dos pontos de paragem a retirar.

O principal comando para definir um ponto de paragem é o comando `break`. Este comando pode incluir uma condição, o que implica que o programa só é parado quando, ao passar no local indicado, a condição se verifica. No entanto, tal condição torna a

execução mais lenta. O local de paragem é especificado como argumento do comando `break`, e pode ser:

***sem argumento*:** introduz um ponto de paragem no local onde a execução se encontra suspensa neste momento. Este ponto pode não corresponder ao ponto do programa que está a ser visualizado ou inspeccionado.

nome de função: o ponto de paragem é colocado antes da primeira instrução executável da função indicada. O nome da função pode ser precedido do nome do ficheiro em que a função se encontra, caso exista mais de uma função (necessariamente não global) com o mesmo nome. Por exemplo, `break file.c:main`.

número de linha: o ponto de paragem é colocado antes da primeira instrução executável da linha no código fonte do ficheiro em análise. O número pode ser precedido do nome do ficheiro para designar a linha de um determinado ficheiro. Caso essa linha não contenha instruções executáveis, o ponto de paragem é colocado antes da primeira instrução executável seguinte. O número pode ser precedido de um sinal de '-' ou '+', designando um deslocamento face á linha actual.

Os pontos de paragem podem ser desactivados e reactivados com os comandos **enable** e **disable** que recebem os números dos pontos de paragem como argumentos.

Controlar a execução Uma vez que a execução do programa é suspensa num ponto de paragem, a execução pode ser controlada com diversos comandos. Notar que, antes de iniciar a execução deve ser colocado pelo menos um ponto de paragem, por exemplo `break main` permite controlar a execução desde a primeira linha do programa fonte.

Os comandos de controlo mais significativos são:

step: permite executar uma linha de código fonte (uma ou mais instruções de acordo com a sua disposição no ficheiro em análise), parando a execução antes da primeira instrução executável da linha seguinte. Este comando pode receber um argumento que equivale a repetir o comando o número de vezes indicado, por exemplo `step 5` avança cinco linhas antes de voltar a parar. O comando pode ser abreviado para apenas **s**. Notar que uma linha em branco, sem especificar nenhum comando, repete o comando anterior.

next: comando em tudo idêntico ao anterior, mas quando é encontrada uma chamada a uma função a chamada é efectuada mas o controlo continua na rotina chamadora. Isto é, a função é executada de uma só vez, como se de uma instrução máquina se tratasse, excepto se existirem pontos de paragem no seu interior ou em funções por

ela chamada. O comando pode ser abreviado para **n** e também recebe o número de repetições como argumento.

continue: continua até ao ponto de paragem seguinte. Pode ser abreviado para **c** e também recebe o número de repetições como argumento.

finish: permite executar até ao fim da função corrente e parar imediatamente antes de a função retornar.

return: retorna imediatamente à função chamadora, sem executar mais instruções, e retornando o valor da expressão indicada como argumento.

watch: permite parar a execução sempre que o valor da expressão, indicada como argumento, muda de valor. Este comando também pode incluir uma condição, como no caso do comando `break`.

Inspeccionar e modificar variáveis Até ao momento, conseguimos compreender o fluxo do programa, verificando quais as instruções que são executadas. O *debugger* permite ainda inspeccionar o valor das variáveis do programa e modificar o seu valor, sem necessidade de alterar o código fonte ou recompilar.

Para inspeccionar ou modificar uma variável, através do seu nome, esta deverá estar visível, de acordo com as regras de visibilidade da linguagem utilizada. Além das variáveis locais, e das variáveis globais que não tenham sido redefinidas por outras do mesmo nome, as variáveis activas de um programa podem ser visíveis de uma das funções presentemente invocadas. Para poder aceder a uma dessas funções existem os comandos **up** e **down** que permitem subir e descer na pilha de dados de acordo com os registos de activação definidos (*stack frames*). O comando **backtrace**, ou apenas **bt**, permite identificar os registos de activação activos.

Além do comando **print**, já abordado atrás, existe ainda o comando **printf** que recebe um formato e uma lista de argumentos como função de **C** homónima. O comando **x** permite inspeccionar zonas de memória através do seu nome ou endereço, e imprimir os seus valores em diversos formatos e bases de numeração.

O comando **set**, com a sintaxe '`variável = expressão`' permite modificar o valor da variável especificada no programa em execução, sem ter de modificar o código fonte. Notar que modificação apenas se reflecte na execução corrente e não no ficheiro original ou futuras execuções do mesmo programa, mesmo sem abandonar o *debugger*.

O comando **call** permite invocar uma função indicando os argumentos e obtendo o resultado dessa invocação. Notar que se essa função modificar variáveis globais, reservar ou libertar memória, abrir ou fechar ficheiros, *etc.* Essas alterações tornam-se efectivas para a execução corrente do programa em análise.

8.2 Análise de código máquina

Uma vez o compilador desenvolvido no âmbito da cadeira de Compiladores é muito simples, e certamente não gera informação simbólica de análise, pode ser necessário efectuar análises sem informação simbólica ou apenas com informação de ligação. Os comandos disponíveis são semelhantes aos anteriormente abordados, mas em vez de variáveis utilizam-se os registos da máquina e, em vez de a execução ser linha a linha, é efectuada instrução máquina a instrução máquina. Os comandos mais importantes são:

set disassembly-flavor: permite escolher a forma como o código é apresentado, podendo o argumento ser **att** ou **intel**. O formato **att** usa uma notação '*origem, destino*' (por exemplo `mov $0x0,%eax`) enquanto a notação **intel** usa o contrário (`mov eax,0x0`). Existem mais diferenças entre as duas notações, nomeadamente na especificação de endereços efectivos.

disassemble: equivalente ao comando **list** apresenta o código fonte, sob a forma de instruções máquina, no formato seleccionado pelo comando anterior.

info registers: imprime o valor dos registos mais usados da máquina. O comando **info all-registers** imprime todos os registos, incluindo aqueles que não directamente acessíveis ao programador.

stepi: tal como comando **step** permite avançar uma instrução, neste caso uma instrução máquina, ou o número especificado de instruções se for indicado como argumento. O comando pode ser abreviado para **si**.

nexti: semelhante ao comando **next** permite avançar uma instrução máquina não seguindo as chamadas a rotinas, podendo ser abreviado para **ni**. Também se pode especificar o número de instruções a executar antes de parar, se for indicado como argumento.

info reg: permite imprimir o valor dos registos, ou de um registo em particular, caso o seu nome seja indicado.

set: permite alterar o valor de um registo, como se de uma variável se tratasse, por exemplo, **set $eax = 0**.

x/`formato`: permite analisar a memória dado um endereço ou nome simbólico. O formato é constituído por um comprimento, uma letra de tipo e uma letra de dimensão, todos opcionais. O comprimento refere o número de entidades a analisar. O tipo pode ser `o(octal)`, `x(hex)`, `d(decimal)`, `t(binary)`, `a(address)`, `f(float)`, `u(unsigned)`, `i(instruction)`, `s(string)` ou `c(char)`. A dimensão de cada uma das entidades pode ser `b(byte)`, `h(halfword)`, `w(word)`

ou `g(giant, 8 bytes)`.
Por exemplo, **x/6dw main** imprime as primeiras 6 palavras (*word*s de 32 bits) como inteiros decimais.

display/i: permite imprimir o valor de registo sempre que a execução do programa pára. O comando pode ser desactivado com **undisplay** e usar **info display** para identificar os pedidos registados.

9 flex: gerador de analisadores lexicais

O `lex`, tal como o equivalente da GNU `flex`, são geradores de analisadores lexicais, ou seja, produzem programas capazes de fazer a análise de linguagens regulares. Uma linguagem regular, a forma mais simplificada de linguagens, pode ser descrita exclusivamente à custa de expressões regulares. Para tal o `lex` produz, a partir de uma descrição num ficheiro com a extensão **.l**, uma máquina de estados, designado por autómato finito, que quando executado processa a linguagem descrita no ficheiro **.l**. Um autómato finito, além de ter um número finito de estados, memoriza apenas o estado actual, tomando as decisões apenas com base nesse estado e no próximo carácter de entrada. Por exemplo, se estou no estado 5 e li o carácter 'x' passo para o estado 9. Os autómatos finitos têm a vantagem de serem simples, eficientes e compactos quando comparados com outras formas de análise de linguagens, por exemplo os analisadores sintácticos.

Uma gerador de analisadores lexicais pode ser usado para processar linguagens muito simples, as linguagens regulares, ou como primeiro passo na análise de linguagens mais complexas. No segundo caso, o gerador de analisadores lexicais agrupa um conjunto de caracteres, descritos por expressões regulares, e passa esse grupos (ou *tokens*) ao analisador sintáctico.

Um ficheiro de `lex` deve ter a extensão **.l** e inclui três secções, podendo a última ser omitida:

```
zona de declarações
%%
zona de regras
%%
zona de rotinas
```

A geração do analisador lexical é conseguida em dois passos. No primeiro, a ferramenta `lex` produz uma autómato finito e algumas rotinas auxiliares em **C** num ficheiro designado por **lex.yy.c**, a partir do ficheiro que contém a descrição da linguagem (com a extensão **.l**). No segundo passo, produz-se o analisador lexical compilando o ficheiro **lex.yy.c**.

prompt$ lex scanner.l

prompt$ cc lex.yy.c -ll

prompt$ a.out < linguagem

De uma forma geral, o analisador lexical gerado pelo `lex`, designado por **yylex()**, pode ser invocado a partir qualquer rotina em **C**. Na realidade a biblioteca de lex, designada por **-ll** (**-lfl** no caso do `flex`), inclui apenas duas rotinas:

```
int main() { while (yylex() != 0); return 0; }
int yywrap() { return 1; }
```

9.1 Zona das regras

A zona das regras permite definir um conjunto de linguagens regulares, permtindo associar a cada uma dessas expressões regulares uma acção semântica descrita numa linguagem de apoio. Nos exemplos a linguagem usada será **C**, mas também se pode usar **C++**, por exemplo. A acção semântica realiza a operação de interpretação desejada, na sequência do reconhecimento da expressão regular.

9.1.1 Expressões regulares

Uma expressão regular permite definir uma linguagem regular, usando para tal um alfabeto (conjunto de símbolos reconhecidos). Uma expressão regular define uma linguagem com base nas regras:

- $\emptyset$ é uma expressão regular que define a linguagem vazia.
- ε é uma expressão regular que define a linguagem que consiste na frase nula $\{\varepsilon\}$.
- um símbolo do alfabeto é um expressão regular que define a linguagem formada pela frase constituída por esse símbolo.
- um conjunto de operadores sobre expressões regulares. Considerando duas expressões regulares p, q:

  escolha: designado por $p|q$ é comutativo e associativo, representando a união das expressões regulares originais.

  concatenação: designado por $p\ q$ é associtivo e distributivo em realção à escolha e mais prioritário que a escolha. ε é elemento neutro na concatenação.

  fecho de Kleene: designado por $p*$ é idempotente ($p** = p*$) e mais prioritário que a concatenação e representa o conjunto de frases constituídas por zero ou mais repetições das frases de p. Além disso $p* = (p|\varepsilon)*$.

  parênteses: permite alterar a prioridade dos operadores anteriores.

Além dos operadores base, é comum usar algumas extensões aos operadores que tornam mais sucinta e cómoda a especificação das expressões regulares:

opção: designado por $p?$ designa zero ou uma ocorrências de p.
É equivalente a $p|\varepsilon$.

fecho transitivo: designado por $p+$ designa uma ou mais ocorrências de p.
$p+$ é equivalente a $p\ p*$, e $p*$ é equivalente a $p + |\varepsilon$.

parênteses rectos: designado por $[pq]$ ou $[p - q]$ designam $p|q$ e os símbolos de ordem entre p e q (inclusivé).

9.1.2 Definição de expressões regulares em LEX

A sintaxe, ou conjunto de regras, de uma linguagem regular permite identificar se uma frase, ou sequência de símbolos do alfabeto, pertence à linguagem. Contudo, de um ponto de vista prático, torna-se necessário associar acções semânticas a subconjuntos da linguagem.

Por exemplo, numa linguagem que aceite qualquer sequência de vogais pode ter interesse definir acções dependendo da vogal encontrada. Assim, embora a sintaxe da linguagem das vogais seja `a|e|i|o|u` o **lex** torna possível separar a expressão regular em grupos para o seu processamento. Notar que se pode separar cada vogal, ou tratar `a|i` separadamente de `e|o|u`, ou manter uma única expressão regular, caso não haja interesse em distinguir as acções semânticas.

Em **lex**, a zona das regras é constituída por uma expressão regular por linha, separada da acção semântica por um ou mais espaços brancos (ou tabuladores horizontais). Caso a acção semântica ocupe mais de uma linha deverá ser delimitada por chavetas. Por exemplo,

```
%%
a|e|i printf("Encontrei 'a', 'e' ou 'i'\n");
o|u printf("Encontrei 'o' ou 'u'\n");
```

O **lex** acrescenta mais alguns operadores às expressões regulares para faciltar a descrição das linguagens, nomeadamente para quem utiliza **C**:

x	O carácter 'x'
"x"	O carácter 'x', mesmo que seja um carácter especial
$\backslash x$	O carácter 'x', mesmo que seja um carácter especial
x\$	O carácter 'x' no fim da linha
$\hat{}x$	O carácter 'x' no início da linha
$x?$	Zero o uma ocorrência de 'x'
$x+$	Uma ou mais ocorrência de 'x'
$x*$	Zero ou mais ocorrência de 'x'
xy	O carácter 'x' seguido do carácter 'y'
$x\|y$	O carácter 'x' ou o carácter 'y'
[az]	O carácter 'a' ou o carácter 'z'
[a-z]	Do carácter 'a' ao carácter 'z'
[^ a-z]	Qualquer carácter excepto de 'a' a 'z'
$x\{n\}$	'n' ocorrências de 'x'
$x\{m,n\}$	de 'm' a 'n' ocorrências de 'x'
x/y	'x' se seguido por 'y' (só 'x' faz parte do padrão)
.	Qualquer carácter excepto $\backslash n$ (*newline*)
(x)	O mesmo que 'x', parenteses alteram a prioridade dos operadores
$<< EOF >>$	Fim do ficheiro

No caso dos grupos de caracteres especificados por parenteses rectos é conveniente ter especial cuidado em casos do tipo:

$[a\backslash -z]$	Os caracteres 'a', '-' ou 'z'
$[-az]$	Os caracteres 'a', '-' ou 'z'
$[a-c\hat{}b]$	Os caracteres 'a', 'b', 'c' ou '^'
$[a\|z]$	Os caracteres 'a', '|' ou 'z'
$[.\|\backslash n]$	Os caracteres '.', '|' ou '$\backslash n$'

9.1.3 Tratamento de expressões regulares em LEX

Para identificar qual a acção semântica a executar, quando mais de uma expressão regular é válida, o **lex** usa as seguintes regras:

- A sequência de entrada mais comprida é a escolhida.
- Em caso de igualdade de comprimento é usada a que se encontra primeiro no ficheiro de especificação.

Notar que não se trata da expressão regular maior, mas da sequência de entrada maior, por exemplo em

```
%%
dependente printf("Encontrei 'dependente'\n");
[a-z]+   ECHO;
```

apenas a palavra 'dependente' é reconhecida pela primeira regra, enquanto 'dependentes' ou 'interdependente' são reconhecidas pela segunda regra. Caso as regras estivessem pela ordem trocada o **lex** avisava que a segunda regra, no caso a expressão 'dependente', nunca seria identificada pois mesmo quando a entrada era 'dependente' qualquer das duas regras identificava a sequência e era usada a primeira. Desta forma, as regras mais específicas devem aparecer sempre primeiro na zona das regras do ficheiro **lex** e as mais genéricas sempre no fim da especificação.

9.1.4 Funções utilizadas

int yylex(void) : rotina, gerada pelo **lex**, que realiza a análise lexical. Devolve o número do elemento lexical encontrado ou 0 (zero) quando atínge o fim do processamento. Quando o **yylex()** funciona autónomo, as acções semânticas não incluem instruções de `return`, e o **yylex()** retorna uma única vez no fim do processamento, com o valor 0 (zero). Quando o **yylex()** funciona em conjunto com outras ferramentas, por exemplo um analisador sintáctico como o **yacc**, as acções semânticas que produzem elementos lexicais relevantes para a análise sintáctica (excluem-se, em geral, espaços brancos e comentários) incluem uma instrução de `return`. Esta instrução faz a função **yylex()** terminar, devolvendo o valor da expressão do `return`. O valor dsevolvido deverá ser sempre diferente de 0 (zero), para não se confundir com o fim de processamento, e distinto para cada tipo de elemento lexical, por forma a permitir a sua identificação pela ferramenta que invocou o **yylex()**. Assim, esta ferramenta deverá invocar a rotina **yylex()** sempre que pretende obter um novo elemento lexical.

int yywrap(void) : rotina, escrita pelo programador (pode-se usar a realização por omissão definida na biblioteca **-ll**), que quando um ficheiro chega ao fim permite continuar o processamento noutro ficheiro. Caso não haja mais ficheiros a processar **yywrap()** devolve 1 (um), caso contrário actualiza a variável **yyin** para o ficheiro seguinte e devolve 0 (zero). Por exemplo, ao terminar o processamente de um ficheiro de declarações em **C** (**.h**) pode-se retornar ao ficheiro anterior, que o incluia, desde que no processamento da directiva `#include` se tenha salvaguardado o respectivo ponteiro **FILE***.

void yymore(void) : rotina, invocada numa acção semântica, que permite salvaguardar o texto reconhecido pela expressão regular para seja concatenado com a expressão regular seguinte. Esta rotina pode ser invocada em diversas acções semânticas e, no limite, se for invocada em todas as acções semânticas permite guardar a totalidade do ficheiro lido.

void yyless(int) : rotina, invocada numa acção semântica, que permite considerar apenas os primeiros **n** carácteres de **yytext**, sendo os restantes reconsiderados para processamento. Por exemplo, se dos 10 caracteres reconhecidos pela expressão regular se pretender consumir apenas os primeiros 4 e permitir que os restantes 6 sejam novamente emparelhados com outras expressões regulares, basta fazer **yyless(4)**.

9.1.5 Variáveis globais

char yytext[ ] : cadeia de caracteres que contém o texto reconhecido pela expressão regular, ou acumulado de outras regras através de sucessivas invocações a **yymore()**. Notar que embora a cadeia de caracteres termine em 0 (NULL) pode conter, no seu interior, outros 0 (zero).

int yyleng : comprimento da cadeia de caracteres que contém o texto reconhecido. Pode não corresponder a **strlen(yytext)** se a cadeia de caracteres contiver NULLs.

int yylineno : número de linha do ficheiro de entrada onde se encontra o último carácter reconhecido pela expressão regular. Esta variável só existe definida em **flex** se este for invocado com a opção **-l** ou o ficheiro **.l** contiver uma declaração **%option yylineno** ou **%option lex-compat**.

FILE *yyin : ponteiro para o ficheiro de onde são lidos os carácteres a analisar.

FILE *yyout : ponteiro para o ficheiro de onde é escrito o texto através da `macro` **ECHO**, ou outro texto que o programador deseje.

YYSTYPE yylval : variável que transporta o valor do elemento lexical reconhecido para outra ferramenta. (ver 9.4).

9.1.6 Macros predefinidas

ECHO : imprime o texto reconhecido pela expressão regular, ou acumulado de outras regras através de sucessivas invocações a **yymore()**. Na realidade está definido como

```
#define ECHO fwrite(yytext, yyleng, 1, yyout)
```

REJECT : depois de processada a acção semântica que inclui a chamada ao **REJECT** o processamento recomeça no início do texto reconhecido pela regra corrente (ou seja, **yyless(0)**), mas a regra actual não é reconsiderada para o texto actual. Assim, serão emparelhadas regras mais genéricas.

9.1.7 Acesso directo a funções de entrada/saída

int input(void) : esta rotina permite ler o carácter seguinte, a partir do ficheiro de entrada, sem que seja processado pelo analisador lexical. Caso o ficheiro corrente chegue ao fim, tem efeito o processamento habitual de invocar a rotina **yywrap()** para que seja encontrado o caracter seguinte, pelo que o valor `-1` (fim de ficheiro) é apenas devolvido no fim do precessamento. Notar que a utilização desta rotina apenas se justifica caso não seja possível efectuar parte do processamento da linguagem através de expressões regulares.

void output(int) : imprime o carácter em **yyout**. Esta rotina não é suportada pelo **flex**.

void unput(int) : recoloca o carácter passado como argumento para processamento pelas expressões regulares seguintes. Notar que caso se pretenda recolocar diversos caracteres este devem ser recolocados pela ordem inversa, ou seja, o primeiro carácter a ser reprocessado deve ser o argumento do último **unput()**.

9.2 Zona de declarações

A zona de declarações permite definir elementos a ser utilizados na zona das regras. As declarações incluem declarações de código, definição de substituições, declaração de condições e activação de opções.

As declarações de código são delimitadas pelas sequências '`%{`' e '`%}`' devendo o carácter '`%`' situar-se na primeira coluna da linha. No interior podem-se declarar variáveis, estruturas, tipos de dados ou incluir ficheiros de declarações que sejam relevantes para o código descrito nas acções semânticas. Notar que as declarações de código também podem conter o código propriamente dito, e não apenas as suas declarações, mas tal procedimento é altamente desaconselhado. Embora não haja justificação para tal, podem-se incluir várias declarações de código na zona de declarações.

9.2.1 Definição de substituições

Com objectivo de simplificar a escrita as expressões regulares é possível definir um conjunto de subtituições, que podem ser posteriormente utilizadas com vantagem nas expressões regulares. A definição de uma substituição, uma por linha, é constituída por um identificador de substituição separado por um ou mais espaços brancos (ou tabuladores horizontais) da expressão regular. A utilização da substituição nas expressões regulares subsequentes, quer na zona de declarações quer na zona de regras, é efectuada colocando o identificador de substituição entre chavetas. Desta forma, as chavetas são caracteres especiais em **lex** pelo que a sua utilização literal deve ser colocada dentro de aspas ou precedida de '\'.

```
LET [a-zA-Z]
DIG [0-9]
INT {DIG}+
EXP [Ee][+-]?{INT}
REAL {INT}"."{INT}({EXP})?
```

9.2.2 Agrupamentos

Como a ferramenta **lex** modela o analisador lexical como um autómato finito é possível definir um conjunto de estados com outras regras que apenas podem ser acedidos em certas situações. Para tal o **lex** cria agrupamentos de estados identificados por um nome simbólico. As expressões regulares precedidas desse nome simbólico entre '<' e '>' só são utilizadas apenas depois de uma acção semântica iniciar o agrupamento com a acção 'BEGIN'. O agrupamento '0' (zero) ou 'INITIAL' representa as expressões regulares sem agrupamento explícito. Os agrupamentos são identificados, na zona de declarações, com '%s' ou '%start' seguido de uma lista de identificadores usados para designar agrupamentos.

Por exemplo, para imprimir apenas a zona das regras de um ficheiro **lex**

```
%s IN
%%
<IN>.|\n ECHO;
<IN>^"%%" BEGIN INITIAL;
^"%%" BEGIN IN;
.|\n ;
```

Notar que quando o agrupamento 'IN' está activo as regras do agrupamento 'INITIAL' mantêm-se activas, pelo que as expressões regulares dependentes de um ou mais agrupamentos devem preceder, no ficheiro **lex**, as do agrupamento 'INITIAL'. No entanto, as expressões regulares de um agrupamento não interferem uma nas outras. Notar que uma acção de 'BEGIN' transita para outro agrupamento mas não guarda o anterior, nem é possível retornar ao agrupamento anterior de uma forma implícita (a variável `yy_current_state`, definida em **flex**, não se encontra documentada não sendo portável para outros analisadores lexicais equivalentes).

Como um autómato finito é uma máquina de estados podemos modelar um semáforo em **lex**

```
%s VERDE AMARELO VERMELHO
%%
inicio |
<AMARELO>muda                       printf("Vermelho\n");   BEGIN VERMELHO;
<VERMELHO>muda                      printf("Verde\n");      BEGIN VERDE;
<VERDE>muda                         printf("Amarelo\n");    BEGIN AMARELO;
<VERDE,AMARELO,VERMELHO>panico      printf("Vermelho\n");   BEGIN VERMELHO;
avariado                                                    BEGIN INITIAL;
fim                                 return 0;
.|\n                                ;
```

Os agrupamento definidos em **lex** exigem muito cuidado para garantir que as expressões regulares globais, as do agrupamento 'INITIAL', não entram em conflito com nenhum dos agrupamentos definidos. Para tal é necessário garantir que se duas expressões regulares emparelham a mesma sequência de entrada a do agrupamento global encontra-se depois no ficheiro **lex**. Tal é facilmente conseguido declarando todas os agrupamento primeiro e o agrupamento global no fim. No entanto, se no agrupamento global houver uma expressão regular que emparelhe uma sequência maior, então esta última será preferida a qualquer uma existe nos outros agrupamentos. Assim, ao remover comentários em **C** não basta ignorar todos os caracteres isoladamente ($.|\backslash n;$), mas também os identificadores, cadeias de caracteres, números inteiros, *etc.* Notar que não se torna necessário ignorar as palavras reservadas pois, caso não exista uma regra para tal, a expressão regular dos identificadores cobre todos os casos.

Agrupamentos exclusivos em *flex* Para obviar ao problema enunciado no parágrafo anterior, a ferramenta **flex** inclui agrupamentos exclusivos, declarados com '%x', além dos habituais '%s' do **lex**. A única diferença nestes agrupamentos reside no facto apenas as regras do agrupamento estarem válidas. Desta forma, basta garantir que existe uma regra '$.|\backslash n$' em cada um deles. Esta simples diferença simplifica significativamente a especificação de linguagens mais complexas, com muitas regras e agrupamentos.

Uma primeira aproximação à remoção dos comentários em **C** pode ser conseguida através do exemplo seguinte.

```
%x COM
%%
"/*" BEGIN COM;
<COM>"*/" BEGIN INITIAL;
<COM>.|\n ;
.|\n ECHO;
```

Questão: identifique os casos não cobertos por esta solução e corrija a especificação.

9.3 Zona de código

Esta zona inicia-se no segundo **'%%'** e termina no fim do ficheiro **.l**. Como esta zona é facultativa, toda ela incluindo os **'%%'** podem ser omitidos, como ilustram os exemplos anteriores.

Caso seja incluída, esta zona contém código na linguagem de apoio, ou seja **C** nos exemplos deste texto. Nesta zona podem-se incluir rotinas auxiliares que serão copiadas integralmente para o ficheiro resultante **lex.yy.c**, podendo incluir a rotina de entrada da aplicação: **main**.

9.4 Ligação com outras ferramentas

Um analisador lexical pode ser utilizado como um primeiro passo no processamento de uma linguagem. Neste caso, a ferramenta **lex** permite que a análise seja feita por sub-conjuntos da linguagem, a pedido da aplicação que usa o analisador lexical. O caso mais comum, e que estudaremos em maior detalhe, corresponde à ligação a um analisador sintáctico. Um analisador sintáctico permite processar linguagem independetes do contexto, mais complexas que as linguagens regulares. Um analisador sintáctico pode processar qualquer linguagem regular, embora um analisador lexical não tenha capacidade de processar uma linguagem independente do contexto. Assim, uma linguagem não regular só pode ser parcialmente processada por um analisador lexical. Nomeadamente, aquelas partes da linguagem que conseguem ser expressas sob a forma de expressões regulares.

Nestes casos usa-se um analisador lexical em conjunção com um analisador sintáctico porque permite:

modularidade: permite separar a sintaxe em duas fases distintas: análise lexical e sintáctica. Diversas alterações podem ser efectuadas num analisador sem implicações no outro.

legibilidade: expressões regulares são, em geral, mais legíveis que regras gramaticais.

simplicidade: a separação em análise lexical e sintáctica permite que o analisador sintáctico seja significativamente mais simples.

eficiência: Notar que um analisador sintáctico é muito menos eficiente a processar expressões regulares, em termos de espaço ocupado e tempo de execução, que um analisador lexical.

portabilidade: variações entre ambientes, dispositivos ou sistemas operativos podem ficar contidos no analisador lexical.

Quando um analisador lexical é utilizado como pré-processador de um analidor sintáctico as suas tarefas são essencialmente:

- Identificação de elementos lexicais (tokens): literais, palavras reservadas, identificadores, operadores, separadores e delimitadores.
- Tratamento de caracteres brancos e comentários.
- Manipulação dos atributos de alguns dos elementos lexicais.
- Utilização da tabela de símbolos para classificar e guardar informação auxiliar de alguns elementos lexicais.
- Análise do grafismo (posicionamento do elementos em linha e coluna) do programa.
- Processamento de macros.
- Tratamento de erros, em geral ignorando os caracteres, embora este tipo de erros seja raro.

Numa análise lexical deste tipo torna-se necessário identificar dois tipos de construções:

elementos a ignorar : as acções semânticas associadas às respectivas expressões regulares são processadas localmente, ou seja, não retornam. Por exemplo, comentários ou espaços brancos são ignorados pela maioria das linguagens.

elementos lexicais úteis (*tokens*) : conjuntos de 1 ou mais carácteres que, podendo ser expressos por expressões regulares, têm significado para a análise lexical.

No caso dos *tokens*, o analisador lexical devolve o código ASCII do carácter quando isolado (representado por valores de 1 a 255), por exemplo, parenteses, chavetas ou alguns dos operadores mais comuns. Caso se trate de conjunto de caracteres utiliza-se um número inteiro distinto e superior a 256. Por exemplo, palavras reservadas, identificadores, números inteiros ou reais e operadores com mais de um carácter ($>=$, por exemplo). O valor 256 é usado para erro e o valor 0 (zero) para fim de processamento. O valor é devolvido pela instrução de `return` do **C**, que retorna da rotina **yylex()** onde são incluídas as acções semânticas.

Caso o elemento lexical necessite de indicar um valor além do elemento encontrado, a variável `yylval` deve ser preenchida com esse valor. Notar que a identificação de um número inteiro, através da expressão regular $[0-9]+$, pode fazer a rotina **yylex()** retornar INTEIRO, por exemplo, mas o valor do inteiro reconhecido necessita ser passado na variável auxiliar **yylval**. Esta variável é, em geral, definida como uma **union** em **C** para poder transportar os diversos tipo de dados, dependendo o tipo efectivamente preenchido do valor devolvido por **yylex()**. Esta **union** pode ser definida automaticamente pelos analidadores sintácticos. O analisador sintáctico **yacc**, por exemplo, gera um ficheiro designado por **y.tab.h** que contém a referida **union** e o valor das constantes maiores que 256, quando executado com a opção **-d**.

```
%{
#include "y.tab.h"
%}
%%
">" return '>';
[-+/*=] return yytext[0];
">=" return GE;
while return WHILE;
[0-9]+ yylval.i = atoi(yytext); return INTEIRO;
\"[^"]*\" yylval.s = strdup(yytext); return STRING;
[a-zA-Z]+ yylval.s = strdup(yytext); return IDENT;
```

onde o fcheiro **y.tab.h** tem um aspecto semelhante a

```
#define INTEGER 257
#define ID 258
#define STRING 259
#define WHILE 260
#define GE 261
typedef union { int i; char *s; } YYSTYPE;
extern YYSTYPE yylval;
```

9.5 Eficiência de processamento

O processamento das expressões regulares pela ferramenta **lex** é efectuado, como foi referido no início desta secção, por um autómato finito. Sendo assim, por cada carácter do ficheiro a processar o autómato evolui para o estado seguinte. Alguns desses estados têm acções semânticas associadas que são executadas sempre que o respectivo estado é atíngido. Desta forma, o tempo de processamento de um ficheiro é, do ponto de vista do autómato excluindo as acções semânticas, proporcional ao número de caracteres no ficheiro.

Na realidade, o aumento do número ou complexidade das expressões regulares apenas pode aumentar o número de estados do autómato e a respectiva dimensão, mas não o tempo de processamento. Assim, tudo o que poder ser tratado por expressões regulares deve sê-lo. Em particular, o processamento de uma linguagem "*passa-tudo*", com uma única regra '$.|\backslash nECHO;$', executa o autómato no mesmo tempo de um analisador lexical para a linguagem **C**, se exceptuar-mos o tempo gasto nas acções semânticas. Logo, quanto menos e mais eficientes forem as acções semânticas mais eficiente será a análise. Desta forma, as acções semânticas devem ser o mais simples possível, sendo o tratamento efectuado quase totalmente pelas expressões regulares.

Por exemplo, em vez de detectar palavras completas e depois detectar se se tratam de palavras reservadas ou identificadores

```
%%
[a-zA-Z_][a-zA-Z_0-9]*  { int resw;
                          if ((resw = findKeyword(yytext)) == 0)
                            return IDENTIFIER;
                          return resw;
                        }
```

devem-se indicar as palavras reservadas no topo nas regras e, depois de indicadas todas as palavras reservadas (ver 9.1.3), indicar a expressão regular para os identificadores

```
%%
if                  return IF;
elif                  return ELIF;
else                  return ELSE;
...
[a-zA-Z_][a-zA-Z_0-9]* return IDENTIFIER;
```

Notar que a ordem pela qual as palavras reservadas aparecem no ficheiro de especificação **lex** (com a extensão **.l**) é irrelevante, desde que antecedam a expressão regular dos identificadores. Nomeadamente, no exemplo acima, a sequência '`if`' só emparelha com a primeira se aparecer isolado, pois caso seja '`elif`' emparelha com a segunda e '`fifo`' apenas com a última. Notar que em qualquer dos 3 exemplos de '`if`'s a última regra emparelha sempre, mas é encoberta pelas regras acima com a excepção da sequência '`fifo`', que não é uma palavra reservada.

No entanto, deve ser tido especial cuidado em evitar situações que obrigem o analisador a recuar ('*backtracking*'). Tais situações acontecem quando uma expressão comprida é processada quase na totalidade mas, ao não ser completada, obriga o analisador a recuar para emparelhar com outras expressões regulares. Notar que tal não acontece no exemplo anterior pois a sequência `eli` não é suficiente para ser reconhecida como um `elif` nem necessita recuar para considerar um `if`, pois é reconhecido como um identificador `eli`. Situações que obriguem a recuo incluem por exemplo

```
%%
ah                return SARCASMO;
ahahah            return RISADA;
```

estas situações são raras e podem, em geral, ser evitadas através da inclusão de novas regras.

9.6 *Debug* de um analisador lexical

O **flex** permite gerar um analisador lexical com suporte para *debug*. Neste modo, activado com a opção **-d** na execução do comando **flex**, o autómato gerado pode imprimir mensagens de estado à medida que processa os caracteres do ficheiro da linguagem a analisar. Se a variável `yy_flex_debug`, apenas disponível quando o analisador é gerado com a opção **-d**, for colocada a 1 (um) o analisador informa quais as regras emparelhadas e a sequência de entrada que orginou esse emparelhamento.

```
extern int yy_flex_debug;
int main(int argc, char *argv[]) {
  yy_flex_debug = 1;
  while (yylex() != 0);
  return 0;
}
```

O modo de *debug* pode, por exemplo, ser activado através de um argumento

```
if (argc > 1 && strcmp(argv[1], "-d") == 0) yy_flex_debug = 1;
```

ou variável de ambiente

```
yy_flex_debug = getenv("DEBUG") ? 1 : 0;
```

9.7 jflex: análise lexical em Java

Embora existam outras ferramentas para a análise lexical em Java, como o JLex, a escolha do jflex justifica-se pelas semelhanças com a ferramenta flex para **C/C++** acima abordada. Além de ser considerado uma evolução do JLex, mantendo alguma compatibilidade, apresenta a possibilidade de ligação com as ferramentas de análise sintáctica CUP e BYacc/J. A ferramenta CUP tem a vantagem de, tal como o JLex ser totalmente escrito em **Java**, enquanto a ferramenta BYacc/J introduz a geração para **Java** na ferramenta BYacc, utilizada mais à frente para a análise sintáctica. O objectivo desta secção não é o descrever a ferramenta, mas apenas ressaltar as diferenças para o flex acima descrito.

Como em **Java** não existem rotinas isoladas, mas apenas classes, o analisador lexical jflex gera uma classe que realiza a análise. Por omissão, a classe gerada é designada por **Yylex**, correspondendo ao ficheiro Yylex.java, podendo o seu nome ser modificado com a opção **%class** seguida do nome desejado. As opções que controlam a geração da classe incluem:

%extends : seguido do nome da classe de onde a classe a gerar deve herdar;

%implements : seguido da lista, separada por vírgulas, das interfaces realizadas pela classe a gerar;

%public : declara a classe a gerar como pública;

%final : declara a classe a gerar como não podendo ser especializada (herdada);

%abstract : declara a classe a gerar como não podendo ser instanciada;

%apiprivate : declara os métodos gerados como privados, podendo ser públicos os incluídos pelo utilizador;

%scanerror : seguido do nome da excepção, declara o tipo de excepção a ser criada, em caso de erro interno (não para erros de análise lexical), sendo java.lang.Error por omissão;

%buffer : declara a dimensão inicial do tampão a utilizar;

%function : seguido do nome da função, declara o nome do método que contém o analisador, por oposição ao `yylex` usado por omissão;

%integer : ou apenas **%int**, declara o método que contém o analisador como devolvendo o tipo de dados **int**;

%type : declara que o tipo de dados, como no caso do inteiro acima, que o método que contém o analisador devolve;

%standalone : gera uma função `main` que, ao ser invocada, apenas imprime o texto não emparelhado, sem que os elementos lexicais sejam passados a outra ferramenta.

%debug : gera uma função `main` que, ao ser invocada, imprime os emparelhamentos efectuados durante o processamento, sem que os elementos lexicais sejam passados a outra ferramenta.

%cup : declara a classe gerada de forma compatível com o analisador sintáctico **CUP**, podendo as opções `%cupsym` e `%cupdebug` ser usadas neste modo;

%byacc : declara a classe gerada de forma compatível com o analisador sintáctico **BYacc/J**, devolvendo **0** (zero) no fim do ficheiro e a função `yylex` devolvendo valores inteiros;

Apesar de longa, a lista acima está incompleta, devendo sempre ser consultado o manual da ferramenta correspondente à versão instalada para outras opções.

Embora semelhantes, as ferramentas `flex` e `jflex` apresentam diferenças na própria estrutura do ficheiro de especificação. De facto, o código do utilizador, a ser colocado fora da classe gerada, é colocado antes do primeiro separador `%%` e não no fim do ficheiro, como no caso da ferramenta `flex`. Qualquer código que o utilizador deseje colocar dentro da classe, e que pode incluir construtores e variáveis ou métodos, deve ser colocada entre os delimitadores `%{` e `%}`. Igualmente importante é o facto de todas as acções semânticas deverem ser colocada entre chavetas, embora em `flex` tal só seja necessário se ocuparem mais de uma linha. Também, no caso da ferramenta `jflex`, é necessário utilizar um sinal de igual '`=`' para separar o nome da substituição ou macro da expressão regular a ser expandida. De igual forma, a expressão regular deve ser balanceada, isto é, não deixar parênteses não equilibrados, pois é analisada *de per si*.

Tal como no caso do `flex`, também todos membros da interface à disposição do utilizador começam pelo prefixo **yy**, mas neste caso fazem parte da classe gerada. Algumas das variáveis, como `yytext` e `yylineno` passaram a ser métodos, devendo o utilizador ter cuidado que qualquer cópia dos seus valores não reflecte posteriores alterações (óbvio!). As funções da interface disponibilizadas para as acções semânticas são:

String yytext() : o texto identificado pela expressão regular em causa;

int yylength() : o comprimento do texto identificado sem criar um objecto `String` para tal;

void yybegin(int agrupamento) : substitui a macro **BEGIN(agr)** do `flex`, designando `YY_INITIAL` o estado inicial;

char yycharat(int pos) : devolve o carácter na posição `pos` do texto identificado de uma forma mais rápida;

int yystate() : devolve o número do estado interno do analisador lexical gerado;

void yypushback(int number) : repõe no tampão, para nova análise, o número de caracteres indicados, sendo os valores devolvidos pelos métodos `yylength` e `yytext` actualizados em conformidade;

void yypushStream(java.io.Reader reader) : passa a considerar o argumento como o novo canal de leitura, tendo o cuidado de guardar numa pilha o canal de leitura actual, bem como o estado interno, linha, coluna e carácter de antevisão;

void yypopStream() : fecha o canal de leitura actual e repõe o canal de leitura anterior que estava no topo da pilha, em geral numa acção associada à expressão «`EOF`»;

boolean yymoreStreams() : indica se ainda existem mais canais de leitura na pilha;

void yyreset(java.io.Reader reader) : fecha o canal de leitura actual, substituindo-o pelo canal indicado. Toda as variáveis são reiniciadas e a informação existente no tampão perdida, não podendo o canal voltar a ser utilizado;

void yyclose() : fecha o canal de leitura, devolvendo fim-de-ficheiro todos os subsequentes pedidos de leitura;

A ligação com a ferramenta `BYacc/J` efectua-se da mesma forma indicada na secção 9.4 com a restrição que todos os elementos necessários não são globais. Como o analisador lexical, quando é utilizado por uma ferramenta de análise sintáctica, é manipulado por esta última, apenas a classe que realiza a análise sintáctica deve ser acessível às restantes partes da aplicação. Assim, o analisador lexical deve ser um membro privado da classe que realiza a análise sintáctica. De qualquer forma, também o analisador lexical necessita de uma referência ao analisador sintáctico para devolver os elementos lexicais e à variável `yylval` para devovler eventuais valores associados a esses elementos. A integração das duas ferramentas resume-se a: indicar o modo de compatibilidade através da opção `%byacc` e incluir entre os `%{` e `%}` uma variável privada da classe do analisador sintáctico e definir, na mesma zona, um construtor que permita receber e preencher

esse objecto. O analisador sintáctico declara a variável `yylval` e os elementos lexicais a devolver, por exemplo

```
import java.io.*;
%%
%byaccj
%{
private Parser parse; // analisador gerado pelo byaccj
public Yylex(Reader r, Parser p) { this(r); parse = p; }
%}
// agrupamentos e substituicoes
%%
"while"                 { return parse.WHILE; }
[A-Za-z][A-Za-z0-9_]*   { parse.yylval = new String(yytext());
                          return parse.VARIABLE; }
// outras regras
```

Por fim, é de referir mais algumas das opções de uso frequente:

%7bit : o analisador aceita apenas caracteres de 7 bits correspondente ao código ASCII, lançando a excepção `ArrayIndexOutOfBoundsException` se o carácter não tiver valores entre 0 e 127;

%8bit : ou **%full** para analisar caracteres com valores entre 0 e 255 usados em codificações como o **ISO-LATIN-1** ou as *'code pages'* do `DOS/Windows`, mas não o **unicode**. Notar que os caracteres com valores entre 128 e 255 dependem da codificação utilizada no ficheiro, sendo dependentes da configurações locais da máquina e, por vezes, do sistema operativo;

%16bit : ou **%unicode** para analisar ficheiros codificados com caracteres representados pela norma **unicode**;

%caseless : ou **%ignorecase** converte as maiúsculas para minúsculas, evitando considerar ambos os casos nas expressões regulares (por exemplo, `[aA]`);

%char : activa o contador de caracteres lidos, cujo valor é guardado na variável `int yychar`;

%line : activa o contador de linhas, cujo valor é guardado na variável `int yyline`;

%column : activa o contador de caracteres lidos por linha, cujo valor é guardado na variável `int yycolumn`;

%switch : controi o autómato finito determinista (DFA) com a estrura `switch` da linguagem, oferecendo uma boa compressão no ficheiro resultante combinado com um bom desempenho. Contudo, para autómatos com mais de 200 estados devem ser consideradas uma das opções de codificação abaixo, não só por o desempenho se degradar, mas especialmente devido à limitação de 64KB por método existente na linguagem **Java**;

%table : codifica o DFA como uma tabela, com alguma compressão, pelo processo clássico. Pelas mesmas razões apresentadas acima, também as tabelas estão limitadas a 64KB em **Java**;

%pack : comprime a codificação do DFA em uma ou mais tabelas, evitando o limite dos 64KB, necessitando de descomprimir as tabelas na primeira instanciação do analisador. Este é o processo utilizado por omissão.

10 byacc: gerador de analisadores sintácticos

A ferramenta **yacc** recebe uma descrição gramatical e produz uma rotina, designada **yyparse()**, capaz de efectuar a análise sintáctica de sequências de texto que obedecem à gramática especificada. Assim, **yacc** não é um analisador sintáctico mas um gerador de analisadores sintácticos.

A gramática admitida pelo gerador deverá ser LALR(1), um subconjunto de gramáticas livre de contexto. Caso a gramática especificada não seja LALR(1) serão reportados conflitos. Notar que, quando o **yacc** reporta conflitos ainda gera um analisador sintáctico sem conflitos, mas que não é uma gramática equivalente à especificada. Logo, só quando não existem conflitos é que podemos ter a certeza de ter um analisador que processa correctamente a gramática especificada.

A análise gramatical dirigida por tabela, utilizada pela ferramenta **yacc**, é eficiente. Uma vez que a construção da tabela é efectuada directamente a partir da descrição da gramática por uma ferramenta automática, é fácil de modificar e corrigir.

A versão que usaremos, designada por **byacc**, é disponibilizada em código fonte para diversas arquitecturas e foi desenvolvida em *Berkeley*.

10.1 Especificação da gramática

O ficheiro, com a extensão **.y**, que inclui a especificação da gramática, é dividido em três zonas – declarações, gramática e código (facultativo) – separados por uma linha contendo %%, tal como num ficheiro **lex**.

Por exemplo,

```
%token ID INT
%%
attrib: ID '=' expr
      ;
expr  : INT
      ;
expr  : ID
      | expr '+' expr
      | expr '-' expr
      ;
```

A zona da gramática é constituída por um conjunto de produções (ou regras), onde um símbolo não terminal situado à esquerda do separador '**:**', designado por alvo, pode ser substituído por uma sequência de símbolos (terminais e não terminais) do lado direito, finalizado pelo terminador '**;**'. Ao processo de substituição de um símbolo alvo por uma das suas sequências é chamado derivação (do símbolo alvo na regra), enquanto a substituição inversa é designada por redução (de uma regra num símbolo alvo). O primeiro símbolo alvo de uma gramática é designado por símbolo objectivo e representa, em última análise toda a gramática, sendo o ponto de partida da análise sintáctica.

Se várias sequências produzem o mesmo símbolo terminal, as alternativas podem ser colocada na mesma produção, separadas pelo símbolo '`|`'. Todos os símbolos que não surjem do lado esquerdo de uma produção (símbolo alvo) são considerados terminais, devendo a sua ocorrência ser identificada por um analisador lexical (por exemplo, gerado pela ferramenta **lex**). Se os símbolos terminais não corresponderem a um só carácter (7 ou 8 bits, não caracteres unicode) necessitam ser declarados com **%token**.

10.1.1 Recursividade

Gramáticas não recursivas são muito limitadas já que representam, embora de uma forma compacta, todas as possíveis combinações de símbolos de entrada no máximo uma só vez. Para permitir um número arbitrário de repetições é necessário incluir produções recursivas na gramática. A forma mais simples de incluir recursividades na gramática consiste na recursividade directa, onde o símbolo alvo faz também parte (pelo menos uma vez) de uma das sequências que deriva. No entanto, também é possível ter recursividades indirectas, envolvendo duas ou mais regras, onde o símbolo alvo só aparece ao fim da derivação das referidas regras.

A ferramenta **yacc** (como qualquer gramática LALR(1)) permite a inclusão de recursividades em qualquer ponto da regra mas, quando for possível, é desejável que a recursidade seja efectuada à esquerada. Por recursividade à esquerda entende-se o símbolo alvo surgir como primeiro elemento da sequência que deriva, ou seja, o mais à esquerda possível na regra. Por exemplo, uma lista de um ou mais identificadores (ID), separados por vírgulas, pode ser representado por

```
ids: ID
   | ids ',' ID
   ;
```

A utilização de recursividades à esquerda permite reduzir significativamente a utilização da pilha auxiliar, por parte do analisador gerado pelo **yacc**, permitindo maior número de recursões para uma mesma dimensão da pilha.

No exemplo anterior constam duas regras, a primeira serve de iniciação pois não contém recursividade, enquanto a segunda permite um número de repetições arbitrárias da sequência **',' ID**. Notar que a ordem das regras não é relevante.

Para efectuar, por exemplo, zero ou mais repetições da sequência **',' ID**, basta que a regra que não contém a recursividade (neste caso a primeira) seja vazia. Da mesma forma, caso se trate de um terminador em vez de um separador, por exemplo zero ou mais repetições de declarações terminadas por **';'**

```
decls: decls decl ';'
     | /* empty */
     ;
```

onde **decl** designa uma declaração (que pode ser especificada noutras regras como sendo de diversos tipos) e **decls** as repetições. Neste exemplo, a regra de iniciação foi colocada em segundo lugar para exemplificar a inexistência de ordem e o comentário, não sendo necessário, evita erros de interpretação. (em especial por parte de outros utilizadores)

10.1.2 Ambiguidade

Uma gramática é considerada ambígua, e consequentemente não é processável por processos deterministicos como as gramáticas LALR(1) usadas pela ferramenta **yacc**, se uma mesma sequência de entrada for reconhecível através de derivações distintas. Em resumo, uma gramática não ambígua reconhece uma sequência de entrada (ficheiro com a linguagem) com uma e uma só sequência de derivações.

Por exemplo,

```
decls: /* empty */
     | decl ';'
     | decls decl ';'
     ;
```

representa zero, uma ou mais declarações terminadas por **';'**, mas uma declaração pode ser reconhecida através da segunda regra ou através da primeira seguida da terceira. Como existem duas formas de chegar à solução, a gramática é ambígua (embora a linguagem que pretende descrever não seja ambígua) resultando num conflito.

Da mesma forma, a utilização de mais de uma recursividade na mesma regra pode originar conflitos. Por exemplo,

```
expr: INT
    | expr '-' expr
    ;
```

permite que a sequência de entrada **5 - 3 - 7** seja reconhecida de duas formas, consoante seja o primeiro ou segundo símbolo **expr** da regra de subtracção a incluir a outra subtracção. O resultado equivale a **(5 - 3) - 7** (ou seja, **-5**) no primeiro caso e **5 - (3 - 7)** (ou seja, **9**) no segundo caso. Notar que o valor entre parênteses representa a primeira aplicação da regra da subtracção. Notar que a gramática é ambígua, mesmo que o resultado seja idêntico, como no caso da soma. Para eliminar a ambíguidade é necessário escolher uma das soluções, neste caso a subtracção é matematicamente associativa à esquerda logo deve-se escolher a primeira derivação apresentada, tornando a outra impossível de acontecer,

```
expr: INT
    | INT '-' expr
    ;
```

Problemas semelhantes podem surgir não apenas em operadores, mas noutras partes da linguagem como, por exemplo, a instrução **if** (com e sem **else**) da linguagem **C**.

10.1.3 Associatividade e prioridade

A solução gramatical acima apresentada é a mais robusta, pois pode ser aplicada a qualquer tipo de gramática, mas a ferramenta **yacc** permite a definição de associatividades e prioridades como forma de eliminar certos tipos de conflitos. Contudo, é necessário ter em atenção que o facto de os conflitos desaparecerem, por imposição de prioridades ou associatividades, não significa que a gramática passe a reconhecer a linguagem pretendida. Tal expediente deve apenas ser utilizado onde o resultado corresponde efectivamente ao comportamento desejado para a gramática (ver 10.3.4).

As vantagens de manter a gramática ambígua, mas eliminar essa ambiguidade através de associatividades e prioridades, residem numa maior legibilidade da gramática e maior eficiência do analisador. A maior eficiência do analisador gerado traduz-se num menor número de regras e, consequentemente, num menor número de passos necessários para reconhecer uma mesma sequência. A desvantagem reside no facto de a gramática, por depender de declarações específicas da ferramenta **yacc**, não ser portável para outros ambientes.

A associatividade e prioridade em **yacc** é especificada na zona de declarações através da indicação do tipo de associatividade (esquerda **%left**, direita **%right** ou não associativo **%nonassoc**) dos operadores, um por linha, em linhas de prioridade crescente. Por exemplo,

```
%left EQ NE
%nonassoc LE GE '<' '>'
%left '+' '-'
%left '*' '/' '%'
%right POW
%nonassoc UMINUS
```

correspondendo às regras da gramática

```
expr: expr EQ expr
    | expr NE expr
    | expr LE expr
    | expr GE expr
    | expr '<' expr
    | expr '>' expr
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | expr '%' expr
    | expr POW expr
    | '-' expr %prec UMINUS
```

onde a ordem das regras é irrelevante e a regra é identificada pelo símbolo terminal que univocamente a distingue. Em caso de ambiguidade, como é o caso do operador **'-'**, usa-se uma declaração **%prec** na regra (ou regras) para as distinguir. Notar que os símbolos terminais incluídos em declarações de associatividade não necessitam ser redeclarados com **%token**.

Todos os símbolos correspondentes a regras com a mesma prioridade têm de ser declarados na mesma linha, caso contrário ficam com prioridades distintas. Numa mesma prioridade não pode haver mais de um tipo de associatividade, caso contrário a gramática fica automaticamente ambígua, daí a restrição imposta pela ferramenta **yacc**. Por regras não associativas entende-se a impossibilidade de repetir sucessivamente a regra, consumada na geração de um erro sintáctico durante o processamento do ficheiro de

entrada. Por exemplo, a entrada **4 == 5 == 6** produz um erro ao atingir a segunda ocorrência do operador, enquanto **4 + 5 + 6** é uma entrada válida.

Notar que outras regras, como o **if** na linguagem **C**, podem também ser reolvidos pelo mesmo processo, devendo o programador garantir que a prioridade e associatividade impostas correspondem ao comportamento pretendido no processamento das sequências de entrada.

10.2 Interligação com o *lex*

A interligação com o analisador lexical gerado pela ferramenta **lex** ou, para o caso, qualquer outro analisador lexical utilizado, baseia-se na invocação da função **yylex()**. Esta rotina deverá devolver o número do elemento lexical (ou *token*) encontrado, preenchendo a variável **yylval** caso o elemento lexical tenha algum valor associado.

Na realidade, não existe necessidade de ter um analisador lexical, podendo a função **yylex** reduzir-se a `yylex() { int ch = getchar(); return ch == EOF ? 0 : ch; }` . Contudo, neste caso o processamento de sequências simples, que podem ser especificadas por expressões regulares, torna-se mais trabalhoso, além de aumentar a complexidade, tempo de processamento e espaço ocupado pelo analisador sintáctico (quando comparado com a soma dos analidadores lexical e sintáctico em conjunto) e poder dar mais facilmente origem a conflitos gramaticais. Por exemplo,

```
while: 'w' 'h' 'i' 'l' 'e';
dig: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' ;
digs: dig | digs dig ;
```

Em qualquer dos casos, a função **yylex()** deve retornar o código do carácter encontrado (valores compreendidos entre 1 e 255, para `unsigned char`). Isto inclui os caracteres ASCII base (entre 1 e 127) e os caracteres acentudos (entre 128 e 255, se `unsigned`, ou -1 e -128, se `signed`). O valor 0 (zero) é utilizado para indicar o fim do processamento e o valor 256 para erro. Os valores maiores que 256 são reservados para elementos lexicais compostos, como palavras reservadas.

Para evitar ser o utilizador a definir os valores, a ferramenta **yacc** (não a ferramenta **lex**) pode ser instruída para os gerar a partir das declarações **%token** (e todas as outras que podem declarar símbolo não terminais como, por exemplo, as associatividades). Para tal basta incluir a opção **-d** na invocação do **yacc** (por exemplo, `yacc -d gram.y`, sendo gerado um ficheiro designado por **y.tab.h** (alguns analisadores, que não o **yacc** ou o **byacc** aqui referidos, geram o ficheiro **gram.h**). Este ficheiro inclui um conjunto de definições para os elementos lexicais compostos, por exemplo

```
#define ID 257
#define INT 258
#define GE 259
```

Este ficheiro deve depois ser incluído pela especificação lexical

```
%{
#include <stdio.h>
#include "tipos.h"
#include "y.tab.h"
%}
%%
...
```

Notar que só depois de gerado o ficheiro **y.tab.h** é que o analisador produzido pela ferramenta **lex** pode ser compilado, logo o comando **yacc** necessita ser executado **antes** do comando **lex**. Para que tal não falhe, e não estejamos a compilar a versão actual do analisador sintáctico com a versão anterior do analisador lexical, aconselha-se vivamente a utilização um gestor de configurações, como a ferramenta **make**, e a incluir a dependência acima:

```
...
y.tab.c: gram.y
        byacc -dv $<
lex.yy.c: scan.l y.tab.c
        flex -l $<
...
```

Notar que a dependência pode ser estabelecida com o ficheiro **y.tab.c** e não com o ficheiro **y.tab.h** desde que eles sejam gerados ao mesmo tempo, como é o caso. (outras soluções são também possíveis)

10.2.1 Passagem de valores

Caso os elementos lexicais identificados sejam, por exemplo, identificadores ou inteiros, não basta saber que se encontrou um inteiro, é também necessário saber qual valor do inteiro que se encontrou. Para tal é necessário definir os tipos de dados a utilizar e quais os elementos lexicais que os usam.

Se existir um único tipo de dados, o que só acontece em casos muito triviais, pode-se definir **YYSTYPE** como o tipo de dados desejado, por exemplo, `#define YYSTYPE`

`double`. Na maioria dos casos, diversos tipos de dados podem ser transferidos, consoante o elemento lexical encontrado. Para tal a ferramenta **yacc** tem a declaração **%union** que permite incluir todos os tipos de dados utilizados durante o processamento da gramática, sejam associados a elementos lexicais (símbolos terminais da gramática) ou associados a símbolos não terminais (internos ao processamento sintáctico e semântico). Por exemplo,

```
...
%union { int i; char *s; Node *n; }
%token<s> ID STR
%token<i> INT
%type<n> expr decl
...
```

determina três tipos de dados (notar que o tipo `Node` deverá ter sido previamente definido, por exemplo através da inclusão do ficheiro que o define, no caso `node.h`). Na definição dos símbolos terminais (*tokens*) indica-se o nome da variável da `union` a utilizar (não o tipo de dados) e no caso dos símbolos não terminais a mesma definição é efectuada através da declaração **%type**. A partir deste momento o analisador sintáctico já sabe qual a variável que contém o valor associado.

Notar que muitos elementos lexicais, como as palavras reservadas, continuam a não ter valor associado. Da mesma forma, aqueles elementos lexicais com valor associado podem ser declarados apenas por `%token` e ser-lhes posteriormente atribuído um tipo de dados, com `%type`, tal como no caso dos símbolos não terminais.

O ficheiro **y.tab.h** também irá incluir a declaração, em formato **C**, da `union` indicada e uma variável desse tipo designada por **yylval**. As acções da especificação **lex** deverão preencher o campo correspondente da variável **yylval** e retornar o elemento lexical correspondente:

```
...
[0-9]+ yylval.i = atoi(yytext); return INT;
[a-zA-Z][a-zA-Z0-9]* yylval.s = strdup(yytext); return STR;
"while" return WHILE;
";" return ';';
//.* ;
...
```

Notar que apenas regras lexicais que produzam elementos lexicais úteis à análise sintáctica devem incluir a instrução `return`, logo exclui-se comentário, espaços brancos,

etc. Os elementos lexicais sem valor associado limitam-se a retornar o elemento lexical correspondente. No caso de elementos lexicais com valor associado, este deve ser preenchido antes de retornar, já que as instruções após uma instrução de `return` nunca são executadas.

10.3 Identificação e resolução de conflitos

Nem todas as gramáticas livres de contexto são gramáticas LALR(1) e logo processáveis pelo **yacc**. Nomeadamente, nenhuma gramática ambígua é LALR(1), mas existem gramáticas não ambíguas que não são LALR(1). Quando tal acontece o **yacc** reporta um conflito e produz um analisador onde o conflito é retirado, ou seja, o analisador gerado nunca é ambíguo nem corresponde à gramática especificada. A remoção do conflito por parte do **yacc** não significa que o problema tenha sido eliminado, apenas que de entre vários candidatos a determinada posição na tabela de análise foi escolhido um, não necessariamente o que o utilizador desejaria. No entanto, o **yacc** permite escolher determinado candidato num conflito, por exemplo através da determinação de prioridades e associatividades, eliminando o conflito e permitindo gerar analisadores sintácticos LALR(1) não ambíguos até com base em gramáticas ambíguas.

Um analisador ascendente dirigido por pilha, como é o caso do analisador gerado pela ferramenta **yacc**, baseia-se em 5 acções simples de que salientamos o deslocamento e a redução. Um deslocamento representa a aceitação de um símbolo terminal (o seu deslocamento do ficheiro para a pilha interna do analisador gerado pela ferramenta **yacc**), enquanto uma redução significa o reconhecimento de uma regra completa (a sua redução ao símbolo alvo que a deriva). Os conflitos reportados podem ser do tipo deslocamento-redução (*shift-reduce*) ou redução-redução (*reduce-reduce*), não existem conflitos deslocamento-deslocamento.

Para poder identificar o conflito e a sua origem é necessário pedir à ferramenta (opção **-v**) que gera um ficheiro auxiliar, designado por **y.output**, que contém os estados internos da gramática. No final deste ficheiro vêm indicados o número de símbolos terminais e não terminais da gramática processada, bem como o número de regras e o número de estados dos analisador gerado, além dos conflitos, caso existam.

O primeiro passo consiste em identificar cada um dos estados com conflitos. Em cada estado são indicados os itens que constituem o estado do analisador. Por exemplo, no item `attrib: • ID '=' expr`, onde o ponteiro • indica a posição actual de processamento. Após a leitura do elemento lexical **ID** o ponteiro passa para a posição seguinte, ou seja, o item `attrib: ID • '=' expr`. Quando o ponteiro atínge a última posição, no item `attrib: ID '=' expr •`, a regra é reduzida para todos os símbolos

de *lookahead*. Os símbolos de *lookahead* são aqueles que, de acordo com a gramática em questão, podem aparecer após o símbolo alvo, neste caso, `attrib`.

Não existem conflitos de deslocamento-deslocamento, pois se um mesmo símbolo pode ser deslocado por duas ou mais regras, o estado seguinte é constituído pelo próximos itens dessas regras. Um conflito surje quando um dos itens do estado tem o ponteiro na última posição e um dos símbolos de *lookahead* é também usado no deslocamento ou como *lookahead* na redução de outra regra, respectivamente um conflito deslocamento-redução ou redução-redução.

10.3.1 Conflitos deslocamento-redução

Consideremos a gramática,

```
X: 'a' 'b' 'c'
 | Y 'b'
 | Y 'a'
 ;
Y: 'a'
 ;
```

que apresenta um conflito deslocamento-redução, pois depois de ler o primeiro **'a'** estamos num estado representado no ficheiro **y.output** por

```
1: shift/reduce conflict (shift 4, reduce 4) on 'b'
state 1
        X : 'a' . 'b'  (1)
        Y : 'a' .  (4)

        'b'  shift 4
        'a'  reduce 4
```

ou seja, no estado 1 (*state 1*) o item `X : 'a' •'b'` da regra **(1)** pode-se deslocar para o estado 4 (*'b' shift 4*) com o símbolo **'b'**. Da mesma forma o item `Y : 'a'` • da regra **(4)** pode reduzir com o símbolo de *lookahead* **'a'** (*'a' reduce 4*) e prosseguir o processamento na regra 3 (`X : Y•'a'`). Por outro lado, a regra **(4)** também pode reduzir com o símbolo de *lookahead* **'b'** e prosseguir o processamento na regra 2 (`X : Y•'b'`). Como o símbolo **'b'** pode ser utilizado quer para reduzir a regra 4 como para deslocar a regra 1 para item seguinte, existe um conflito:

```
1: shift/reduce conflict (shift 4, reduce 4) on 'b'
```

No caso de um conflito deslocamento-redução, a ferramenta **yacc** opta sempre por efectuar o deslocamento e nunca a redução, produzindo um analisador em conformidade com essa decisão, mas reportando o conflito. Notar que a opção tomada pela ferramenta pode ser a desejada em certos casos, mas a errada noutros.

Para resolver um conflito deslocamento-redução pode-se recorrer à aplicação de prioridades e associatividades, mas tal permite apenas optar por uma das soluções (deslocamento ou redução) de acordo com as prioridades e associatividades estabelecidas.

O princípio base para eliminar conflitos consiste em reduzir o número de regras e consequentemente o número de reduções, logo reduzindo a possibilidade de uma redução entrar em conflito com outras regras. Tal pode ser conseguido aplicando a propriedade distributiva, enumerando as possibilidades sempre que possível.

O símbolo não terminal **Y**, da gramática acima, pode ser eliminado, obtendo-se a gramática equivalente

```
X: 'a' 'b' 'c'
 | 'a' 'b'
 | 'a' 'a'
 ;
```

A utilização de regras vazias deve também ser evitada pois acarreta muitos símbolos de *lookahead* (todos os do símbolo alvo) aumentando a possibilidade de um desses símbolos entrar em conflito com alguma regra desse estado.

10.3.2 Conflitos redução-redução

Os conflitos redução-redução significam que duas regras pretendem reduzir com o mesmo símbolo de *lookahead*. Neste caso, a ferramenta **yacc** opta por dar prioridade à regra que aparece primeiro no ficheiro. Como a ordem das regras não é importante, excepto neste caso, a regra escolhida pode não ser a desejada. Além disso, a regra preterida pode nunca ser usada, ou não ser utilizada quando desejado.

Considere-se a gramática

```
S: X
 | Y
 ;
X: 'a' | 'b' ;
Y: 'a' | 'c' ;
```

que apresenta um conflito deslocamento-redução, pois depois de ler o primeiro **'a'** estamos num estado representado no ficheiro **y.output** por

```
1: reduce/reduce conflict (reduce 3, reduce 5) on $end
state 1
        X : 'a' .  (3)
        Y : 'a' .  (5)

        .  reduce 3
```

ou seja, no estado 1 (*state 1*) quer a regra 3 como a regra 5 podem ser reduzidas para o mesmo símbolo de *lookahead*, no caso **$end** (ou seja EOF).

Para resolver o conflito é necessário factorizar as regras comuns, fazendo-as subir na hierarquia, por exemplo,

```
S: 'a'
 | X
 | Y
 ;
X: 'b' ;
Y: 'c' ;
```

Contudo deixa de ser possível distinguir a ocorrência de **'a'** em **X** ou em **Y**, pelo que a distinção terá de ser efectuada pelas acções semânticas. Tal resulta da falta de flexibilidade de gramáticas com um só símbolo de antevisão como as LALR(1). No entanto, gramáticas mais poderosas produzem tabelas de enormes dimensões e acarretam tempos de processamento muito superiores.

10.3.3 Gramáticas não LALR(1)

Os exemplos apresentados são delibradamente simples, por forma a compreender o seu funcionamento, mas numa gramática complexa os problemas são semelhantes. A principal diferença consiste no facto de cada estado poder ter muitos itens e dezenas de símbolos de *lookahead*, mas a maioria nada tem a ver com o conflito em questão.

Finalmente, existem situações em que não é possível, através das transformações simples, como as acima descritas, obter gramáticas LALR(1) equivalentes. Neste casos pode ser necessário fazer o **yacc** trabalhar em colaboração com o **lex** e a tabela de símbolos para resolver ambiguidades, quer criando novos elementos lexicais, quer diferindo para as acções semânticas (análise semântica) certas características sintácticas.

A criação de novos elementos lexicais pode consistir na separação em diversos tipos de identificadores, consoante representam tipos de dados, variáveis, constantes, funções, *etc*. Este expediente permite cada novo elemento lexical seja usado em situações distintas, reduzindo a possibilidade de conflito. Claro que a criação de novos elementos lexicais só deve ser efectuada para os símbolos de *lookahead* envolvidos em conflitos, por forma a não complicar a gramática e o analisador gerado.

Uma outra solução, consiste em agrupar as regras que provocam o conflito e resolver o problema na análise semântica, onde não existem muitas das restrições impostas pelo processo de análise utilizado pela ferramenta **yacc**. Tal é o caso de operadores com tipos de dados distintos, por exemplo a soma entre diversos tipos de dados com promoção de tipos. Em vez de enumerar a soma entre inteiro e inteiro, entre inteiro e real, entre real e real, entre real e lógico, *etc.*, utiliza-se apenas o tipo valor (ou variável), que resulta numa só regra para a soma, resolvendo-se as várias soluções polimórficas para a operação soma na acção semântica, à custa de estruturas de controlo da linguagem **C** (por exemplo, **if**s encadeados ou **switch**).

10.3.4 Depuração da gramática

Para acompanhar o processamento de um ficheiro de entrada e verificar quais as regras usadas e seu emparelhamento, a ferramenta **yacc** pode gerar um analisador modificado para efeitos de depuração (*debug*). Para tal é necessário usar a opção **-t** na invocação do **yacc** ou definir a macro **YYDEBUG**, quer no comando de compilação (`gcc -c -DYYDEBUG y.tab.c`), quer dentro do próprio ficheiro de especificação **yacc** (usando `#define YYDEBUG 1`), na zona das declarações da linguagem.

Para que as mensagens de depuração sejam impressas é necessário colocar a variável inteira **yydebug**, só disponível quando o analisador foi gerado para depuração (`YYDEBUG=1`), a um valor diferente de zero (`yydebug=1;`). Além disso, passa a estar disponível um vector de cadeias de caracteres, designado por **yyname**, que contém o nome dos elementos lexicais indexado pelo número associado ao elemento lexical.

Ao invocar o analisador (`yyparse()`) serão impressas mensagens à medida que os símbolos vão sendo lidos do ficheiro de entrada e agrupados em regras da gramática especificada. É assim possível verificar se o comportamento do analisador, mesmo para gramáticas ambíguas ou com outro tipo de conflitos, é o desejado. Contudo, aconselha-se vivamente a não utilizar gramáticas com conflitos nem a eliminar esses conflitos cegamente através da introdução indiscriminada de prioridades e associatividades, já que a depuração aqui descrita cobre apenas os casos testados e não todos os casos possíveis.

10.4 Tratamento de erros

O tratamento de erros consiste em procedimentos alternativos quando um fcheiro de entrada não está de acordo com a linguagem identificada pela gramática especificada. Claro que uma vez detectado um erro sintáctico no ficheiro de entrada (*sintax error*) já não é, na maioria dos casos, possível produzir resultados correctos. Assim, o tratamento de erros permite, em geral, continuar o processamento para a identificação de mais erros e não para produzir resultados, pois estes dificilmente seriam correctos.

Na sequência de um erro sintáctico, o analisador gerado pela ferramenta **yacc**, incrementa a variável global inteira **yynerrs** e invoca a função (a definir pelo utilizador) **yyerror(char*msg)**, onde **msg** é o texto descritivo do erro encontrado (mensagem *sintax error*, quando a função é invocada pelo analisador).

Os erros podem igualmente ser gerados nas acções semânticas, através da invocação da macro **YYERROR**, cujo procedimento é igual aos restantes erros sintácticos detectados pelo analisador. Notar que embora se possam designar os erros detectados nas acções semênticas como `erros semânticos`, tratam-se efectivamente de erros sintácticos que não é possível detectar com as ferramentas de análise sintáctica actuais, como o analisador gerado pelo **yacc**.

10.4.1 Recuperação de erros

Tal como foi referido, é possível continuar o processamento de um ficheiro de entrada, mesmo quando se encontra um erro. Para tal é necessário ignorar parte da sequência de entrada, nomeadamente, desde o ponto onde foi detectado o erro até um ponto conhecido, em geral identificado por um ou mais símbolos terminais bem conhecidos. Na linguagem **C**, o terminador **';'** será, certamente, um bom candidato pois permite ignorar uma declaração ou instrução onde ocorra o erro, continuando o processamento nas seguintes. Claro que se se tratar de uma declaração, a sua posterior utilização da entidade mal declarada irá gerar mais erros devido a ter sido retirada pelo mecanismo tratamento de erros. Este facto é aliás frequente em certo tipo de erros na linguagem **C**, onde a correcção de um só erro pode eliminar todos os restantes.

Para poder recuperar de um erro, ou melhor, continuar o processamento após a detecção de um erro, devem ser introduzidas regras específicas para tal. Notar que estas regras podem introduzir conflitos, como quaisquer outras, pelo que se aconselha a construir a gramática sem processamento de erros, e só depois de ter eliminado todos os conflitos introduzir estas regras. É também desejável não abusar da recuperação de erros já que as regras de recuperação de erro irão competir entre si para determinar o ponto de recuperação, podendo o comportamento ser difícil de compreender e controlar.

Uma regra de recuperação de erro inclui o símbolo terminal reservado **error** (que não pode ser utilizado para outras situações), devendo ser seguido de um ou mais símbolos terminais que identifiquem o ponto do ficheiro de entrada a partir do qual o processamento deve ser continuado. Por exemplo,

```
decls: /* empty */
     | error ';'{ $$ = 0; yyerrok; }
     | decls decl ';'
     ;
```

permite processar zero ou mais declarações, eliminando em caso de erro, todo o texto desde o erro até ao próximo **';'** encontrado no ficheiro. Neste caso, pode-se considerar um declaração nula e é necessário indicar ao analisador que o erro foi recuperado, através da invocação da macro **yyerrok**.

Caso se pretenda reposicionar o ponto de leitura no ficheiro de entrada, quer através de sucessivas chamadas ao analisador lexical (`yylex()`), quer através da manipulação directa do ponteiro para o ficheiro de entrada (`yyin`), deve-se invocar a macro **yyclearin** para que o analisador reponha o símbolo de antevisão e o restante estado do analisador. Esta invocação deve suceder a invocação da macro **yyerrok** anteriormente referida. No entanto, este procedimento deve ser evitado já que é frequentemente mal executado, sendo preferível usar os mecanismos disponibilizados pela ferramenta.

Notar que é possível não seguir o símbolo reservado **error** de nenhum símbolo, sendo usados todos os possíveis símbolos terminais que possam existir a seguir à redução da regra de erro ao símbolo alvo onde se encontra. Contudo, este processamento pode facilmente introduzir novos conflitos na gramática e incluir vários pontos de recuperação, dificultando o seu controlo.

10.5 Acções semânticas

As acções semânticas em **yacc**, tal como em **lex**, permitem efectuar processamento quando as regras são reconhecidas. Nas ausência de acções semânticas, a gramática permite apenas verificar se o ficheiro de entrada tem erros sintácticos ou está de acordo com a gramática (sintacticamente correcto).

Só depois da gramática estar correcta e sem conflitos é que se deve incluir acções semânticas, sob o risco de posteriores correcções à gramáticas obrigarem a alterações nas acções semânticas, ou pior, as acções semânticas existentes poderem ficar inconsistentes com a nova gramática.

10.5.1 Avaliação de atributos

As acções semânticas são blocos de código na linguagem **C** (o **yacc** também suporta **C++**) que são introduzidas no fim de uma regra ou, inclusivamente, no meio da regra (acções internas, ver 10.5.3). A principal limitação destas acções é o facto serem executadas quando o analisador gerado pela ferramenta **yacc** reduz a regra, pelo que a ordem de execução pode não ser a desejada.

Os valores associados aos símbolos são designados por atributos, sendo designados por atributos sintetizados quando referem valores já lidos (situados à esquerda da acção semântica em questão) da mesma regra.

Atributos sintetizados Os blocos de código tem acesso aos valores associados aos símbolos da regra a reduzir, designados por **$1** para o primeiro, **$2** para o segundo, *etc*. No momento da redução da regra estes valores são apagados, sendo substituídos pelo valor associado ao símbolo alvo, designado por **$$**. Por exemplo,

```
%union{ int i; ... }
%token<i> INT
%type<i> expr
%%
...
expr: expr '+' expr { $$ = $1 + $3; }
    | '-' expr      { $$ = -$2; }
    | INT           { $$ = $1; }
...
```

onde ... representa código omitido. Notar que na primeira regra o segundo símbolo **'+'**, designado por **$2**, não terá valor associado pelo que não pode aparecer na acção semântica. Os restantes valores deverão ser do tipo exigido pela linguagem **C** utilizada para as operações a efectuar.

Atributos globais De igual forma é possível aceder a funções e variáveis globais, tendo presente que a ordem pela qual os blocos são executados pode não ser a esperada, pois embora a leitura do ficheiro de entrada seja efectuada sequencialmente do início para o fim, a redução das regras é efectuada logo que possível. Por exemplo, o acesso a variáveis numa tabela global pode ser realizado por

```
%{
extern int setval(char *var, int val), getval(char *var);
%}
%union{ int i; char *s; ... }
%token<s> ID
%type<i> expr
%%
...
expr: ID '=' expr { $$ = setvar($1, $3); }
    | ID          { $$ = getvar($1); }
...
```

onde ... representa código omitido.

Atributos herdados à esquerda Além de atributos sintetizados, funções e variáveis globais é ainda possível aceder a atributos herdados à esquerda, ou seja, valores lidos noutras regras que não a actual, mas que ainda se encontram disponíveis na pilha auxiliar do analisador gerado ferramenta **yacc**. A utilização destes atributos é, em geral, desaconselhada pois exige um conhecimento sólido da gramática em uso e do funcionamento do analisador. Para mais, qualquer alteração à gramática pode traduzir-se em resultados caóticos, pois a pilha pode já não conter os valores esperados e a ferramenta não consegue efectuar nenhum tipo de verificações.

Considere-se o exemplo de declarações de variáveis na linguagem **C**,

```
%type<i> tipo ids
%token<s> ID
%%
decl: tipo ids ';' ;
ids : ID          { newvar($<i>0, $1); }
    | ids ',' ID  { newvar($<i>0, $3); }
    ;
```

na regra **ids : ID** não se dispõe do tipo de dados, mas este ainda está na pilha, pois só será retirado na redução da regra **decl: tipo ids ';'**. Como o símbolo **ids** está imediatamente a seguir ao símbolo **tipo**, na regra **ids : ID** a pilha irá conter no topo o símbolo **ID** (a ser reduzido a **ids**) e logo por baixo o símbolo **tipo** (e respectivo valor associado). Assim, se **$1** representa o topo da pilha, neste caso, **$0** será o valor imediatamente abaixo, ou seja o valor associado ao **tipo**. Este raciocínio só é valido se o símbolo **ids** aparecer sempre antecedido de **tipo**, pois se noutra regra podermos ter, por exemplo **extern ids**,

corre-se o risco de assumir que **$0** representa tipo, quando em certos casos (quando **ids** é invocado pela regra que contém o **extern**) não é verdade.

Além de se ter de conhecer completamente a gramática em determinado instante, para garantir que o símbolo é sempre usado no mesmo contexto, o analisador desconhece o tipo de dados associado a estes atributos. Assim, é necessário intrepor, entre o símbolo **$** e o valor posicional (neste caso **0**), a variável da **union** que deve ser utilizada, como se de uma declaração **%type** se tratasse.

Além de **$0**, existem outros atributos herdados associados a valores posicionais negativos, nomeadamente **$-1**, **$-2**, *etc.*, desde que a pilha tenha profundidade suficiente.

Atributos herdados à direita Considere-se o exemplo de declarações de variáveis na linguagem **Pascal**, onde o tipo aparece no fim da declaração:

```
decl: VAR ids ':' tipo ;
ids : ID
    | ids ',' ID
    ;
```

no momento da leitura dos identificadores que designam as variáveis (**ID**), não é possível saber o tipo da variável, para que possa ser criado espaço para ela. De facto trata-se de um atributo herdado à direita pois o valor desejado encontra-se à direita da posição actual, ou seja, ainda não foi lido. Nestes casos, a única solução reside em guardar os identificadores, por exemplo numa lista, e quando se conhecer o **tipo** percorrer a lista e criar as variáveis.

A criação deste tipo de estruturas intermédias é indesejável pois complica a escrita do compilador, a análise da linguagem, além de tornar o processamento mais lento e exigente em memória devido à criação e destruição das listas. No entanto, a solução passa por uma nova linguagem que dependa apenas de valores lidos e não de valores futuros, não tendo solução em gramáticas que analisam as linguagens da esquerda para a direita como as LA**L**R(1) utilizadas pela ferramenta **yacc**.

10.5.2 Conversão de atributos herdados à esquerda

Uma vez que a utilização de atributos herdados à esquerda no analisador gerado pela ferramenta **yacc** pode ser perigosa, aconselha-se a converter esses atributos em atributos sintetizados. Para tal existem três formas:

- escolha de outros atributos para efectuar o processamento. Tal só é possível em casos muito particulares e resulta, por exemplo de igualdades algébricas, que permitem efectuar o cálculo dos mesmos valores por outros algoritmos;
- alterações gramaticais. Consiste em alterar a gramática de tal forma que os atributos desejados apareçam associados a símbolos na regra que deles necessita (ver exemplo a seguir);
- construção de estruturas temporárias. Apesar de sobrecarregar a gramática e o seu processamento pelo analisador é uma solução muito mais segura e robusta. Esta solução deve ser preferida à utilização de atributos herdados à esquerda, sempre que nenhuma das soluções acima seja aplicável.

Considere-se o exemplo de declarações de variáveis na linguagem **C**,

```
%type<i> tipo decl
%token<s> ID
%%
decl: declvar ;
declvar: tipo ID          { newvar($1, $2); $$ = $1; }
       | declvar ',' ID   { newvar($1, $3); $$ = $1; }     {
       ;
```

a nova gramática transporta o tipo de dados no símbolo **declvar**, que é transportado para a regra seguinte. Notar que como existe recursividade na última regra, a outra é sempre reduzida primeiro.

10.5.3 Acções internas

Além de introduzir acções semânticas associadas à redução das regras, também é possível introduzir acções semânticas entre os símbolos de uma regra. Contudo, estas acções não são mais que um expediente para introduzir um novo símbolo não terminal associado a uma regra vazia. Tal facto implica que novos conflitos podem surgir, pois reduz-se a antevisão do analisador gerado.

Notar que a acção interna funciona como um símbolo da regra, podendo retornar valores associados a **$$**, podendo esses valores ser acedidos pelas acções seguintes (internas ou não) dessa regra, através da sua posição. Por exemplo,

```
instr : WHILE '('     { label($$ = lbl++);  }
           expr       { jz($$ = lbl++);     }
   ')' instrs { jmp($3); label($5); }
```

as acções intermédias que ladeiam a **expr**, e que geram as instruções de controlo de fluxo, devolvem o número da etiqueta que reinicia o ciclo e que termina o ciclo, respectivamente. Assim, a acção final pode usar esses valores para gerar o código que termina o ciclo. Notar que uma regra é sempre processada sequencialmente um item após outro, pelo que a ordem de execução dentro de cada regra é garantida, logo o código é executado da esquerda para a direita garantindo a disponibilidade de todos os atributos sintetizados.

10.6 BYacc/J: análise sintáctica em Java

Embora existam outras ferramentas para análise sintáctica em **Java**, como por exemplo o `CUP`, a ferramenta `BYacc/J` é constituída pela mesma base da ferramenta `byacc` apresentada acima. Na realidade o código gerado é **C**, caso não seja indicada a opção **-J** na linha de comando.

Resumindo, a zona entre `%{` e `%}` deve ser utilizada para a declaração de `import` e `package`, antes da geração da classe que realiza a análise sintáctica. A zona dedicada ao código do utilizador é utilizada para declarar as variáveis e métodos membro da classe, que devem inclui os métodos `int yylex()` e `void yyerror(String msg)`. A função `yylex` deve invocar o método do mesmo nome do analisador lexical, que deve ser declarado como membro e iniciado com o canal de leitura. Finalmente de referir que os valores associados à variável `yylval`, do analisador lexical, devem todos derivar da mesma classe, não devendo haver declarações de `%type` ou equivalentes. Para indicar a classe da variável `yylval` usa-se a opção **-Jsemantic=Node** da linha de comando (neste exemplo a `class Node`). Se a opção **-Jsemantic** for omitida, a ferramenta gera a classe `ParserVal`, no respectivo ficheiro `ParserVal.java`, que inclui as variáveis `int ival`, `double dval`, `String sval` e `Object obj`.

As opções **-Jclass=className**, **-Jpackage=packageName**, **-Jextends=extendName** e **-Jimplements=implementsName** permitem controlar a declaração da classe e a opção **-Jstack=SIZE** redefine a dimensão da pilha interna. Existem, ainda, opções para impedir a geração do construtor sem argumentos **-Jnoconstruct** e **-Jnorun** para não gerar o método `void run()` que invoca o analisador `yyparse()`.

11 Tabela de símbolos

A tabela de símbolos, tal como o nome indica, mantém o controlo sobre todos os símbolos utilizados pelo programa a ser gerado. Os símbolos incluem as variáveis do programa, mas também funções, etiquetas, estruturas, classes ou outros tipo de dados que a linguagem a compilar permita. A tabela de símbolos mantém um entrada para cada símbolo que esteja activo em cada parte do programa a gerar. Associado a cada símbolo deverá ser indicado, além do seu nome, a classe (por exemplo, variável ou função, ...), o tipo (inteiro, real, ...). Dependendo do símbolo pode ser necessário informação adicional, como por exemplo o deslocamento no caso de variáveis locais ou o número e tipo de argumentos no caso das funções.

As tabelas de símbolos podem conter todos os símbolos existentes em qualquer instante no programa a gerar, devendo para tal indicar quais as linhas do programa fonte em que se encontram activas. Alternativamente, pode-se manter na tabela de símbolos apenas os símbolos activos na linha em processamento. Para tal os símbolos têm de ser inseridos e retirados à medida que o processamento do programa fonte evolui, por forma a simular a entrada e saída de rotinas e blocos de código.

A tabela de símbolos designada por `tabid`, é uma solução simples e minimalista, permitindo contudo o processamento da maioria das linguagem de programação. Todos os símbolos encontram-se arrumados numa pilha, pelo que a sua utilização adapta-se melhor a linguagens com visibilidade sintáctica (*lexical scoping*) como é o caso de grande parte das linguagens compiladas, como **C**, **Java**, **C++**, *etc*. A sua utilização integra-se especialmente com a ferramenta **yacc** (ou equivalente) numa interacção dirigida pela sintaxe.

11.1 Informação associada ao símbolo

Associado a cada símbolo existem, além do nome (`char*`), apenas um tipo (`int`) um atributo (`long`). O tipo pode ser codificado como valores inteiros positivos consecutivos a partir de um (**1**) ou pode-se utilizar os próprios *tokens* criados na ferramenta **yacc** (`INT, REAL, STRING`, ...). No caso das funções, tal como para outros tipos de dados, pode-se somar uma constante decimal ou binária (utilizando um bit específico do inteiro). Por exemplo, não existindo mais `1000` tipo de dados, ou se o maior *token* não exceder esse valor, pode-se atribuir a uma função que devolve um valor inteiro o tipo `INT+1000`. Na solução binária pode-se optar pelo décimo bit ($2^{10} = 1024$), assumindo os mesmo pressupostos, passado o tipo a ser `INT+1024 = INT | 1024 = INT | (1 « 10)`. Assim, ao perguntar o tipo de um símbolo podemos concluir que se trata

de uma função se o seu valor estiver compreendido entre `1000` e `2000` (pois assumimos menos de `1000` tipos de variáveis ou *tokens*). Valores acima de `2000` podem ser utilizados para outros tipos complexos da linguagem.

O atributo de cada símbolo tem de ser codificado como um inteiro. No caso das variáveis este valor pode representar o deslocamento da variável local de uma rotina face ao registo de activação (*frame pointer*), sendo codificado como zero para variáveis globais. Notar que as variáveis globais são referidas pelo seu próprio nome, não necessitando de informação adicional, pelo que o deslocamento pode ser considerado zero. Um deslocamento zero é uma impossibilidade para uma variável local pois essa posição encontra-se ocupada pelo próprio registo de activação da rotina anterior. No caso da funções, o atributo tem de guardar mais informação. No caso de linguagens com um único tipo de dados apenas é necessário saber quantos argumentos são suportados pela rotina, pelo que o atributo pode representar precisamente esse valor. Para linguagens com múltiplos tipos de dados um inteiro não é suficiente, pois é necessário guardar o tipo de cada argumento, podendo existir diversos argumentos. No entanto, em **C/C++** desde que um inteiro (`long`) e um ponteiro tenham a mesma dimensão, um pode ser utilizado para guardar o valor do outro, e vice-versa. Assim, pode ser passado como atributo um ponteiro para uma estrutura, mais ou menos complexa, que contém a informação específica do símbolo. No entanto, não esquecer de fazer as respectivas conversões de tipos (*casts*) na invocação e retorno das rotinas que recebam ou devolvam os atributos do símbolo.

11.2 Processamento básico

A inserção dos símbolos na tabela é efectuada com recurso à rotina `int IDnew(int type, char *name, long attrib)`. Caso o símbolo ainda não exista na tabela é devolvido o valor 1 (um). Em caso de erro, a própria rotina invoca a rotina `yyerror` indicando que o símbolo já se encontra definido. Para impedir que a rotina seja invocada em caso de erro, permitindo uma forma alternativa de processamento de erros, deve ser passado como atributo (terceiro parâmetro) a macro `IDtest` (cujo valor corresponde a `-1L`). Neste caso, o atributo não é inserido em conjunto com o símbolo, podendo ser posteriormente associado ao símbolo com recurso à rotina `IDreplace`.

Para determinar se um símbolo já existe na tabela invoca-se a rotina:
`int IDfind(char *nome, long *attrib)`. O primeiro argumento é o nome do símbolo a procurar. O segundo argumento permite recolher, por referência, o valor do atributo associado ao símbolo, caso o símbolo seja encontrado. Caso se pretenda apenas saber se o símbolo existe ou não, ou saber o seu tipo, pode se passado um ponteiro nulo (valor `0`). A rotina irá devolver o tipo associado ao símbolo, caso exista, ou o valor `-1`

em caso de erro. Tal como no caso anterior, a rotina `yyerror` será invocada em caso de o símbolo não existir. Para impedir a invocação da rotina `yyerror` em caso de erro, deverá ser passada a macro `IDtest` como segundo argumento, fazendo um *cast* para o tipo do argumento:

```
if ((tipo = IDfind("x", (long*)IDtest)) == -1)
  processamento_em_caso_de_erro("x");
else
  printf("O símbolo 'x' é do tipo %d\n", tipo);
```

A rotina `int IDreplace(int type, char *name, long attrib)` permite substituir simultaneamente o tipo e atributo associado a um símbolo. Para se alterar apenas um deles deverá invocar a rotina `IDfind` para obter os valores anterior, substituindo aquele que pretende ao invocar a rotina `IDreplace`. A rotina devolve `1` (um) caso o símbolo exista e a substituição tenha sido efectuada e `-1` em caso de erro. Em caso de erro não é impressa qualquer mensagem nem invocada qualquer rotina de tratamento de erro.

11.3 Blocos aninhados

As rotinas acima são suficientes para uma linguagem que possua apenas variáveis globais. A maioria das linguagens permite definir variáveis que são apenas visíveis em certas zonas do código, facilitando a escolha de nomes pois estes necessitam ser apenas únicos no contexto local. Estas zonas correspondem frequentemente a funções ou blocos de código que podem ser colocados uns dentro dos outro, ou seja aninhados. Nestas zonas, os símbolos podem ter os mesmos nomes de outros símbolos definidos noutras zonas.

As rotinas `void IDpop(void)` e `void IDpush(void)` permite controlar o aninhamento dos diversos conjuntos de símbolos. Assim, ao iniciar o processamento da uma rotina, depois de ter inserido na tabela o seu nome (que é em geral global) e antes de inserir os seus argumentos, deverá invocar a rotina `IDpush`. Desta forma é criado um novo espaço onde serão colocados os símbolos locais da rotina. No fim do processamento da rotina, estes símbolos locais podem ser todos retirados com a invocação da rotina `IDpop`. Tal como referido no início, o `tabid` utiliza uma lista ligada simples, pelo que a rotina `IDpush` deixa uma marca na pilha, enquanto a rotina `IDpop` retira todos os símbolo da pilha até encontrar a primeira marca. Notar que se o nível de aninhamento for elevado, pelo que a rotina `IDpush` foi invocada diversas vezes, deixando outras tantas marcas na pilha, a primeira invocação da rotina `IDpop` retira apenas os

símbolos até à última marca colocada (trata-se de uma pilha, logo como um processamento LIFO - *last in first out*).

Como a rotina `IDpop` vai retiando símbolos da tabela de símbolos até encontrar a marcar, se houver uma invocação da rotina `IDpop` sem que a respectiva invocação à rotina `IDpush` tenha sido efectuada, serão removidos todos os símbolo globais, pois a marca nunca é encontrada. A rotina `int IDlevel(void)` permite saber quanto níveis de aninhamento estão activos, sendo os símbolos globais colocados no nível `0` (zero). A rotina `void IDclear(void)` permite remover todos o níveis de aninhamento existentes, mas deixando o nível global intocado. A rotina não tem qualquer efeito se não existirem aninhamentos activos.

Notar que a rotina `IDnew` insere um novo símbolo no último nível de aninhamento. Para inserir um símbolo num nível de aninhamento mais profundo, contornando o funcionamento LIFO da tabela de símbolos, a rotina `int IDinsert(int level, int type, char *name, long attrib)` pode ser invocada. Neste caso será considerado um erro caso exista um símbolo com o mesmo nome nesse nível.

A rotina de procura `IDfind` procura um símbolo, dado o seu nome, desde o nível de aninhamento activo até ao nível dos símbolos globais, passando ordenadamente por todos os níveis de aninhamento sucessivamente inferiores. Assim que o símbolo é encontrado a pesquisa termina, não sendo percorridos os níveis inferiores. Logo, um símbolo só não é encontrado se não existir em nenhum dos níveis de aninhamento, nem mesmo no nível global. A rotina `int IDsearch(char *name, long *attrib, int skip, int lev)` permite efectuar a pesquisa controlando o nível onde é inicada a pesquisa e o número de níveis a pesquisar a partir daí. O argumento `skip` indica o número de níveis a ignorar antes de iniciar a pesquisa, enquanto o argumento `lev` indica o número de níveis a pesquisar a partir daí. A rotina `IDfind` não é mais que uma versão simplificada e mais eficiente da rotina `IDsearch` onde `skip = 0` e `lev = IDlevel() + 1`.

11.4 Utilização de diversos espaços de nomes

Algumas linguagens utilizam espaços de nomes simultâneos, ou seja existe mais de um espaço de nomes não aninhados e activos. Desta forma pode-se utilizar os mesmo nomes em cada um dos espaços, sem que haja qualquer conflito. Depende da estrutura sintáctica da linguagem determinar qual dos espaços de nomes deve ser acedido em cada situação. Algumas linguagens permitem seleccionar os espaços de nomes com recurso a palavras reservadas específicas como `namespace` (**C++** e outros) ou `package` (**Java**). Em certas linguagens os nomes da variáveis e das rotinas podem ser iguais pois não existe confusão possível (por exemplo, não são permitidos ponteiros para funções).

Em algumas linguagens orientadas para objectos, como **Java**, os nomes dos atributos das classes não se confundem com os nomes dos métodos, ao contrário do **C++** em que têm de possuir nomes distintos (neste último caso sugere-se a utilização do _ para evitar conflitos de nomes). Na linguagem **C**, existem espaços de nomes distintos para estruturas, tipos de dados do utilizador (`typedef`), campos das estruturas e variáveis. Assim, embora pouco legível e muito confuso, é possível escrever:

```
typedef struct x { int x; struct x *next; } x;
int main() {
  x x;
  x.x = 2;
  x.next = 0;
  return x.x;
}
```

ou seja, na rotina `main` existe uma variável `x` do tipo `x` definido acima. Este tipo `x` não é mais que uma estrutura com o mesmo nome `x`. A estrutura possui dois campos, um dos quais também designado por `x`.

Para permitir realizar este tipo de espaços de nomes sem complicar a invocação das restantes rotinas, a tabela de símbolo `tabid` recorre apenas à rotina `void *IDroot(void *namespace)`. Esta rotina permite substituir um espaço de nomes activo, cuja referência é devolvida, por um outro, passado como argumento. Todas as restantes rotinas manipulam exclusivamente o espaço de nomes activo. Um primeiro espaço de nomes, inicialmente activo, é fornecido pelo próprio `tabid`. Os restantes deverão ser indicados pelo programador, devendo estar inicialmente iniciados a `0` (zero).

```
void *myspace = 0;
int search_in_myspace(char *name)
{
  int type;

  /* change the current namespace to 'myspace' */
  void *oldspace = IDroot(myspace);

  type = IDfind(name, 0);

  /* swap back to oldspace */
  myspace = IDroot(oldspace);
```

```
  return type;
}
```

Notar que a atribuição ao `myspace`, antes do `return`, é apenas necessária caso a tabela de símbolos tenha sido alterada, por forma a estas alterações serem salvaguardadas na variável global `myspace` para posterior utilização.

11.5 Chamadas inversas (*callbacks*)

As chamadas inversas permitem a invocação de uma rotina do programador por cada símbolo encontrado. A rotina do programador deverá ser do tipo `IDfunc`, ou seja, uma função inteira com quatro argumentos. Por exemplo,

```
int myprint(int tipo, char *nome, long atribs, long user)
{
  if (nome != 0) {
    printf("%s do tipo %d com os atributos %ld\n",
           tipo, nome, atribs);
    return 1;
  }
  else
    printf("marca de aninhamento\n");
  return 0;
}
```

Na sua forma mais simples, a invocação pode ser efectuada através da macro `int IDevery(IDfunc func, long user)`. Neste caso, para cada um dos símbolos no espaço de nomes activo será chamada a rotina `func`, começando no nível de aninhamento mais elevado e terminado nos símbolos globais. O valor devolvido resulta da soma de todos os valores devolvidos por cada uma das chamadas à rotina indicada. O argumento `user` é um parâmetro passado à rotina do utilizador, ignorado pelo `tabid`, e que pode ser utilizado para controlar o funcionamento da rotina. Por exemplo, caso se pretenda imprimir num ficheiro, pode ser passado o seu descriptor ou mesmo o um `FILE*` desde que sejam efectuados os *casts* necessários.

A rotina `int IDforall(IDfunc func, long user, int skip, int lev)` é a forma genérica da rotina `IDevery` e permite controlar quais os níveis de aninhamento a ser percorridos através dos parâmetros `skip` e `lev` (ver rotina `IDsearch` acima). A utilização de diversos espaços de nomes obriga a sucessivas invocações da rotina `IDforall` (ou `IDevery`) efectuando a trocas de espaços de nomes entre invocações (`IDroot`).

11.6 Depuração (*debug*)

Para auxiliar a depuração da tabela de símbolos, em especial se o seu comportamento não se revelar o esperado, existe uma rotina que permite imprimir no terminal todos os símbolos `void IDprint(int skip, int lev)`. Tal como com as rotinas `IDsearch` e `IDforall` os parâmetros `skip` e `lev` permitem controlar os níveis de aninhamento de início e fim a imprimir. Para imprimir toda a tabela de símbolos, do nível mais elevado para o nível global, indicar o valor `0` em ambos o parâmetros.

A rotina `IDprint` imprime um `:` (dois pontos) por cada marca de aninhamento e o padrão `nome:tipo#atrib` por cada símbolo encontrado.

Para uma depuração mais interactiva atribuir o valor `1` (um) à variável global `IDdebug`. Neste caso, as operações de inserção `IDnew` e `IDinsert`, bem como a alteração do aninhamento `IDpush` e `IDpop` produzirão mensagens no terminal no momento da sua invocação. As restantes rotinas não alteram os símbolos na tabela e não produzem qualquer mensagem.

12 Árvore sintáctica

Nos casos mais simples, a geração de código pode ser efectuada directamente nas acções do analisador sintáctico, ou seja, dirigido pela sintaxe. A construção de uma estrutura de dados temporária torna-se necessária sempre que a geração não pode ser efectuada pela ordem de processamento analisador sintáctico.

A estrutura de dados a construir necessita representar cada um dos elementos constituintes do programa a gerar. Uma vez que a informação que é necessário guardar em cada caso é distinta, são necessárias dezenas de estruturas de dados diferentes, uma para cada caso. Mesmo considerando apenas instruções, é claro que uma instrução de **return** necessita de salvaguardar uma expressão, enquanto um ciclo **while** carace de um expressão e de uma instrução, mas uma instrução **if-else** necessita de uma expressão e de duas instruções. A selecção entre as diversas estruturas poderá ser efectuada por um valor distinto de um parâmetro comum de uma **union** (por exemplo em **C** ou **C++**) ou através da própria classe numa hieraquia de classes.

Em vez de construir uma estrutura para cada caso, pode-se reduzir construção a uma estrutura básica que se repete, muito à semelhança de linguagens como **lisp** ou **scheme**. Se não tivermos considerações de espaço podemos considerar como elementos da estrutura os tipos base e uma árvore binária para as repetições. Por exemplo,

```
struct ramo {
  int atributo;
  int inteiro;
  double real;
  char *cadeia;
  struct ramo *direito;
  struct ramo *esquerdo;
};
```

Nesta solução, cada tipo de elemento é identificado por um valor do **atributo** distinto. Este valor permite, mais tarde, identificar o elemento encontrado no programa a processar. Caso se trate de um literal, apenas um dos três campos **inteiro**, **real** ou **cadeia** será preenchido. No caso de um nó interno, como os três exemplos de instruções atrás apresentados, são utilizados os ramos **direito** e **esquerdo**. Notar que a instrução **return** utiliza apenas um ramo para a expressão, enquanto uma instrução **if-else** requer duas estruturas, pois necessita de salvaguardar três valores, por exemplo **direito = expressão**, **esquerdo->direito = instrução-if** e **esquerdo->esquerdo = instrução-else**. Existem ainda casos em que não é necessário guardar qualquer valor, como por exemplo uma

instrução **break**, bastando indicar o valor do atributo, por exemplo **atributo = BREAK**. Neste último caso, os restantes cinco valores da estrutura ficam inutilizados (sem valor atribuído).

12.1 A estrutura `Node`

Os ficheiros `node.h` e `node.c` representam as declarações e a realização, respectivamente, de um tipo de dados semelhante à estrutura **ramo** acima, mas registando mais informação e de uma forma mais eficiente.

Tal como acima, os nós da árvore podem não conter informação adicional (`nodeNil`), guardar um valor inteiro (`nodeInt`), guardar um valor real representado em vírgula flutuante de precisão dupla (`nodeReal`), guardar uma cadeia de caracteres (`nodeStr`) ou guardar uma sequência de dados que não seja terminada em NULL, como por exemplo imagens (`nodeData`). Para cada uma destas folhas da árvore existe uma rotina que reserva o espaço para a estrutura e preenche os campos correspondentes.

```
Node *nilNode(int attrib);
Node *intNode(int attrib, int i);
Node *realNode(int attrib, double d);
Node *dataNode(int attrib, int size, void *data);
Node *strNode(int attrib, char *s);
```

Notar que um identificador, por exemplo `str`, pode ser guardado como `strNode(ID, "str")`, enquanto uma cadeia de caracteres, por exemplo `“hello”`, pode ser guardada como `strNode(CADEIA, "hello")`. Embora os valores a guardar sejam, em ambos os casos, cadeias de caracteres (`char*`) é possível distinguir entre o elemento através do valor do attributo (`attrib`), seja `ID` ou `CADEIA`.

No caso da estrutura (`Node`), a árovre não está limitada a dois ramos, ou seja a árvore pode não ser binária. Noa casos mais simples, como as instruções acima referidas, existem rotinas para construir um nó com um, dois ou três sub-ramos:

```
Node *uniNode(int attrib, Node *n1);
Node *binNode(int attrib, Node *n1, Node *n2);
Node *triNode(int attrib, Node *n1, Node *n2, Node *n3);
```

No caso geral, a rotina `subNode` permite criar um nó interno com qualquer número de sub-ramos, incluindo nenhum sub-ramo. Os sub-ramos são indicados na *ellipsis* (...), devendo o número de sub-ramos ser indicado no argumento (`nops`).

```
Node *subNode(int attrib, int nops, ...);
```

Caso o número de sub-ramos seja superior ao valor do `nops` os últimos argumentos são ignorados. Caso o número de sub-ramos seja inferior ao valor do `nops` os últimos sub-ramos serão preenchidos com valores arbitrários (retirados de outros valores existentes na pilha ou nos registos). Adicionalmente, todos os argumentos da *ellipsis* devem ser do tipo **Node***, tendo em conta que a utilização de outros valores, como inteiros ou cadeias de caracteres, não será considerado erro pelo compilador. Pela três razões acima, aconselha-se muito cuidado na utilização desta rotina.

Os sub-ramos podem ser acrescentados ou removidos com recurso às rotinas

```
Node *addNode(Node *base, Node *node, unsigned pos);
Node *removeNode(Node *base, unsigned pos);
```

a posição (`pos`) começa em zero, como os vectores em **C**. O valor **-1** permite acrescentar um sub-ramos após o último (*append*), enquanto o valor **0** permite acrescentar um sub-ramo antes do primeiro.

Contudo, algumas aplicações que manipulam as árvores sintácticas restringem as árvore a um máximo de dois ramos (árvore binárias), como por exemplo a ferramenta **pburg**. Neste sentido, a rotina (`seqNode`) admite os mesmos argumentos que a rotina (`subNode`), mas em vez de criar um só nó com `nops`-ramos cria uma cadeia de nós binários. Estes nós binários são todos identificados com o mesmo atributo, sendo ligados entre si pelo segundo sub-ramo, com o índice **1**:

```
Node *seqNode(int attrib, int nops, ...);
```

Além da informação sintáctica, o utilizador pode associar a cada nó um ponteiro para informação do utilizador. A rotina (`userNode`) permite substituir o ponteiro associado ao nó, devolvendo o valor anteriormente guardado.

```
void *userNode(Node *p, void *user);
```

A libertação da memória associada ao nó pode ser efectuada com acesso à rotina

```
void freeNode(Node *p);
```

Notar que a rotina liberta recursivamente a estruturas **Node**, mas não liberta cadeias de caracteres ou informação do utilizador.

A impressão de toda a árvore pode ser efectuada recursivamente com recurso à rotina (`printNode`). A impressão pode ser redirigida para o ficheiro indicado no segundo argumento. Caso o terceiro argumento seja nulo, a rotina imprime o valor numérico dos atributos. Os valores numéricos podem ser substituídos por cadeias de caracteres inscritas no vector `tab`, na posição indicada pelo valor do atributo.

```
void printNode(Node *p, FILE *fp, char *tab[]);
```

Notar que caso se utilize a ferramenta **yacc** (ou **byacc** ou **bison**) em modo de depuração (*debug*), compilando com `-DYYDEBUG`, o vector **yyname** disponibiliza os nomes dos elementos lexicais (*tokens*) no formato do terceiro argumento da rotina. Finalmente, ressalvar que esta rotina permite verificar visualmente se toda a informação necessária à geração de código foi correctamente incluída na árvore sintáctica.

Sempre que um nó é criado com recurso às rotinas acima apresentadas, o número da linha do ficheiro de entrada (`yylineno`) é salvaguardado no campo **line**, podendo ser posteriormente utilizado para uma indicação mais precisa de algum erro em que o nó esteja envolvido.

Na estrutura **Node** existe ainda um campo `info` que permite associar informação adicional a cada nó da árvore. Esta informação pode incluir o tipo de uma expressão, permitindo propagar esse tipo pelos operadores da expressão ou da declaração.

O acesso aos sub-ramos de um nó pode ser efectuado com recurso à macro `SUB(x)` ou directamente `value.sub.n[x]`, onde `x` representa o índice do sub-ramo, enquanto `value.sub.num` indica o número de sub-ramos no nó.

A estrutura **Node** possui ainda dois campos `state`, utilizado pelo selector de instruções **pburg**, e `place`, utilizado pela selecção de registos. Caso não estejam a ser utilizados pelas respectivas ferramentas, estes campo podem ser utilizados livremente.

O ficheiro **node.h** contém ainda um conjunto de definições que permitem a integração directa com a ferramenta **pburg**, incluindo **STATE_TYPE**, **NODEPTR_TYPE**, **OP_LABEL(p)**, **LEFT_CHILD(p)**, **RIGHT_CHILD(p)**, **STATE_LABEL(p)** e **PLACE(p)**.

13 Selecção de instruções (*pburg*)

A geração de código final parte do código intermédio e procura aproveitar as características do processador final. Os primeiros passos consistem em escolher as instruções do processador que mais se adequam às sequências de instruções de código intermédio logo seguido da atribuição de registos do processador a cada um dos registos virtuais do código intermédio.

A selecção de instruções é um passo complexo do processo de compilação pois é necessário conhecer as instruções do processador. O resultado é código *assembly* específico da arquitectura desejada e é difícil de verificar e corrigir. Por estas razões, a utilização de ferramentas que automatizem o processo de selecção de instruções é crítico e simplifica significativamente a geração de código de um compilador optimizante.

13.1 Gramáticas de instruções

A utilização de gramáticas, que descrevem as instruções do processador a utilizar, permitem sistematizar o processo de selecção de instruções, melhorando a qualidade do código. O processo fica simplificado bem como a sua posterior manutenção, facilitando a alteração da gramática para passar a utilizar instruções de outro processador. A selecção de instruções pode ser crítica em processadores com muitas instruções, pois permite realizar a mesma operação recorrendo a diferentes combinações de instruções, como é o caso dos processadores CISC. Em processadores com poucas instruções, como os processadores RISC ou *stack mahines* as escolhas são poucas, pelo que a selecção de instruções não é tão importante. No entanto, no caso dos processadores RISC, o espaço disponível para constantes embebidas no código (valores imediatos ou *immediates*) é limitado, devido ao facto de as instruções terem todas a mesma dimensão, por exemplo $32 - bits$ ou $16 - bits$. Desta forma, as constantes imediatas e os endereços de memória podem ser carregados através de diversas instruções, o que também justifica a utilização de um selector de instruções.

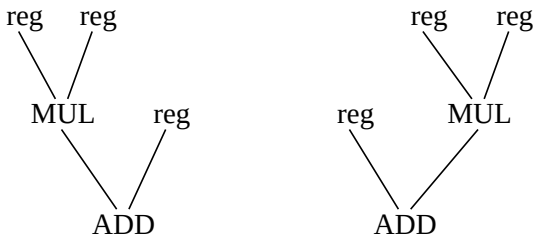
Os algoritmos de selecção de instruções baseiam-se em descrições gramaticais das instruções do processador alvo. Cada regra descreve as possíveis sequências de instruções intermédias necessárias para produzir uma instrução máquina do processador alvo. O algoritmo deve escolher o conjunto de instruções máquina com custo mínimo. O custo de uma instrução é, normalmente, descrito em termos de eficiência computacional, mas outros critérios como a dimensão ou o tráfego com a memória podem ser determinantes em certas arquitecturas ou aplicações específicas. Tal como foi abordado nas descrições das linguagens, também as instruções de um processador constituem uma linguagem que pode ser descrita por uma gramática regular ou livre de contexto.

13.2 Selecção de instruções por árvore sintáctica

O emparelhamento por árvore sintáctica executa a selecção das instruções através da comparação de padrões que representam as instruções do processador com a árvore sintáctica produzida pelas fases de análise do compilador. Sempre que o padrão da instrução coincide com uma parte da árvore sintáctica, então a instrução pode ser utilizada para transformar essa sub-árvore. Notar que anteriormente limitámos a selecção à comparação de um único atributo com recurso ao padrão `visitor` ou à construção `switch-case`. Para poder seleccionar instruções mais complexas torna-se necessário identificar padrões mais complexos.

O primeiro passo consiste em representar a instrução como um padrão que se assemelha a uma pequena árvore sintáctica em que cada nó tem o mesmo atributo do nó utilizado na construção da árvore sintáctica do programa em análise. Por exemplo, o processador `arm` oferece uma soma após uma multiplicação sem custo adicional (`muladd`), desde que um dos argumentos da soma seja o resultado da multiplicação. Como qualquer dos argumentos tem de residir num registo, podemos construir dois padrões para esta instrução, dependendo da posição do outro valor a somar, uma vez que a soma é comutativa:

```
reg: ADD(MUL(reg,reg),reg)
reg: ADD(reg,MUL(reg,reg))
```



O resultado não é mais que uma gramática onde cada uma das linhas não é mais que um padrão, com o símbolo não terminal `reg` e os símbolos terminais `ADD` e `MUL`. Os símbolos terminais correspondem aos atributos da árvore sintáctica gerada para o programa em análise, enquanto os símbolos não terminais representam classes de armazenamento.

Caso o valor a calcular não esteja num registo é necessário carregá-lo, quer seja uma constante inteira ou uma variável, pelo que podemos usar duas árvores simples, sem sub-ramos, para o fazer:

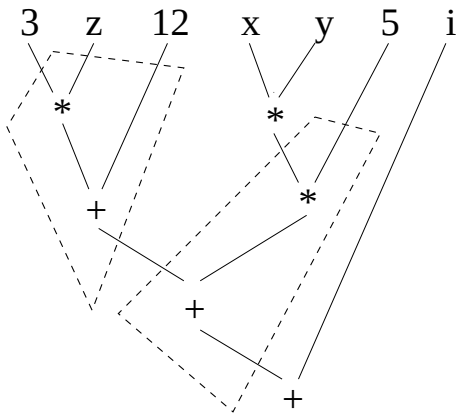
```
reg: INT
reg: VAR
```

É ainda necessário considerar que nem sempre será possível aproveitar a possibilidade de soma após multiplicação, pelo que temos de fornecer os casos mais simples de soma e de multiplicação separadamente. Notar que mesmo que a única instrução para realizar somas ou multiplicações fosse a instrução `muladd` ainda teria de indicar que o processador é capaz de as realizar separadamente, caso apareçam separadamente na árvore sintáctica.

```
reg: ADD(reg,reg)
reg: MUL(reg,reg)
```

Para já consideremos que todas as instruções do processador têm um custo unitário. Na realidade, a soma quando executada isoladamente é mais rápida, mas a multiplicação efectua uma soma com o valor zero.

Agora que dispomos de um processador capaz de efectuar somas, multiplicações e carregamento de valores constantes e conteúdos de variáveis, podemos processar uma expressão algébrica simples. Por exemplo, a expressão `3 * z + 12 + x * y * 5 + i`, a que corresponde a árvore:



sendo gerado o código

```
load r0, 3              ; r0 = 3
load r1, z              ; r1 = z
load r2, 12             ; r2 = 12
muladd r2, r0, r1, r2   ; r2 = r0 * r1 + r2
load r0, x              ; r0 = x
load r1, y              ; r1 = y
mul r0, r0, r1          ; r0 = r0 * r1
load r1, 5              ; r1 = 5
muladd r2, r0, r1, r2   ; r2 = r0 * r1 + r2
```

```
load r0, i              ; r0 = i
add r2, r2, r0          ; r2 = r2 * r0
```

onde a instrução `mul` é uma representação simplificada da instrução `muladd` quando a soma é efectuada com a constante zero.

13.2.1 Selecção de instruções por programação dinâmica

O primeiro passo para efectuar a selecção de instruções por árvore sintáctica consiste em determinar quais as sub-árvores da árvore sintáctica coincidem com cada instrução. Para tal consideremos duas restrições:

os nós das árvores têm apenas dois ramos. Embora a generalização a maior número de ramos seja possível, não só complica a explicação desnecessariamente como raramente é necessária.

a árvore da instrução contém um só nó. Para que tal se verifique é necessário decompor árvores complexas em sequências de árvores simples utilizando símbolos não terminais intermédios. Nomeadamente, as regras que descrevem a instrução `muladd` podem ser decompostas em

```
reg: ADD(tmp1,reg)
tmp1: MUL(reg,reg)
reg: ADD(reg,tmp2)
tmp2: MUL(reg,reg)
```

Notar que os símbolos não terminais são distintos, pois as regras que os utilizam podem ter aplicações distintas. Além disso, os custos das regras introduzidas podem ser considerados nulos, sendo a totalidade do custo imputada à regra final. Estas regras são geradas internamente pela ferramenta apenas para efeitos de emparelhamento.

O objectivo do emparelhamento consiste em etiquetar cada nó da árvore sintáctica com os padrões das instruções utilizadas pelo processador. A etiquetagem indica todos os padrões reconhecidos e pode ser efectuada com base no número da regra, uma vez que esta identifica univocamente o padrão. Assim, associado a cada nó da árvore sintáctica é necessário associar, temporariamente, as regras emparelhadas. O algoritmo percorre a árvore sintáctica em profundidade para garantir que todos os sub-ramos já foram etiquetados. Em cada nó adiciona todas as regras cujos não terminais do padrão da regra coincide com as etiquetas dos sub-ramos.

```
Tile(n)
  Label(n) <- 0
  Tile(child[0]) ... Tile(child[k])
  for each rule that matches n
    if child[r] in Label(child[n]) and ...
      Label(n) <- Label(n) U {r}
```

De acordo com o algoritmo, em cada nó necessitam ser verificados todos os padrões, ou seja as respectivas regras. Este processo é bastante dispendioso, pelo que se pode calcular previamente quais as regras que podem coincidir com o padrão e guardar o resultado numa tabela indexada pelo atributo. Elimina-se, desta forma, os padrões que nunca poderiam emparelhar com o nó em questão. Como os padrões são no máximo binários, ou seja com dois sub-ramos, existem no máximo 2^{regras} emparelhamentos possíveis. Uma gramática com 200 atributos e 1000 emparelhamentos produz uma tabela com 200 milhões de valores. No entanto, como a tabela é muito esparsa o espaço pode ser muito reduzido. Alternativamente, em vez de aceder à tabela com base no atributo, pode-se codificar a tabela como uma construção `switch-case` indexada pelo atributo, tal como nos capítulos anteriores. Em cada um dos `case` é apenas necessário considerar uma pequena lista de possíveis emparelhamentos. Por exemplo, na gramática acima, o `case ADD:` tem de considerar os padrões `reg: ADD(reg,reg)`, `reg: ADD(MUL(reg,reg),reg)` e `reg: ADD(reg,MUL(reg,reg))`.

Para cada símbolo não terminal regista-se apenas a regra de menor custo, ou seja, a regra que consegue colocar os dados no não terminal com um número de ciclos máquina inferior. A determinação do custo considera o custo da própria instrução a produzir mais os custos das instruções necessárias para colocar os argumentos nos nós não terminais utilizados. Assim, na regra `reg: ADD(reg,reg)` é necessário considerar os custos de colocar ambos os argumentos nos registos, enquanto nas regras `reg: ADD(MUL(reg,reg),reg)` e `reg: ADD(reg,MUL(reg,reg))` é necessário somar os custos de colocar os três argumentos nos três registos utilizados. O código necessário pode ser codificado como:

```
case ADD:
    label(left);
    label(right);
    if (left.attrib == MUL){        /* reg: ADD(MUL(reg,reg),reg) */
        cost = left.left.cost[regNT] + left.right.cost[regNT] +
                        right.cost[regNT] + COST1;
        if (cost < p.cost[regNT]) {
            p.cost[regNT] = cost;
            p.rule.reg = RULE1;
    }   }
```

```
if (right.attrib == MUL){        /* reg: ADD(reg,MUL(reg,reg)) */
    cost = left.cost[regNT] + right.left.cost[regNT] +
                    right.right.cost[regNT] + COST2;
    if (cost < p.cost[regNT]) {
        p.cost[regNT] = cost;
        p.rule.reg = RUEL2;
}   }
{        /* reg: ADD(reg,reg) */
    cost = left.cost[regNT] + right.cost[regNT] + COST6;
    if (cost < p.cost[regNT]) {
        p.cost[regNT] = cost;
        p.rule.reg = RULE6;
}   }
break;
```

onde as constantes `COST1`, `COST2` e `COST6` representam os custos das respectivas instruções e `RULE1`, `RULE2` e `RULE6` identificam as regras seleccionadas, ou seja os padrões emparelhados.

Uma vez os valores podem não residir apenas nos registos é necessário definir outros símbolos não terminais, dependendo das capacidades do processador. Por exemplo, se o processador possuir uma unidade de vírgula flutuante com registos independentes, o símbolo não terminal `freg` pode descrever um desses registos. Assim, as regras podem incluir:

```
freg: REAL
freg: ADD(freg,freg)
freg: reg
```

onde a primeira regra permite carregar uma constante, a partir de memória ou imediata consoante as capacidades do processador. A última regra permite utilizar valores inteiros em expressões em vírgula flutuante.

As instruções, por exemplo, não produzem resultados em nenhum registo, pelo que tais instruções podem ser indicadas como:

```
stat: STORE(VAR,reg)
stat: JMP(LABEL,reg)
```

Outras classes de símbolos não terminais podem incluir posições de memória genéricas, posições na pilha de chamada de rotina ou constantes de valores limitados. Estas constantes de valores limitados permitem, especialmente em processadores RISC, seleccionar instruções específicas. Por exemplo, caso o valor da constante possa ser codificado em 12, 11 ou mesmo 8 bits, dependendo das instruções do processador alvo.

O processo completo de selecção de instruções por programação dinâmica é composto de dois passos. O primeiro passo, que acabámos de ver, consiste em percorrer a árvore sintáctica das folhas para a raíz (*bottom-up*), determinando em cada nó qual a regra que produz, com um custo mais reduzido, cada um dos possíveis símbolos não terminais. O segundo passo consiste na geração das instruções e é efectuado da raíz para as folhas, mas tendo em conta os custos óptimos em cada nó.

13.2.2 Emparelhamento com custos variáveis

Desde que a árvore sintáctica gerada já tenha eliminado qualquer ambiguidade, não existe necessidade de utilizar custos variáveis nas instruções. A utilização de custos variáveis permite aceitar ou rejeitar uma instrução durante o processo de selecção. Esta capacidade é particularmente útil se a árvore for genérica, ou seja, independente do processador a utilizar. No entanto, se a árvore é gerada especificamente para o processador alvo, então significa que parte do processo de selecção já foi executado ao gerar a árvore adaptada.

Por exemplo, considere-se o carregamento de uma variável:

```
reg: VAR 2
freg: VAR 5
```

onde o valor inteiro no fim da regra indica o custo da instrução, ou instruções, a serem geradas. Além disso, se uma regra tiver um custo inferior terá, à partida, preferência sobre a outra regra, embora a escolha só seja efectuada após a contabilização geral de todas as regras necessárias. Na realidade, o que se pretende é que variáveis de tipo inteiro sejam carregadas em registos inteiros (`reg`) e variáveis do tipo real sejam carregadas em registos de vírgula flutuante (`freg`). Uma solução passa por produzir atributos dependentes do tipo, ou seja, `IVAR` para inteiros, e `RVAR` para reais.

```
reg: IVAR 2
freg: RVAR 5
freg: IVAR 3
```

onde a última instrução indica a possibilidade de o processador carregar valores inteiros directamente num registo de vírgula flutuante.

Utilizando custos variáveis pode-se associar uma função a cada regra, que será chamada para determinar o custo da instrução:

```
reg: VAR isInteger
freg: VAR floatCost
```

Assim, a rotina `isInteger` devolve o custo de carregamento de um valor inteiro num registo inteiro, caso a variável seja do tipo inteiro, e um custo muito elevado para desencorajar a sua selecção, para outros tipos de variáveis. Da mesma forma, caso seja possível ao processador, a rotina `floatCost` devolve os custos de carregamento de variáveis inteiras ou de variáveis reais, dependendo do caso, e devolve um custo muito elevado caso a variável seja de um tipo inválido para um registo de vírgula flutuante.

Notar que esta abordagem implica que a tabela de símbolos já tem conhecimento do tipo associado a cada variável no momento da etiquetagem da árvore sintáctica.

Uma outra utilização do custo variável permite utilizar a capacidade de incremento na memória de alguns processadores:

```
stmt: STORE(VAR,ADD(VAR,reg)) isSame
stmt: STORE(VAR,ADD(VAR,INT)) isSame
stmt: STORE(VAR,ADD(reg,VAR)) isSame
stmt: STORE(VAR,ADD(INT,VAR)) isSame
```

permitindo que instruções como `x = x + 1` ou `x = y + x` gerem respectivamente `x++` e `x += y`. Para tal a rotina `isSame` deverá verificar se ambos os argumentos `VAR` referem a mesma variável, ou seja, se têm o mesmo nome. Caso contrário, devolve um custo muito elevado, por forma a que as instruções normais sejam seleccionadas em sua substituição. Notar que o mesmo resultado pode ser obtido, com a árvore adaptada, fazendo previamente a substituição da sub-árvore em questão por nós `INCR(VAR,reg)` e `INCR(VAR,INT)`, mas acaba por se estar a efectuar manualmente a selecção das instruções, sem se tirar partido da ferramenta.

13.2.3 Manipulação de tipos no emparelhamento

O código intermédio a incluir na árvore sintáctica deve ser tão próximo quanto possível do `assembly`. Para tal, a análise semântica do código a gerar já foi efectuada e já se conhecem os tipos e localizações de todas as variáveis. A análise semântica efectua as verificações que o analisador sintáctico não consegiu efectuar, embora de um ponto de vista linguístico estas verificações são igualmente sintácticas. Estas verificações podem exigir percorrer a árvore sintáctica gerada uma segunda vez para efectuar, por exemplo, verificações dos tipos das variáveis e respectivas operações. Para evitar percorrer a árvore sintáctica uma segunda vez pode-se efectuar um conjunto significativo de verificações quer na construção da árvore, quer no emparelhamento da árvore para a geração

das instruções. As verificações podem ser efectudas na construção da árvore, desde que a sequência pela qual são efectuadas seja compatível com a ordem pela qual o analisador sintáctico vai identificando as regras. Por outro lado, ao efectuar as verificações no emparelhamento da árvore a sequência corresponde à ordem pela qual as instruções são geradas. O emparelhamento óptimo, tal como efectuado pelo **pburg**, exige que os custos sejam todos conhecidos antes de iniciado o processo de geração. Do ponto de vista da manipulação de tipos diversas abordagens são possíveis.

Numa linguagem em que todas as declarações precedem as instruções é necessário que as declarações estejam concluídas antes de iniciado o processo de selecção das instruções. Para tal não basta que as declarações, o nó da esquerda, sejam avaliadas antes das instruções, o nó da direita. Na realidade o processo de etiquetagem é efectuado sobre a totalidade da árvore antes do processo de geração, pelo que todos os custos são determinados e as instruções seleccionadas antes ser invocada qualquer instrução de geração de código.

```
prog: decls instrs { yyselect(binNode(PROG, $1, $2)); }
```

Desta forma, não é possível fazer com que as declarações possam influenciar a forma como as instruções são seleccionadas. O processo pode ser dividido em duas fases distintas.

```
prog: decls instrs { yyselect($1); yyselect($2); }
```

Esta abordagem exige que, quer declarações, quer instruções, derivem o símbolo não terminal inicial (*start symbol*), para que ambos os processos de selecção sejam bem sucedidos.

```
%start prog
%%
prog: decls
prog: instrs
```

Se uma linguagem, como por exemplo a linguagem **C**, permitir declarar variáveis ao longo do código, mas sempre no início de um bloco, torna-se necessário extender o mesmo conceito a blocos.

```
instr: '{' decls instrs '}' { $$ = binNode(BLOCK, $2, $3); }
```

Assim, embora as instruções antes e depois do bloco sejam seleccionadas antes deste, elas apenas referem variáveis com maior visibilidade que este. Por outro lado, quando as instruções do bloco são seleccionadas é necessário garantir que todas as variáveis neste bloco ou em blocos mais abragentes já se encontram analisadas. Logo, mesmo sendo um nó binário, é necessário indicá-lo como uma folha da árvore para que os seus ramos não sejam avaliados conjuntamente com os do bloco anterior. O processamento do bloco, tal como no caso anterior, efectua a selecção das respectivas declarações e só depois de concluído avalia as instruções.

```
expr:   BLOCK     1 { yyselect(LEFT_CHILD(p));
                      yyselect(RIGHT_CHILD(p)); }
```

Quando uma linguagem permite a declaração de variáveis em qualquer ponto do código, como por exemplo **C++**, já não é possível controlar o processamento das declarações de forma independente quando se efectua a selecção das instruções. Assim, todo o processamento que influencie a geração de instruções tem de ser efectuado anteriormente. Se a linguagem permitir a utilização de variáveis apenas após a sua declaração, a manipulação de tipo pode ser efectuada durante o processo de análise sintáctica. Para tal, quer a declaração de variáveis globais (`decl: type ID`) como a declaração de variáveis locais (`instr: type ID`) introduzem a declaração simultaneamente na árvore e na tabela de símbolos. A utilização da variável (`expr: ID`) introduz na árvore dois tipos de nós distintos, dependendo da declaração.

```
prog: decls instrs { yyselect(binNode(PROG, $1, $2)); }
decl: type ID   { $$ = binNode(DECL, $1, $2); IDnew($1, $2, 0); }
instr: type ID  { $$ = nilNode(AUTO); IDnew($1, $2, pos -= 4); }
expr: ID        { int pos, type = IDfind($1, &pos);
                  if (type < 0) $$ = nilNode(END);
                  else $$ = pos == 0 ? strNode(type, $1)
                                     : intNode(type, pos);
                }
```

Notar que no caso de variáveis locais não existe código a ser gerado, pelo que é introduzido na árvore um nó `AUTO` que não gera código, pois apenas é necessário contabilizar a dimensão da variável para incluir na instrução `ENTER` (gerada no início da função). Da mesma forma, caso a variável não esteja definida, é incluído um nó `END` na árvore, sendo a rotina de erro `yyerror` invocada pela rotina `IDfind`.

13.2.4 Construção da árvore sintáctica para emparelhamento

A construção da árvore necessária ao processo de selecção restringe os nós da árvore a um máximo de dois ramos. Além disso, para simplificar o processo de selecção, é desejável gerar a árvore já com instruções mais próximas do `assembly`. Assim, os ciclos e condições podem ser decompostos em instruções de salto e etiquetas, como por exemplo:

```
stmt: WHILE '(' expr ')' stmt
        { int lbl1 = ++lbl, lbl2 = ++lbl;
          $$ = binNode(';', strNode(JMP, mklbl(lbl1)),
                binNode(';', strNode(LABEL, mklbl(lbl2)),
                 binNode(';', $5, /* instr */
                  binNode(';', strNode(LABEL, mklbl(lbl1)),
                   binNode(JNZ, $3 /* cond */,
                    strNode(ETIQ, mklbl(lbl2)))))));
        }
```

O atributo **';'** é utilizado para ligar as instruções, sendo um operador binário de custo nulo que não deve gerar qualquer código. Alternativamente, a sequência de nós binários de atributo **';'** pode ser gerada com:

```
stmt: WHILE '(' expr ')' stmt
        { int lbl1 = ++lbl, lbl2 = ++lbl;
          $$ = seqNode(';', 5,
                       strNode(JMP, mklbl(lbl1)),
                       strNode(LABEL, mklbl(lbl2)),
                       $5, /* instr */
                       strNode(LABEL, mklbl(lbl1)),
                       binNode(JNZ, $3, /* cond */
                        strNode(ETIQ, mklbl(lbl2))));
        }
```

Neste caso, a sequência é decomposta em pares binários com o mesmo atributo, como no exemplo antecendente.

Notar que as árvores podem continuar a conter mais de dois ramos, desde que a informação que contêm não seja utilizada no processo de selecção, mas podendo ser utilizada na geração do código. Na realidade, o processo de selecção vai necessitar em cada nó da árvore de um campo, do tipo ponteiro, para associar a cada nó a informação das regras seleccionadas e respectivos custos acima descritos. Além disso, pode ser útil incluir ainda um campo adicional para auxílio à geração do código, a efectuar após a selecção, que poderá indicar qual o registo, deslocamento na pilha ou posição de memória ocupada pelo dados referidos no nó.

A mesma instrução pode ser gerada utilizando uma árvore mais simples:

```
stmt: WHILE '(' expr ')' stmt
        { $$ = binNode(WHILE, binNode(DO, nilNode(START), $5), $3); }
```

Neste caso, são definidos dois *tokens* internos (i.e. sem correspondência com a análise lexical ou a gramática da linguagem) `DO` e `START`. Como a árvore é avaliada da esquerda para a direita, o nó `START` é necessário pois o ciclo requer a geração de instruções de controlo antes da avaliação de qualquer um dos seus argumentos. A avaliação dos nós `DO` e `WHILE` permite gerar instruções de controlo entre os dois argumentos e após estes serem avaliados.

De uma forma simplificada, as instruções de controlo a introduzir pelo ciclo são:

```
stmt:   WHILE(do, expr) 1 { jnz again: }
do:     DO(begin, stmt) 1 { inicio: }
begin:  START           1 { jmp inicio; again: }
```

Na realidade, torna-se necessário salvaguardar as etiquetas de salto no nó da árvore, para que estejam disponíveis nos nós subsequentes. Em vez de guardar ambas as etiquetas no nó `START`, basta gerar duas etiquetas consecutivas e guardar o número (ou nome) da primeira (ou segunda), podendo a outra ser deduzida. Para tal, pode-se utilizar o campo `user` disponível em cada nó da árvore (ver `node.h`).

```
stmt: WHILE(do, expr) 1 { printf("jnz %s:\n",
                          mklbl(1+((int)LEFT_CHILD(LEFT_CHILD(p))->user)); }
do:   DO(begin, stmt) 1 { printf("%s:\n", mklbl((int)LEFT_CHILD(p)->user); }
begin: START          1 { int lbl1 = ++lbl, lbl2 = ++lbl;
                          p->user = (void*)lbl1;
                          printf("jmp %s\n%s:\n",
                                 mklbl(lbl1), mklbl(lbl2)); }
```

Notar que como as instruções de salto condicional, tal como as de salto incondicional, são geradas no emparelhamento final, e não na árvore. Assim, não é possível efectuar o seu emparelhamento do salto condicional com a comparação que a antecede, pelo que esta optimização não é efectuada.

13.2.5 Recursão e ordem de selecção

Quando existe recursão na gramática pode-se optar por realizar a recursão à direita ou à esquerda, excepto em analisadores descendentes que não suportam recursão à esquerda. Por outro lado, o selector de instruções avalia sempre o nó da esquerda antes do nó da direita. Assim, a forma como a árvore é construída determina se os nós serão seleccionados pela ordem que surjem no ficheiro fonte, caso se processem sequências de instruções, ou pela ordem inversa, caso se tratem de argumentos de uma função com convenção de chamada *à la* **C**.

Utilizando recursão à esquerda podemos ter

```
list: elem            { $$ = $1; }
    | list SEP elem   { $$ = binNode(SEP, $1, $3); }
```

ou

```
list: elem            { $$ = binNode(SEP, nilNode(START), $1); }
    | list SEP elem   { $$ = binNode(SEP, $1, $3); }
```

em qualquer dos casos o nó `SEP` contém a recursão do lado esquerdo pelo que os elementos `elem` serão seleccionados pela ordem que aparecem no ficheiro de entrada. Neste caso, as respectivas gramáticas de selecção deverão ser

```
target: elem
target: SEP(target,elem)
```

ou

```
target: SEP(START,elem)
target: SEP(target,elem)
```

respectivamente. Por outro lado, podemos ter

```
list: elem            { $$ = $1; }
    | list SEP elem   { $$ = binNode(SEP, $3, $1); }
```

ou

```
list: elem            { $$ = binNode(SEP, $1, nilNode(END)); }
    | list SEP elem   { $$ = binNode(SEP, $3, $1); }
```

onde a recursão ainda é à esquerda mas os ramos da árvore encontram-se trocados com os do exemplo anterior, ficando a recursão no ramo direito da árvore. Neste caso, as respectivas gramáticas de selecção deverão ser

```
target: elem
target: SEP(elem,target)
```

ou

```
target: SEP(elem,END)
target: SEP(elem,target)
```

respectivamente. Caso a recursão da gramática estivesse do lado direito, os papéis invertiam-se uma vez que a selecção da árvore com os nós colocados pela mesma ordem da gramática iria ser efectuada pela ordem inversa do ficheiro fonte.

Notar que existe sempre uma correspondência directa entre cada instrução de construção da árvore e o respectivo padrão de selecção. No entanto, esse padrão pode aparecer em diversas regras de selecção, sózinho ou em conjunto com outros padrões.

13.2.6 A ferramenta pbrug e a linguagem Compact

Existem diversas ferramentas que realizam o mesmo algoritmo, com pequenas variações no formato de entrada. O ficheiro segue uma estrutura semelhante à utilizada pelas ferramentas **lex** e **yacc**, estando dividida em três secções separadas por uma linha contendo apenas a sequência `%%`. A primeira secção é reservada para declarações podendo o código **C/C++** ser incluído da mesma forma. Também a última secção é escrita em **C/C++**, sendo copiada para o final do ficheiro de saída. Nas duas primeiras zonas são considerados comentários as linhas iniciadas pela sequência `%!`.

A secção das declarações permite definir os símbolos terminais da gramática e o símbolo inicial da mesma. O símbolo inicial da gramática é o primeiro símbolo não terminal da gramática, excepto se existir a declaração **%start** seguido do nome do símbolo. Os símbolos terminais são definidos com **%term** seguido dos símbolos terminais da gramática a definir, sob a forma **id=valor**, onde **id** representa o nome do símbolo e **valor** o valor inteiro com que o atributo do nó vem etiquetado, por exemplo, **%term ADD=43**. A ferramenta permite definir os símbolos terminais utilizando caracteres, por exemplo **%term ADD='+'** ou mesmo incluir todas as definições de um ficheiro como o `y.tab.h` com o formato gerado pela ferramenta **yacc**, utilizando **%include "y.tab.h"**. Notar que, ao contrário da ferramenta **yacc**, não é possível utilizar caracteres na descrição dos padrões, ou seja, não se pode utilizar `'+'` mas sim o símbolo terminal `ADD`. A declaração com `%include` permite carregar todas as etiquetas definidas na análise gramatical, por exemplo **%include "y.tab.h"**, assumindo que estas foram geradas por ferramentas como o **yacc**, **byacc** ou **bison**. A declaração permite processar as primeiras linhas do ficheiro indicado, declarando, como se tivessem sido declarados com `%term`, as linhas iniciadas

por `#define`, terminando na primeira linha que não seja um `#define`. Devido às diferenças entre os formatos gerados de análise gramatical, a ferramenta **pburg** permite processar todos `#define` existentes no ficheiro, sem parar na primeira linha que não seja um `#define`, desde que seja indicada a opção **-A** (All) na invocação do **pburg**.

A secção central é constituída pela gramática. O símbolo não terminal da primeira regra é utilizado como símbolo inicial, caso não tenha sido definido um símbolo inicial nas declarações. Todas as sequências de emparelhamento têm de o produzir o símbolo inicial, caso contrário é dado um erro pois não existem padrões que permitam gerar a sequência representada na árvore sintáctica. O formato de cada uma das regras é constituído por um símbolo não terminal seguido do padrão da árvore a emparelhar; depois indica-se o custo a associar à regra e, finalmente, o código a executar quando a regra é seleccionada. O custo pode ser um valor inteiro não negativo ou o identificador de uma função que recebe a raíz da sub-árvore a emparelhar e devolve o valor do custo da instrução. Se o custo for omitido é considerado que a regra tem um custo nulo (0). Notar que a função custo pode ser invocada diversas vezes no processo de selecção (*labeling*), mesmo que a respectiva regra nunca chegue a ser seleccionada. O código associado a cada regra está delimitado por chavetas e só é chamado após todo o processo de selecção ter sido concluído (*reduction*), podendo ser omitido.

```
stat:  mem                 {    }
mem:   store(reg)     19   {    }
mem:   store(cte)     20   {    }
reg:   load(mem)      18   {    }
reg:   load(cte)      4    {    }
reg:   add(reg,reg)   3    {    }
reg:   add(reg,cte)   4    {    }
reg:   add(reg,mem)   19   {    }
mem:   add(mem,cte)   31   {    }
mem:   add(mem,reg)   30   {    }
reg:   uminus(reg)    3    {    }
mem:   uminus(mem)    30   {    }
```

Para que a ferramenta manipule a árvore sintáctica é necessário incluir na zona de declarações um conjunto de macros que indicam os campos específicos a aceder. Utilizando a estrutura de árvore definida em **node.h**, esta já define o tipo de dados (`NODEPTR_TYPE`), o campo que define o operador (`OP_LABEL`), o campo que acede aos ramos da esquerda (`LEFT_CHILD`) e da direita (`RIGHT_CHILD`). Para que a ferramenta possa guardar a estrutura com os custos e as regras seleccionadas em cada nó utiliza-se o campo `state` (`STATE_LABEL`) e definimos o seu tipo (`STATE_TYPE`).

```
#define NODEPTR_TYPE Node*
#define OP_LABEL(p) ((p)->attrib)
#define LEFT_CHILD(p)  ((p)->type == nodeOpr ? (p)->value.sub.n[0] : 0)
#define RIGHT_CHILD(p) ((p)->type == nodeOpr ? (p)->value.sub.n[1] : 0)

#define STATE_TYPE void*
#define STATE_LABEL(p) ((p)->state)
```

Caso seja utilizada outro tipo de estrutura de dados para representar a árvore sintáctica, as definições acima terão de ser adaptadas aos campos a disponibilizar.

Notar que as ferramentas não permitem que um mesmo operador seja utilizado com número de argumentos diferente, pelo que se o operador **';'**, acima referido, for usado como um operador binário que não gera código. Operadores distintos terão de ser utilizados com a mesma função para operadores unários ou operadores sem argumentos.

O ficheiro gerado, designado por **yyselect.c** se não for indicado outro nome na invocação da ferramenta, inclui a rotina `yyselect()` que efectua a selecção das intruções dada a árvore sintáctica. A rotina `yyselect()` devolve o valor **0** (zero) caso a etiquetagem e redução tenham sido bem sucedidas ou **1** (um) caso não tenha sido possível seleccionar todas as instruções da árvore fornecida.

13.2.7 Suporte para Java

A opção **-J** na linha de comando activa a geração de código **Java**, caso a opção não seja indicada o código gerado é **C**, A opção **-J** gera a classe **Selector** no ficheiro **Selector.java**. A opção **-p** pode ser utilizada para alterar o nome da classe a gerar. O nome do ficheiro pode ser especificado como o último argumento, tal como em **C/C++**.

A árvore sintáctica assume que todos os nós são instâncias da classe **Tree**. O nome desta class pode ser alterado na própria opção **-J** seguindo-a do nome da nova classe, por exemplo **-JNewTree**. Os diversos nós da árvore sintáctica podem ser instâncias de classes derivadas. No entanto, todos os nós devem permitir obter a respectiva etiqueta através da invocação do método **int label()**. Da mesma forma, todos os nós devem oferecer os métodos **void state(Object)** e **Object state()** para permitir associar a informação de etiquetagem a cada nó. Os nós que não sejam folhas da árvore devem possuir os métodos **left()** e **right()** se tiverem aridade binária, ou apenas o método **left()** caso a aridade seja unária. Notar que a aridade é especificada no padrão e representa o número de símbolos não terminais associados ao padrão de uma determinada etiqueta.

A zona entre `%{` e `%}` deve ser utilizada para a declaração de `import` e `package`, antes da geração da classe que realiza a selecção das instruções. A zona dedicada ao código do utilizador é utilizada para declarar as variáveis e métodos membro da classe. A

declaração `%include` assume que o ficheiro a incluir foi gerado pela ferramenta **byaccj**, considerando as declarações iniciadas por `public final static short` .

A classe gerada inclui os métodos **panic()** e **trace()**, que podem ser redefinidos por classes derivadas. Notar que, tal como em **C/C++**, o método **trace()** é apenas gerado e invocado quando é indicada a opção **-T**. O método **int select(Tree)** efectua a selecção das instruções, retornando 0 (zero) em caso de sucesso e **1** (um) caso não seja possível emparelhar alguma parte da árvore sintáctica indicada como argumento.

13.2.8 Geração postfix com pbrug

A ferramenta **burg** efectua a selecção das instruções, permitindo que o código seja gerado de diversas formas. Nesta secção exploramos a possibilidade de utilizar a máquina de pilha definda no **postfix** para gerar as instruções. A linguagem **compact**, que funciona como exemplo, define instruções e expressões. As instruções não produzem resultados, como nas outras linguagens, enquanto as expressões produzem um resultado. Assim, definem-se dois símbolos não terminais `stat` e `reg`, que correspondem aos resultados produzidos pelas instruções e expressões, respectivamente. Estes símbolos não terminais correspondem aos símbolos `stmt` e `expr` utilizados ao definir a gramática da linguagem. Por outro lado, os símbolos `list` e `args` da gramática da linguagem podem ser incluídos, respectivamente, nos símbolos `stat` e `reg`, pois não realizam qualquer operação. Também é possível fazer corresponder a símbolo da gramática da linguagem vários símbolos não terminais, tal como referido na utilização de custos variáveis, por exemplo.

No caso das expressões o resultado é deixado no topo da pilha e não num registo, pois o **postfix** é uma máquina de pilha. Os custos assumem que cada instrução **postfix** tem um custo unitário, permitindo acrescentar regras de optimização que agregem conjuntos de instruções.

```
stat:   LIST(stat,stat)    { }
stat:   STRING           9 { char *l = mklbl(++lbl);
                             fprintf(outfp, pfRODATA pfALIGN pfLABEL pfSTR
                                     pfTEXT pfADDR pfCALL pfCALL pfTRASH,
                                     l, p->value.s, l, "_prints",
                                     "_println", 4); }
stat:   PRINT(reg)       3 { fprintf(outfp, pfCALL pfCALL pfTRASH,
                                     "_printi", "_println", 4); }
stat:   READ             1 { IDnew(0, p->value.s, IDtest);
            fprintf(outfp, pfCALL pfPUSH pfADDRA, "_readi", p->value.s); }
stat:   JZ(reg,ETIQ)     1 { fprintf(outfp, pfJZ, p->SUB(1)->value.s); }
stat:   JNZ(reg,ETIQ)    1 { fprintf(outfp, pfJNZ, p->SUB(1)->value.s); }
stat:   JMP              1 { fprintf(outfp, pfJMP, p->value.s); }
```

```
stat:   LABEL              { fprintf(outfp, pfLABEL, p->value.s); }
stat:   ASSIGN(VARIABLE,reg) 1 { IDnew(0, LEFT_CHILD(p)->value.s, IDtest);
               fprintf(outfp, pfADDRA, p->SUB(0)->value.s); }
stat:   reg             1 { fprintf(outfp, pfTRASH, 4); }
reg:    VARIABLE        1 { IDnew(0, p->value.s, IDtest);
               fprintf(outfp, pfADDRV, p->value.s); }
reg:    INTEGER         1 { fprintf(outfp, pfIMM, p->value.i); }
reg:    ADD(reg,reg)    1 { fprintf(outfp, pfADD); }
reg:    UMINUS(reg)     1 { fprintf(outfp, pfNEG); }
```

Apenas o operador binário `ADD` é apresentado, mas a mesma solução é aplicável aos restantes operadores binários da linguagem **compact**, ou seja, `SUB`, `MUL`, `DIV`, `MOD`, `EQ`, `NE`, `LT`, `LE`, `GE` e `GT`.

Para optimizar o código basta identificar a sub-árvore que instruções mais complexas do **postfix** realizam. Por exemplo, as sequências constituídas por uma comparação seguida de um salto `JZ` ou `JNZ` podem ser substituídas por um salto condicional. Notar que do ponto de vista gramatical os elementos da sequência são gerados em zonas afastadas. Enquanto a comparação é gerada por uma expressão, o salto é gerado por uma instrução. No caso do **compact**, são as instruções **if** e **while** que geram os saltos `JZ` e `JNZ` respectivamente. Ao juntar a sequência é necessário ter em atenção que nos saltos `JZ` a condição deve ser negada.

```
stat:   JZ(LT(reg,reg), ETIQ)   1 { fprintf(outfp, pfJGE,
                                          p->SUB(1)->value.s); }
stat:   JNZ(LT(reg,reg), ETIQ)  1 { fprintf(outfp, pfJLT,
                                          p->SUB(1)->value.s); }
```

Embora seja apresentados apenas dois exemplos, deverão ser indicadas todos os 12 saltos condicionais possíveis em **compact**.

Outra optimização possível consiste em identificar incrementos a variáveis, ou seja instruções do tipo `i = i + 1`, e gerar a instrução de incremento. Utilizando um custo variável é possível identificar se os identificadores do lado esquerdo e do lado direito da atribuição referem a mesma variável. Caso se trate da mesma variável o custo é de 2 (`ADDR` e `INCR`) O custo é muito elevado, caso sejam variáveis distintas, para permitir que as regras já existentes possam ser utilizadas. O custo variável pode ser calculado pela rotina `sameVar` e definido na zona das declarações.

```
static int sameVar(NODEPTR_TYPE p) {
  return strcmp(LEFT_CHILD(p)->value.s,
                LEFT_CHILD(RIGHT_CHILD(p))->value.s) ? 32767 : 2;
}
```

```
%%
stat:   ASSIGN(VARIABLE,ADD(VARIABLE,INTEGER))   sameVar {
              IDnew(0, LEFT_CHILD(p)->value.s, IDtest);
              fprintf(outfp, pfADDR pfINCR, LEFT_CHILD(p)->value.s,
                      RIGHT_CHILD(RIGHT_CHILD(p))->value.i);
        }
```

13.2.9 Reserva de registos com pbrug

Recorrendo à linguagem **Compact** como exemplo, pode-se utilizar a ferramenta **burg**, em conjunto com uma reserva de registos *greedy*, para gerar código PENTIUM directamente. Para tal, definem-se os nomes dos registos a utilizar (`name`) e um conjunto de variáveis em igual número (`busy`) usadas para indicar se o registo está livre ou em uso. A rotina `getReg()` devolve um registo livre representado pelo índice nos vectores anteriores. Por simplificação não é efectuado *spilling* de registos, pelo que é gerado um erro quando já não existem registos disponíveis. O número do registo utilizado para armazenar o valor representado por cada nó da árvore é guardado no campo `place` da árvore sendo referido por `PLACE` para maior clareza.

```
static char *name[] = { "eax", "ecx", "edx", "ebx", "esi", "edi" };
static char busy[6];
static int getReg();
#define PLACE(p) ((p)->place)
```

As declarações dos símbolos terminais e a gramática resultante para a linguagem Compact pode ser resumida a:

```
%term LIST=';' ASSIGN='=' ADD='+' SUB='-'
%term MUL='*' DIV='/' MOD='%' LT='<' GT='>'
%include "y.tab.h"
%%
stat:  LIST(stat,stat) { }
stat:  STRING 1 { char *l = mklbl(++lbl);
                  printf(" segment .rodata\n align 4\n"
                         "%s: db '%s', 10, 0\n"
                         " segment .text\n push dword $%s\n"
                         " call _prints\n add esp,4\n",
                         l, p->value.s, l);
                }
stat:  PRINT(reg) 1 { printf(" push dword %s\n call _printi\n"
                             " call _println\n add esp, 4\n",
                             name[PLACE(p)=PLACE(LEFT_CHILD(p))]);
                      busy[PLACE(LEFT_CHILD(p))]=0;
```

```
                    }
stat:  READ 1 { printf(" call _readi\n mov [%s], eax\n", p->value.s); }
stat:  JZ(reg,ETIQ) 1 {
                printf(" jz %s, %s\n", name[PLACE(p)=PLACE(LEFT_CHILD(p))],
                        p->value.sub.n[1]->value.s);
                busy[PLACE(LEFT_CHILD(p))]=0; /* free child register 1 */ }
stat:  JMP 1 { printf(" jmp %s\n", p->value.s); }
stat:  LABEL { printf("%s:\n", p->value.s); }
stat:  ASSIGN(VARIABLE,reg)   19 {
                IDnew(0, LEFT_CHILD(p)->value.s, IDtest);
                printf(" mov [%s], %s\n", LEFT_CHILD(p)->value.s,
                        name[PLACE(RIGHT_CHILD(p))]);
                busy[PLACE(RIGHT_CHILD(p))]=0; /* free child register 1 */ }
stat:  reg { PLACE(p) = 0; /* free register */ }

reg:   VARIABLE 18 { printf(" mov %s, [%s]\n",
                        name[PLACE(p) = getReg()], p->value.s); }
reg:   INTEGER 4 { printf(" mov %s, %d\n",
                        name[PLACE(p) = getReg()], p->value.i); }
reg:   ADD(reg,reg)     3 {
          printf(" add %s, %s\n", name[PLACE(p)=PLACE(LEFT_CHILD(p))],
                                  name[PLACE(RIGHT_CHILD(p))]);
          busy[PLACE(RIGHT_CHILD(p))]=0;
        }
reg:   UMINUS(reg)       3 {
                printf(" neg %s\n", name[PLACE(p)=PLACE(LEFT_CHILD(p))]); }
```

Na gramática acima foram omitidos os diversos operadores binários, semelhantes à operação de soma, e outras operações semelhantes idênticas às apresentadas.

Para explorar as capacidades do processador podemos adicionar regras específicas, além das genéricas atrás indicadas. No caso do processador PENTIUM podemos guardar valores imediatos directamente na memória, somar com valores imediatos, combinar a instrução de salto com a comparação que o controla ou incrementar variáveis directamente na memória,

```
stat:  ASSIGN(VARIABLE,INTEGER) 20 {
                IDnew(0, LEFT_CHILD(p)->value.s, IDtest);
                printf(" mov [%s], dword %d\n",
                        p->value.sub.n[0]->value.s,
                        p->value.sub.n[1]->value.i); }
reg:   ADD(reg,INTEGER)   4 {
                printf(" add %s, %d\n",
                        name[PLACE(p)=PLACE(LEFT_CHILD(p))],
                        RIGHT_CHILD(p)->value.i); }
stat:   JZ(LT(reg,reg), ETIQ)    2 { char *cond = "jge";
```

```
          printf(" cmp %s, %s\n %s %s\n",
                  name[PLACE(LEFT_CHILD(LEFT_CHILD(p)))],
                  name[PLACE(RIGHT_CHILD(LEFT_CHILD(p)))],
                  cond, RIGHT_CHILD(p)->value.s);
          busy[PLACE(LEFT_CHILD(LEFT_CHILD(p)))] = 0;
          busy[PLACE(RIGHT_CHILD(LEFT_CHILD(p)))] = 0;
        }
stat:  ASSIGN(VARIABLE,ADD(VARIABLE,INTEGER))   isSame {
                if (IDfind(0, LEFT_CHILD(p)->value.s, 0) >= 0)
                        printf(" add [%s], dword %d ; incr\n",
                        LEFT_CHILD(p)->value.s,
                        RIGHT_CHILD(RIGHT_CHILD(p))->value.i);
}
```

Num processador complexo, e com centenas de instruções, muitas outras optimizações são possíveis.

14 Ambiente de apoio à execução

O compilador, na sua forma mais elaborada, permite converter uma linguagem de alto nível em código máquina, em geral sob a forma de `assembly`. No entanto, para pode executar o código produzido é necessário mais algum código que invoque o código gerado e algumas rotinas utilitárias de suporte. Estas rotinas são genéricas designadas por ambiente de apoio à execução ou *runtime*. Estes ambientes variam consoante a linguagem e a arquitectura onde o programa executa.

A linguagem pode requerer inicializações específicas, como por exemplo os construtores da instâncias estáticas em **C++**, e oferecer rotinas específicas para imprimir ou ler valores. Os valores dependem quer dos tipos de dados da linguagem quer das funcionalidades oferecidas pelo sistema operativo, uma vez que operações de leitura e escrita necessitam de acesso a periféricos.

14.1 Utilização de outro ambiente

A solução mais simples consiste em utilizar o ambiente de apoio à execução de outra linguagem. Esta solução é utilizada por numerosas linguagens, muitas delas utilizando o ambiente da linguagem **C** como suporte de apoio à execução. Nesta abordagem, o código gerado terá de ter um formato compatível com o ambiente **C** e utilizar as rotinas disponíveis nas bibliotecas de **C**.

Assumindo, por exemplo, o ambiente do compilador de **C** oferecido pelo compilador da GNU, o **gcc**, o *assembly* gerado (`.s`) terá de ser compatível com o *assembler* da **GNU**, o **gas**. Alternativamente, pode-se utilizar o **nasm** para gerar ficheiros objecto (`.o`) em formato **elf** a partir do *assembly* normalmente gerado pelo compilador (`.asm`) em formato INTEL.

A fase final do processo de geração do executável não pode ser efectuada directamente pelo carregador (**ld**) pois teriamos de replicar a sequência executada pelo ambiente da GNU. Assim, será necessário invocar o compilador de **C** mas utilizando ficheiros objecto (`.o`) ou *assembly* (`.s`) em vez dos ficheiros escritos em **C** (`.c`).

Nesta abordagem, a execução inicia-se na rotina global `main` com a assinatura `int main(int argc, char *argv[], char *envp[])`. Tal como em **C** o programa pode optar por ignorar alguns dos últimos argumentos, que são de qualquer forma passado ao programa pelo ambiente de apoio à execução. Igualmente, ficam disponíveis as rotina habituais da linguagem **C**, como `printf`, `scanf`, `fopen`, `strlen` ou `atoi`, entre muitas outras.

14.2 Inicialização

O processo de inicialização, ou *bootstrapping*, depende da organização da arquitectura do processador que irá executar o código e do sistema operativo que controla a sua execução. Um processo inicia a sua execução num endereço específico, por exemplo `0x80000000`, ou num determinado símbolo cujo endereço é colocado num local específico do ficheiro executável.

Em **unix**, o carregador (*loader*) **ld**, utiliza o símbolo `_start` como rotina de inicialização, excepto se for indicado um outro endereço com a opção **-e**. Esta rotina é responsável por realizar a interface entre o sistema operativo que criou o processo e o programa a executar. Por exemplo, em **unix** o processamento dos argumentos é reduzido pois o formato é o mesmo. Em **windows**, a rotina de inicialização recebe uma cadeia de caracteres com os argumentos tal como foram digitados pelo utilizados, devendo a rotina criar as variáveis `argc` e `argv` decompondo a cadeia caracteres, construindo e contando cada um dos argumentos.

Outras inicializações podem incluir a definição da conversão de reais para inteiros pela unidade de vírgula flutuante, que em **C** é por truncatura e não por arredondamento. A invocação de construtores de bibliotecas ou instâncias estáticas necessita igualmente ser efectuada antes da invocação da rotina principal do programa, se for caso disso.

Finalmente, no retorno da rotina principal do programa, equivalente à rotina `main` da linguagem **C**, é necessário aproveitar o seu valor de retorno e terminar o processo, utilizando esse valor como código de terminação. Para tal é efectuada uma chamada ao sistema a pedir a terminação do processo. Em **unix** a chamada é efectuada directamente, utilizando a chamada `void exit(int)`. Para tal, em **linux-i386** é necessário colocar o valor a retornar no registo **ebx**, o valor 1 (um), pelo qual é reconhecida a chamada `exit` em **unix** no registo **eax** e efectuar a chamada gerando a instrução de interrupção `int 0x80`. Em **linux-arm**, como por exemplo no sistema **android**, o número da chamada é passada no registo **r7** (pois os registos **r0** a **r6** são utilizados para passar argumentos quer entre rotinas como nas chamadas ao sistema), o valor de retorno em **r0** e a chamada efectuada com a instrução **swe**. No entanto, em **windows** o programador não tem acesso às chamadas ao sistema, sendo estas tratadas pelas rotinas da biblioteca `KERNEL32.lib`. Desta forma, as diferenças entre os diversos sistemos operativos (*xp*, *vista*, *windows-8*, ...) são resolvidos através de uma versão específica da biblioteca. Assim, basta chamar a rotina `exit`, como qualquer outra rotina de **C** e utilizar a biblioteca na geração do executável.

14.3 Suporte básico

O ambiente de suporte à execução disponibilizado resulta de uma simplificação do ambiente desenvolvido para a linguagem **C**. Assim, um ficheiro *assembly*, dependente do sistema operativo e arquitectura, fornece a rotina de inicialização `_start`, salvaguarda o ambiente de invocação e disponibiliza uma rotina de leitura (`readb`) e uma de esrita (`prints`). O ambiente de invocação pode posteriormente ser utilizado pelas rotinas `int argc()`, `char *argv(int)` e `char *envp(int)` para obter os argumento habitual passados para a rotina `main`. A rotina de leitura `int readb()`, correspondente à rotina `getchar()` de **C**, lê um caracter, devolvendo `-1` (EOF) quando atinge o fim do ficheiro. A rotina de escrita `int prints(char *)` permite escrever uma cadeia de caracteres, terminada no caracter nulo (`0` ou NULL), devolvendo o número de caracteres efectivamente impressos.

Um conjunto adicional de rotinas utiliza as anteriores para fornecer funcionalidades adicionais básicas. Estas rotinas já não dependem da arquitectura ou sistema operativo, pelo o mesmo ficheiro (**lib.asm**) pode ser utilizado pelos diversos sistemas operativos com a arquitectura **i386**. O ficheiro **libarm.s** oferece as mesmas funcionalidades para processadores **arm**.

A rotina `int argc()` devolve o número de argumentos de invocação do programa. Os argumentos podem ser obtidos através da rotina `char *argv(int i)`, onde `i` representa o número do argumento. Não esquecer que o argumento `0` (zero) é o nome do programa e que `argv(argc())` devolve sempre `0` (zero), tal como em **C** onde `argv[argc] == 0`.

As variáveis de ambiente são devolvidas pela rotina `char *envp(int i)`, onde `i` começa em `0` (zero) e termina quando a rotina devolve `0` (zero), à semelhança da variável `envp` da linguagem **C**.

As rotinas `int strlen(const char *s)` e `int atoi(char *s)` são equivalentes às rotinas homónimas da linguagem **C**, enquanto a rotina `char *itoa(int val)` devolve uma cadeia de caracteres com representação do argumento inteiro `val`. Notar que a cadeia de caracteres é reaproveitada na próxima chamada à rotina, destruindo o valor anterior.

As rotinas `void println()`, `void printsp()` e `void printi(int i)` permitem imprimir um caracter de mudança de linha, um espaço branco e o valor inteiro passado como argumento, respectivamente.

Finalmente, as rotinas `char *readln(char *buf, int siz)` e `int readi()` permitem ler uma linha de texto para o bloco de memória `buf` até um

máximo de `siz` caracteres e ler um valor inteiro (devolve zero em caso de erro), respectivamente.

14.4 Suporte de vírgula flutuante

Caso a linguagem suporte variáveis em vírgula flutuante em precisão simples ou dupla, respectivamente equivalentes aos tipos de dados `float` e `double` da linguagem **C**, é necessário incluir o ficheiro **dbl.asm**. Este código serve apenas para compiladores experimentais e não substitui as chamadas da biblioteca de **C**, pois pode introduzir algum erro adicional nos últimos digitos significativos.

As rotinas disponibilizadas permitem a conversão entre cadeias de caracteres (ASCII) e o números em vírgula flutuante na precisão simples (`float`) e dupla (`double`):

```
double atod(char *s)};
char *dtoa(double d, int ndig, char *s)};
float atof(char *s)};
char *ftoa(float f, int ndig, char *s)};
```

Adicionamente, existem rotina para ler e escrever os mesmos formatos acima, utilizando as rotinas acima para converter de e para cadeias caracteres depois de ler e antes de escrever, respectivamente. As rotinas `float readr(void)` e `void printr(float)` permitem ler e escrever em precisão simples, enquanto as rotinas `double readd(void)` e `void printd(double d)` realizam as mesmas operações em precisão dupla.

14.5 Acesso ao sistema operativo

Para o acesso às funcionalidades do sistema operativo, como acesso a ficheiros ou processos, é necessário converter as chamadas a rotinas em chamadas ao sistema. Como vimos acima, em **windows** este processo é executado pelo ficheiro KERNEL32.LIB. Em **unix**, os argumentos necessitam ser retirados e pilha e colocados nos registos, além de ser indicado no registo próprio o número da operação a realizar pelo sistema. No processador **arm** o processo resume-se a preencher o registo **r7** com o número da chamada e a efectuar a interrupção (`swe`), pois os argumentos para rotinas do programador ou chamadas ao sistema são passados nos mesmo argumentos.

O ficheiro **sys.asm** para processadores **i386**, ou **armsys.s** para processadores **arm**, efectua as conversões necessárias para as chamadas ao sistema oferecidas pelo sistema operativo **unix**.

O ficheiro **fork.c** oferece uma versão nula da chamda `fork` quando esta não existe no sistema operativo, pelo que pode ser utilizado em sistemas operativos tipo **windows**. Notar que não realiza a chamada pois esta não existe, apenas devolve um código de erro na esperança que o programador tenha previsto formas alternativas de realizar a operação.

14.6 Outras bibliotecas

Em princípio, qualquer biblioteca de **C** ou de outra linguagem compilada pode ser utilizada, desde que sejam respeitadas as regras de passagem de argumentos e recolha de resultados. Assim, a invocação das rotinas de matemática da biblioteca **libm.a** como `sqrt`, `sin`, `atan2` ou `cosh`, podem ser invocada directamente, desde que as regras de passagem de argumentos e retorno de resultados sejam respeitadas e a biblioteca incluída no processo de carregamento.

No entanto, algumas bibliotecas, como a biblioteca de **C** da GNU, requerem inicializações especiais apenas realizadas pelo respectivo ambiente de apoio à execução. Aliás diversas bibliotecas da GNU utilizam um processo de inicialização importado da inicialização de instâncias de **C++** obrigando a um processo de geração em três fases e a um processo de inicialização específico. Nomeadamente, é gerado um primeiro executável relocatável (**ld -r**), isto é com símbolos indefinidos. Se estes símbolos indefinidos são depois classificados em construtuores, destrutores e outros, dependendo do seu nome. O programa **collect** (modernamente **collect3**) constroi depois uma lista de construtores e destrutores que é incluída numa segunda geração do executável. A rotina de inicialização itera nestas duas listas, antes e depois de invocar a rotina `main` para garantir ao ser invocada a rotina `main` todos os objectos estáticos se encontram devidamente inicializados.

O ambiente de apoio à execução fornecido é simples e não tem suporte para construtores e destrutores, desta forma algumas bibliotecas da **GNU** não podem ser directamente utilizadas. No entanto, a biblioteca **ulibC** oferece funcionalidades semelhantes à biblioteca de **C** da **GNU** sem necessidade de inicializações especiais. Notar a maioria das outras realizações da biblioteca linguagem **C** e de outras bibliotecas não apresentam qualquer limitação, em especial aquelas que não pretendem ser utilizadas simultaneamente em **C** e **C++**.

15 nasm: conversor para código máquina (*assembler*)

O *assembler*, programa montador em português, é um programa que permite substituir as designações simbólicas das instruções máquina do processador a utilizar pelos respectivos códigos de operação.

Nos anos 50, com o aumento da dimensão e complexidade dos programas, substitui-se os códigos de operação por menemónicas que designavam, quer os códigos de operação (soma de inteiros 'long', designado por *addl*), quer os registos do processador (apontador da pilha, ou *stack pointer*, extendido para 32 bits, designado por *esp*). Torna-se também possível utilizar nomes simbólicos (etiquetas ou *labels*) para designar entidades definidas pelo utilizador, como rotinas ou variáveis. Esta nova linguagem designa-se por *assembly* e pode ser convertida para os códigos de operação mediante a utilização de uma ferramenta designada por *assembler*.

Referir que os *assemblers* continuam em uso até aos dias de hoje, como passo intermédio do processo de compilação. A sua utilização directa pelo programador só se justifica em casos muito específicos, como manipulação de certos componentes do computador ou optimização de código mais sofisticada que a disponibilizada pelo compilador. Mesmo nestes casos, o programador recorre ao compilador para gerar o *assembly* e posteriormente acrescenta ou modifica apenas as instruções que não foram gerada da forma desejada pelo compilador.

Uma rotina em **C** que realize a soma de dois inteiros

```
int add(int x, int y) {
    return x + y;
}
```

poderá ser escrita em *assembly*, no caso **i386** gerado pelo **gcc**, para poesterior tratamento por um *assembler* como:

```
add:
        pushl %ebp
        movl %esp,%ebp
        movl 8(%ebp),%eax
        addl 12(%ebp),%eax
        leave
        ret
```

O *assembler*, em UNIX, é designado por `as`, converte o código acima na sequência de códigos de operação `55 89 E5 8B 45 08 03 45 0C C9 C3` que será executada pelo processador. Esta sequência de códigos de operação é guardada num ficheiro em formato binário, designado por *ficheiro objecto*[1].

Como existe uma relação directa entre cada código de operação e a respectiva menemónica, é possível obter o programa em *assembly* original (onde os nomes simbólicos que foram substituídos pelas posições de memória que passaram a ocupar) a partir do *ficheiro objecto*, designado pela extensão **.o** em UNIX e por **.obj** em DOS/WIN.

```
00000000 <add>:
   0:   55              pushl  %ebp
   1:   89 e5           movl   %esp,%ebp
   3:   8b 45 08        movl   0x8(%ebp),%eax
   6:   03 45 0c        addl   0xc(%ebp),%eax
   9:   c9              leave
   a:   c3              ret
```

Os números da coluna da esquerda representam as posições de memória onde ficam guardados os códigos de operação, as colunas centrais representam os valores numéricos desses códigos, e as colunas da direita o programa original em *assembly*. O resultado acima foi obtido com a aplicação `objdump` em **linux**.

Os exemplos apresentados nesta secção baseiam-se no *assembler* normal do **linux**, o **gas**. Este *assembler* apresenta uma notação algo trabalhada e onde o resultado é colocado no segundo argumento. Contudo, o principal problema deste *assembler* é o facto de ter sido desenhado para funcionar de *backend* para compiladores, como o **gcc**, que lhe fornecem sempre código correcto. Desta forma, as mensagens de erro e o tratamento dos erros é menos boa.

15.1 O nasm

O NASM, ou *netwide assembler*, é um *assembler* para a família de processadores 80x86, tendo por objectivo a portabilidade e modularidade. Permite gerar código para DOS-16bit, Win32, COFF, NetBSD/FreeBSD, `a.out` e ELF. Além disso utiliza uma sintaxe simples, semelhante à da Intel.

[1] não tem qualquer relação com programação por objectos.

Embora, não seja necessário um conhecimento profundo do NASM, a consulta do seu manual pode-se revelar inestimável em momentos de apuro. O manual, além de descrever as opções do *assembler*, inclui uma descrição simplificada de todas as instruções do 80x86.

Uma vez que a utilização do *assembler* pressupõe que o código é gerado por um compilador, muitas das facilidades oferecidas podem ser omitidas como, por exemplo, macros ou inclusão de ficheiros. Também o ambiente de trabalho utilizado reduz significativamente a informação necessária. Assim, as opções da linha de comandos reduzem-se a '-felf' para gerar código ELF e '-o file.o' para indicar o ficheiro de destino, caso o nome difira do argumento. Por exemplo, `nasm -felf tri.asm` produz um ficheiro `tri.o` em formato ELF.

15.1.1 Declarações no ficheiro objecto

Um ficheiro objecto é constituído, principalmente, por três secções:

.text uma área protegida para escrita (read-only) onde são colocadas as funções e as strings constantes (em C: char *s="texto constante";)

.data uma área de dados iniciados onde são colocadas as variáveis globais não nulas e as strings modificáveis (em C: char s[ ]="texto modificável";).

.bss uma área de dados não iniciados cujo valor será iniciado a 0 (zero) na execução do programa. Esta área justifica-se pelo facto de muitas variáveis não estarem iniciadas (em C: int x, y[20];) e não necessitarem de ocupar espaço no ficheiro executável, sendo o espaço criado antes do início da execução.

A inciação de valores iniciados é declarado através das pseudo-instruções `db` para bytes, `dw` para inteiros e `dd, dq, dt` para reais em precisão simples, dupla e quadrupla, respectivamente:

```
section .data
    $x   db    0x55            ; em C: char x = 0x55;
    $y   db    12,24,36        ; em C: char y[] = { 12,24,36 };
    $h   db    'hello',10,0    ; em C: char h[] = "hello\n";
    $n   dw    112590          ; em C: int n = 112590;
    $pi  dq    3.1415926535    ; em C: double pi=3.1415926535;
```

A declaração de valores não iniciados utiliza as pseudo-instruções `resb`, `resw`, `resd`, `resq`, `rest` para os tipos de dados com as mesmas terminações dos iniciados.

```
section .bss
    $buf    resb    80      ; em C: char buf[80];
    $err    resw    1       ; em C: int err;
    $tab    resd    100     ; em C: float tab[10][10];
```

Notar que se podem declarar valores não iniciados no segmento `.data`, mas a única diferença é o facto de passar a ocupar espaço em disco.

15.1.2 Constantes

As constantes numéricas são representadas em formato decimal. No entanto, mediante a utilização dos sufixos H, Q e B, podem ser representadas em hexadecimal, octal e binário. O formato hexadecimal pode, ainda, utilizar o prefixo 0x, tal como em C, ou o prefixo $ (Borland), tendo em conta que, caso o número comece por uma letra (de 'a' a 'f'), deve ser precedido de um 0 (zero):

```
mov    eax, 100        ; decimal
mov    eax, 0xa2       ; hexadecimal
mov    eax, 0a2h       ; hexadecimal
mov    eax, $0a2       ; hexadecimal: necessario o 0 (zero)
mov    eax, 777q       ; octal
mov    eax, 10010011b  ; binario
```

Os cadeias de caracteres só podem ser utilizadas em algumas pseudo-instruções. Notar que, para carregar o acumulador (eax) com o endereço de uma cadeia de caracteres, é necessário declarar a cadeia no segmento apropriado e utilizar o endereço, designado por um nome simbólico, para carregar o endereço.

Uma cadeia de caracteres pode ser delimitada por plicas ou aspas, mas deverá iniciar e terminar com o mesmo símbolo. Três cadeias de caracteres iguais, mas ocupando posições de memória distintas e consecutivas:

```
db   'hello',10
db    'h','e','l',"l","o",0xa
db    "hel",'lo',$0a
```

As constantes em vírgula flutuante só podem ser utilizadas nas pseudo instruções `DD`, `DQ` e `DT`, sendo obrigatória a utilização do ponto decimal e começando necessariamente por um dígito decimal. A componente exponencial é facultativa.

```
dd      1.2
dd      12.
dq      1.e10
dq      1.e+10
dq      1.e-10
```

Finalmente, convém referir que um *assembler* não controla os tipos declarados, pelo que nada impede de utilizar um valor real como um endereço ou uma cadeia de caracteres como um valor inteiro. Desta forma, é responsabilidade do compilador, ou do programador, garantir que os valores são utilizados coerentemente pois do ponto de vista do *assembler* são apenas números e posições de memória.

15.1.3 Labels locais e globais

O NASM trata de forma especial os *labels* começados por um ponto. Assim, os *labels* iniciados por um ponto são considerados locais, isto é associados ao último *label* global. Esta característica permite produzir *labels* localmente sem perigo de conflitos e pode ser utilizado com vantagens na realização de instruções condicionais e ciclos.

```
label1:     ; codigo
.loop:      ; mais codigo
            jne .loop
            ret
label2:     ; outro codigo
.loop:      ; codigo apos outro
            jne .loop
            je label1.loop
            ret
```

15.1.4 Alinhamento de dados e código

O carregamento de informação a partir da memória, seja código ou dados, é feita em blocos. Estes blocos podem ser de 8, 16, 32, 64, 128 ou mais bits. Hoje em dia, a maioria dos processadores usa valores entre 32 (80x86) e 128 (sparc). Quanto maior for o número de bits mais informação é carregada de cada vez, aumentando a largura de banda da comunicação com a memória de massa, permitindo aumentar a velocidade de processamento. No entanto, existe um senão, uma variável, mesmo que ocupe apenas dois bytes, pode ficar com um byte num bloco e outro byte noutro, obrigando a dois carregamentos. Desta forma, os processadores obrigam a que a maioria das entidades de alto

nível, variáveis, estruturas e funções começem no início de um bloco. Este facto obriga a que, antes de iniciar uma função (antes do *label* que a designa), seja necessário forçar o alinhamento. No entanto, ao forçar o alinhamento pode ser necessário preencher o fim do bloco actual com alguma informação. Este enchimento, designado de *padding*, é em geral efectuado com NOPs na zona de código e com 0 (zero) na zona de dados. Assim, o NASM oferece duas directivas de alinhamento ALIGN para código e ALIGNB para dados. Qualquer destas macros aceita como argumento a dimensão do bloco de alinhamento, um valor em bytes que é sempre uma potência de 2, que no caso do ELF 32bits é de pelo menos 4 bytes.

```
align 4            ; alinhar em multiplos de 4-bytes
align 16           ; alinhar em multiplos de 16-bytes
align 8, db 0      ; preencher com zeros em vez de NOPs
align 4, resb 1    ; alinhar em multiplos de 4-bytes no BSS
alignb 4           ; igual ao anterior
alignb 4, db 0x90  ; preencher com NOPs
```

Assim, para variáveis inteiras (32 bits) é necessário um alinhamento de 4 bytes. Notar que numa sequência de variáveis inteiras é necessário declarar o alinhamento apenas antes da primeira, pois a partir daí está alinhado. Por outro lado, uma variável em precisão dupla necessita de um alinhamento de 8 e uma de precisão quadrupla de um alinhamento de 16.

Notar que em **C** declarar uma estrutura com um 'char' seguido de um 'double', seguido de um 'short' e terminado por um 'long' tem uma dimensão significativamente superior a uma sequência 'long', 'short', 'char', 'double'. Tal pode ser verificado declarando as duas estruturas e imprimindo a sua dimensão (sizeof), e constatando que a soma de cada um dos elementos (15 bytes) não corresponde a nenhum dos casos.

```
struct_x:
var_byte:       resb 1
                alignb 2
var_short:      resw 1
                alignb 4
var_long:       resd 1
                alignb 8
var_double:     resq 1
var_str:        resb 80
```

15.1.5 Definição de símbolos globais e exteriores

Um ficheiro, por omissão, considera todos os *labels* locais. Assim, variáveis ou funções que necessitem ser referênciadas a partir de outros ficheiros, num ambiente de compilação separada, necessitam ser declarados globais. Em **C**, pelo contrário, todas as variáveis globais e funções são acessíveis a partir de outros ficheiros, excepto se tiverem o prefixo `static`. Desta forma, em NASM, torna-se necessário declarar um símbolo como global para que ele possa vir a ser reconhecido por outros ficheiros. Notar que, se dois ficheiros exportarem o mesmo símbolo global não podem estar juntos no mesmo executável, pois o mesmo nome designaria duas posições de memória distintas.

```
            global _main
_main:      ; codigo da funcao main
```

Da mesma forma, caso um programa pretenda reconhecer um símbolo, previamente declarado global num outro ficheiro, deve utilizar a directiva 'extern'. Esta directiva tem a mesma função que a sua homónima em **C**, tornando o símbolo conhecido localmente. Notar que, só na edição binária de ligações (linkage) é que os dois ficheiros são juntos e a referência a esse símbolo deixa de estar indefinida, embora conhecida, e passa a estar associada a uma posição de memória.

Um exemplo completo de um programa "hello world" escrito em NASM e utilizando a função de biblioteca 'prints' para imprimir a cadeia de caracteres:

```
segment .data
        $str    db      'hello world', 10, 0
segment .text
extern prints     ; e' uma funcao exterior (biblioteca)
global main       ; e' global para poder ser invocado
main:
      push dword $str ; coloca o endereco da string na pilha
      call prints     ; invoca a funcao de impressao
      pop  eax        ; retira o argumento da pilha
      mov  eax, 0     ; a funcao 'main' devolve 0 (zero)
      ret
```

No caso do formato ELF, que estamos a considerar, a directiva 'global' pode conter mais informação além do endereço do símbolo. Este facto permite ajudar o processo de edição de ligações e o reposicionamento dos símbolos (relocation), além de permitir

detectar mais alguns erros na compilação separada. Assim, além do endereço, um símbolo global pode conter um tipo: `function`, `data` ou `object` (`object` é sinónimo de `data`). Por exemplo,

```
global hashlookup:function, hashtable:data
```

Para mais, o espaço ocupado pelo símbolo pode ainda ser indicado, através de uma expressão inteira que pode conter referências para símbolos e *labels* ainda não declarados,

```
        global hashtable:data (hashtable.end - hashtable)
hashtable:
        dw     um, dois, tres
.end
```

É assumido um alinhamento de 4 bytes para os dados (`data` e `bss`) e de 16 bytes para o código (`text`).

15.1.6 Passagem de argumentos

Para que a passagem de argumentos e valores de retorno na invocação das funções funcione correctamente, é necessário definir uma convenção. Hoje em dia existem, essencialmente, duas convenções: **C** e **Pascal**. Em **Pascal** quem chama a função coloca os argumentos na pilha da esquerda para a direita, mas é a função chamada que os retira da pilha. Este facto obriga a que o número e tipo de argumentos em cada chamada seja exactamente igual ao da função chamada. A convenção **Win32** empurra os argumentos como o **C** (da direita para a esquerda) mas é o chamado que retira os argumentos, como em **Pascal**.

Em **C** quem chama a função coloca os argumentos antes da chamada e retira-os após o retorno. Este facto permite funções com número variável de argumentos, como o 'printf', bem como a utilização de apenas parte dos argumentos, como no caso dos 3 argumentos do 'main'. A principal desvantagem reside em replicar o código de eliminação dos argumentos por todas as chamadas, em vez de existir uma só eliminação na função chamada.

A passagem de argumentos em **C** usa a seguinte convenção:

- o chamador coloca os parâmetros na pilha, um após outro, pela ordem inversa (da direita para a esquerda, de tal forma que o primeiro argumento é o último a ser colocado).

- seguidamente o chamador executa a chamada (CALL), passando o controlo para o chamado
- o chamado ao receber o controlo cria, em geral (embora não seja necessário), uma marca na pilha (stack frame). Desta forma, os argumentos da função estão sempre acessíveis na mesma posição, relativa ao *stack frame*, durante toda a rotina. Inclusivamente, as variáveis locais podem também ser referidas relativamente ao *stack frame*, mesmo quando é necessário colocar variáveis temporárias na pilha por falta de registos no processador. Em 80x86 usa-se, pelas suas capacidades de endereçamento, o `EBP` (base pointer), pelo que o `EBP` antigo é colocado na pilha e novo topo da pilha `ESP` (stack pointer) é colocado em `EBP`. Assim,
  - **[EBP]** : contém o anterior valor de `EBP`.
  - **[EBP+4]** : contém o anterior endereço de retorno da função, implicitamente colocado na pilha pela instrução CALL.
  - **[EBP+8]** : contém o primeiro argumento.
  - **[EBP+12]** : contém o segundo argumento.
  - **[EBP-4]** : contém a primeira variável local.
  - **[EBP-8]** : contém a segunda variável local.
- o chamado cria espaço para as variáveis locais diminuindo o valor do `ESP`, notar que a pilha tem a origem no topo da memória, logo cresce de dimensão reduzindo o seu valor. Assim, substraindo 12 bytes ao `ESP` cria-se espaço para 3 variáveis inteiras. Embora inicialmente os valores dessas variáveis locais possa ser obtido por deslocamentos face a `ESP` no decorrer da execução da rotina pode ser necessário utilizar a pilha para outros valores, sendo necessária uma actualização permanente dos deslocamentos face a `ESP`. Por outro lado, o deslocamento face a `EBP` é fixo durante a execução da rotina, mas gasta mais um registo do processador, o que pode ser crítico em determinadas situações.
- caso a função retorne um valor, isto é não seja `void`, deve fazê-lo preenchendo o seu valor no acumulador, ou seja `al` ou `ax` ou `eax` dependendo da dimensão. O retorno de valores reais é feito no `ST0`. No caso de estruturas, estas são copiadas para um local na memória e o seu endereço é devolvido no acumulador.
- imediatamente antes de retornar, a rotina repõe o valor do `ESP` a partir do valor do `EBP` e o valor de `EBP` a partir do seu conteúdo. Finalmente, retorna com uma instrução RET.
- quando o chamador retoma o controlo, os parâmetros ainda estão na pilha de dados pelo que devem ser retirados somando a sua dimensão total ao valor do `ESP`

(notar que executar tantos POPs quantos os argumentos é uma operação muito mais lenta para mais de um argumento).

Exemplo de uma chamada a uma função que subtrai 9 de 5:

```
extern printi, println
global main
main:
push  dword 5      ; segundo argumento
push  dword 9      ; primeiro argumento
call  subtrai      ; chama a funcao
add   esp, byte 8  ; retira os dois argumentos da pilha
push  eax          ; coloca o retorno de subtrai como argumento
call  printi       ; invoca a funcao de impressao
pop   eax          ; retira o argumento da pilha
call  println      ; muda de linha
mov   eax, 0       ; a funcao 'main' devolve 0 (zero)
ret
subtrai:
; as tres instrucoes podem ser substituidas por: 'enter 0x40'
push    ebp           ; guarda o antigo frame pointer
mov     ebp, esp      ; novo frame pointer <- stack pointer
sub     esp, 0x40     ; reserva 64 bytes para variaveis locais
; outro codigo da funcao que use as variaveis locais
mov     ebx, [ebp+8]  ; coloca o primeiro argumento em EBX
mov     eax, [ebp+12] ; coloca o segundo argumento em EAX
sub     ebx, eax      ; subtrai o primeiro (9) do segundo (5)
; sequencia de retorno: pode ser 'leave' e 'ret'
mov     esp, ebp      ; repoe o stack pointer no valor inicial
pop     ebp           ; repoe o antigo frame pointer
ret
```

16 ld: editor binário de ligações (*loader*)

O editor binário de ligações, ou carregador (*loader*, em inglês), é um programa que combina ficheiros objecto por forma a produzir outros ficheiros objecto. Se o ficheiro objecto obedecer a certas características pode ser executado. Caso contrário, os ficheiros resultantes podem ser bibliotecas dinâmicas ou outros ficheiros objecto.

O carregador funciona como a parte final do mecanismo de compilação separada disponível em muitas linguagens de programação. Em muitas linguagens, este processo de carregamento é executado por um programa de interface com o utilizador. Na realidade, nem **cc** nem **gcc** são o compilador de **C** mas um outro programa indirectamente executado por estes, designados por 'ccom' ou 'cc1', como pode ser constatado usando a opção '-v' no **gcc**.

16.1 Compilação separada

O princípio da compilação separada permite que um módulo possa utilizar símbolos, funções e variáveis, definidos noutro módulo. Para tal o outro módulo deverá exportar esse símbolo como global, por oposição aos símbolos que permanecem locais, e este módulo deverá importar esse símbolo, marcando-o como indefinido. Ao marcar um símbolo como indefinido, apenas a sua posição em memória é desconhecida, sendo temporariamente preenchida com o valor 0 (zero).

Quando se realiza o carregamento conjunto dos dois módulos, o carregador constata a posição em memória do símbolo exportado colocando o valor dessa posição de memória em todas as ocorrências desse símbolo indefinido nos outro módulos. Notar que um módulo pode exportar alguns símbolos e importar outros.

16.2 Criação de executáveis

O principal objectivo do carregador é combinar diversos ficheiros objecto por forma a formar um executável. Um ficheiro pode ser considerado executável se tiver um ponto de entrada e não tiver símbolos indefinidos. No entanto, para que o programa possa executar correctamente é necessário que, antes de chamar a função `main` em **C** ou código equivalente em outras linguagens, a imagem recém criada seja preparada. Nomeadamente em **C++** é necessário invocar os construtores dos objectos estáticos antes de invocar a função `main` e invocar o destruídores (*destrutors*) após o retorno desta. Notar que um programa necessita pelo menos terminar o processo, pelo que nas linguagens mais simples pode-se resumir a `exit(main(argc,argv,envp))`. Por outro lado, um

programa não trivial necessita igualmente de bibliotecas de funções previamente compiladas. Assim, a geração de um executável pode ser efectuada com

```
prompt$ ld /lib/crt0.o hello.o -lc
```

onde '/lib/crt0.o' é o ficheiro de arranque para a linguagem **C** e outras linguagens semelhantes, 'hello.o' é o programa a compilar e '-lc' designa a biblioteca da linguagem **C** correspondendo, em geral, ao ficheiro '/lib/libc.a'.

O processo de carregamento produz o resultado, seja este executável ou não, concatenando os vários ficheiros com a extensão '.o', pela ordem que são indicados nos argumentos. Como todos estes ficheiros são incluídos, nenhum símbolo global pode estar definido em mais de um desses ficheiros. Depois são resolvidos todos os símbolos indefinidos, isto é, cada referência a função ou variável importada é substituída pelo endereço onde ficou colocada. Se ficarem ainda símbolos indefinidos, que não existam em nenhum dos ficheiros '.o' carregados, o carregador procura sequencialmente nas bibliotecas cada um desses símbolos.

16.2.1 Carregamento de bibliotecas

A grande diferença entre indicar uma biblioteca ou os diversos ficheiros '.o' que a constituem reside no facto de esses ficheiros não serem todos carregados no ficheiro final, apenas aqueles ficheiros '.o', designados por módulos da biblioteca, que contêm os símbolos indefinidos são carregados. Assim, se apenas a função `printf` estiver indefinida, apenas o ficheiro que a contém é carregado, em geral designado por `printf.o` (ver ar). Se ao carregar este módulo surgir um novo símbolo indefinido, provavelmente `write`, então também o respectivo módulo será carregado. O processo repete-se até que a biblioteca não contenha mais nenhum dos símbolos ainda indefinidos.

Refira-se que o carregador, ao ser invocado apenas com bibliotecas como argumentos, produz um ficheiro só com informação de controlo e sem dados, pois não haveria nenhum símbolo indefinido inicial. O facto de não haver símbolos indefinidos implica que nenhuma biblioteca será utilizada.

O carregamento das bibliotecas é feito pela ordem que surjem nos argumentos do comando de carregamento, logo caso duas bibliotecas definam um mesmo símbolo global apenas o módulo que define esse símbolo na primeira biblioteca será carregado. Por outro lado, ao carregar um módulo da segunda biblioteca pode surgir um símbolo ainda não carregado, mas existente na primeira biblioteca. Neste caso, a primeira biblioteca

não volta a ser analisada pelo que se torna necessário repetir o seu nome na linha de comando após a segunda biblioteca, por exemplo '-lc -ly -lc'.

O carregamento de bibliotecas é efectuada através das opções '-l' e '-L', sendo ambos argumentos posicionais, isto é, a ordem pela qual aparecem é relevante:

-l*foo* procura símbolos no ficheiro de arquivo (biblioteca) designado por `lib`*`foo`*`.a` existente na directoria corrente ou na lista de directórios de busca. Caso se pretenda um executável de menores dimensões pode-se optar por gerar um executável com as bibliotecas carregadas dinamicamente, sendo utilizado o ficheiro `lib`*`foo`*`.so`, em substituição do anterior.

-L*dir* acrescenta o directório indicado à lista de directórios de busca. Esta opção pode ser indicada diversas vezes, quer para acrescentar diversos directórios, quer entre as designações das bibliotecas para alterar a ordem de busca das mesmas.

Por exemplo, a sequência '-L/tmp -lc -L/misc -lc -L/lib -lc' utiliza em sucessivamente três bibliotecas 'libc.a' ou 'libc.so' existentes nos três directórios indicados. Notar que o facto de terem o mesmo não significa que tenham conteúdos iguais ou semelhantes.

16.2.2 Executáveis ligados estaticamente

Um executável ligado estaticamente não utiliza bibliotecas dinâmicas, ou seja bibliotecas com a extensão '.so'. Estes executáveis têm uma dimensão significativamente maior, que os ligados dinamicamente, pois contêm todo o código necessário à sua execução. Em contrapartida, podem ser transportados para sistemas que não disponham das bibliotecas dinâmicas que de outra forma necessitariam.

Para gerar um executável ligado estaticamente basta incluir a opção '-static' ou '-Bstatic' no carregamento.

16.2.3 Executáveis ligados a bibliotecas dinâmicas

Um executável ligado a bibliotecas dinâmicas não inclui essas bibliotecas no ficheiro final, reduzindo significativamente a dimensão deste. Em vez disso, inclui um carregador dinâmico no ficheiro, designado por 'ld.so', que antes de chamar a rotina 'main' mapeia no espaço de endereçamento do processo as bibliotecas necessárias. Como as bibliotecas são maioritariamente constituídas por funções e cadeias de caracteres constantes, isto é não modificáveis, podem ser partilhadas por vários processos. De facto, se diversos processos utilizarem uma determinada biblioteca dinâmica basta existir em memória uma única cópia desta, podendo cada processo mapeá-la no seu espaço de

endereçamento, eventualmente em posições distintas. Desta forma as necessidade de memória física, isto é a dimensão da RAM, pode ser inferior.

As principais desvantagens do carregamento dinâmico são a necessidade de fazer esse carregamento no início de cada execução e, principalmente, o facto de ao transportar o executável ter de se garantir que existe uma cópia dessa biblioteca na máquina de destino. Para evitar que, na máquina de destino, seja carregada uma biblioteca dinâmica com o mesmo nome mas não exactamente igual, por exemplo uma versão anterior ou posterior, estas bibliotecas têm um número de versão. Esta versão tem de ser rigorosamente verificada pois, caso contrário, pode-se erradamente assumir uma determinada posição de memória para uma função. Não esquecer que a determinação da posição ocupada pela função foi efectuada considerando a biblioteca dinâmica na máquina original.

Na geração de executáveis que utilizam bibliotecas dinâmicas recorre-se à opção '-Bdynamic' ou '-dy', embora esta seja a geração por omissão na maioria dos ambientes que suportam bibliotecas dinâmicas.

16.3 Agrupamento de ficheiros objecto

Se, no carregamento, os diversos ficheiros com a extensão '.o' são concatenados e as referências simbólicas entre eles resolvidas, então pode-se gerar ficheiros objecto intermédios com essas referências previamente resolvidas. Este processo reduz o número de ficheiros necessários no carregamento final e o tempo desse carregamento, pois apenas as referências exteriores ficaram por resolver. Notar que este novo ficheiro objecto pode continuar a ter símbolos indefinidos, sendo portanto do mesmo tipo que os ficheiros que lhe deram origem.

Estes ficheiros são criados indicando ao carregador que o ficheiro resultante pode conter símbolos indefinidos, opção '-r'. Por exemplo, 'ld -r -o grupo.o primeiro.o segundo.o terceiro.o' gera o ficheiro 'grupo.o' através da concatenação, e resolução das referências simbólicas internas, dos restantes argumentos.

16.4 Criação de bibliotecas dinâmicas

O princípio da criação de uma biblioteca dinâmica é o mesmo do agrupamento de ficheiros objecto, apenas as opções são distintas para que a manipulação da tabela de símbolos seja adequada. Notar que ao contrário de uma biblioteca estática, em que apenas os módulos referenciados simbolicamente são carregados, uma biblioteca dinâmica é sempre carregada integralmente mesmo que se necessite apenas de uma função.

Ao contrário do agrupamento de ficheiros objecto, uma biblioteca dinâmica pode referir outras bibliotecas, estáticas ou dinâmicas, das quais fica a depender. Caso dependa de bibliotecas dinâmicas estas também terão de ser carregadas sempre que esta o for.

A construção de uma biblioteca dinâmica recorre à opção '-Bshareable' ou '-shared'. Por exemplo, 'ld -shared -o libdyn.so um.o dois.o tres.o'.

Geralmente a opção '-Bshareable' é utilizada em conjunto com a opção '-Bsymbolic' que permite que os símbolos globais sejam associados aos símbolos da biblioteca. Normalmente é possível a um programa que use uma biblioteca dinâmica redefinir, e consequentemente ignorar, um símbolo definido nesta. Desta forma é usado o símbolo local e não o da biblioteca. Por exemplo, em 'ld -Bsymbolic -shared -o libdyn.so um.o dois.o tres.o -lc' qualquer dos três ficheiros objecto pode definir símbolos existentes na biblioteca '-lc'. Desta forma uma aplicação que use a biblioteca 'libdyn.so', e consequentemente a biblioteca '-lc', só usará símbolos de '-lc' se estes não estiverem definidos em 'libdyn.so'.

Finalmente, a opção '-E' permite que o carregamento seja efectuado pela chamada à função `dlopen()` e não apenas no início da execução do programa. Este processo permite que as bibliotecas sejam carregadas e libertadas durante a execução de acordo com o fluxo de execução. Pode-se ainda substituir uma biblioteca por outra com o programa em execução, ou mesmo iniciar o programa sem ter concluído a biblioteca. Claro que esta terá que existir no momento da chamada ao `dlopen()`. Os programas que pretendam efectuar este tipo de operações devem incluir o ficheiro `dlfcn.h` que declara as funções necessárias, incluídas na biblioteca '-ldl'. Caso a biblioteca a carregar necessite de usar símbolos definidos no programa que a carrega, este programa deve ser produzido usando a opção '-rdynamic'. Assim, não só as funções comuns podem ser partilhadas como pode ser realizado um mecanismo de *callback*, não ficando limitado aos valores de retorno das funções invocadas.

A geração da biblioteca dinâmica passa a ser

```
ld -E -Bsymbolic -shared -o libdyn.so um.o dois.o tres.o -lc
```

e um executável que a pretenda carregar durante a execução, e não apenas no início

```
ld -o prog prog.o -rdynamic -ldl
```

16.5 Criação de bibliotecas estáticas

A criação de bibliotecas estáticas não usa o carregador mas uma aplicação própria 'ar' pelo que não é tratada nesta secção. Para tal consulte a secção 17 na página 177.

17 ar: gestor de arquivos para ficheiros objecto (*bibliotecas*)

As bibliotecas, que podem ser designadas estáticas por oposição às bibliotecas dinâmicas anteriormente vistas, são meras colecções de ficheiros objecto. Estes ficheiros objecto podem ser adicionados, removidos, enumerados ou extraídos da biblioteca. O ficheiro objecto ao ser colocado no arquivo retém o seu conteúdo, permissões, datas de criação e modificação, utilizador e grupo, podendo essa informação ser reposta ao ser extraído.

Ao incluir-se uma biblioteca dinâmica, todo o seu conteúdo passa a ser incluído no programa. Pelo contrário, numa biblioteca estática apenas os ficheiros objecto necessários são incluídos no executável final. Para acelerar a selecção, cada biblioteca inclui uma tabela de símbolos que indica quais os símbolos definidos na biblioteca e quais os ficheiros objecto que definem cada um deles. Assim, a utilização de biblioteca permite tornar o processo de edição de ligações mais eficiente que manipular os ficheiros objecto separadamente. Além disso, permite designar e movimentar conjuntos de ficheiros objecto relacionados como uma entidade única.

17.1 Opções do arquivador

O arquivador tem a seguinte sintaxe '`ar -opções biblioteca ficheiros`', onde a biblioteca tem o formato **lib***nome***.a**.

De entre as opções do arquivador salientam-se:

-c criar a biblioteca.

-x extrair um ou mais ficheiros da biblioteca. O processo de extracção permite obter o ficheiro com os mesmos atributos do original, mas o ficheiro não é retirado da biblioteca, é apenas copiado.

-t enumera os ficheiros existentes na biblioteca, incluindo a tabela de símbolos, caso exista.

-r subtitui os ficheiros existentes na biblioteca pelos ficheiros com o mesmo nome indicados na linha de comando. Se o ficheiro ainda não existir na biblioteca é introduzido.

-d remove os ficheiros indicados da biblioteca.

-s constroi a tabela de símbolos (ver ranlib).

A manutenção da biblioteca pode ser efectuada por

prompt$ ar crs libteste.a teste1.o teste2.o teste3.o

podendo a regra de uma `Makefile` ser

```
libteste.a: teste1.o teste2.o teste3.o
        ar crs $@ $?
```

17.2 Construção da tabela de símbolos: ranlib

Uma vez que um dos objectivos das bibliotecas consiste em acelerar a edição de ligações, torna-se necessário construir uma tabela de símbolos comum a todos os ficheiros da biblioteca. Esta tabela é construída utilizando o comando `ranlib`, seguido do nome da biblioteca, ou recorrendo à opção `-s` do comando `ar` introduzido atrás. Em algumas versões do sistema UNIX, a tabela de símbolos é construída automaticamente pelo arquivador, pelo que o comando `ranlib` existe apenas para compatibilidade

18 Analisadores de ficheiros objecto

Um ficheiro binário não pode ser editado por um editor de texto, pelo menos de uma forma compreensiva. Além das principais ferramentas de manipulação de ficheiros objecto, *assembler* (as), arquivador (ar) e carregador (ld), existem um conjunto de ferramentas auxiliares para analisar os ficheiros objecto.

18.1 file

Este comando pode ser aplicado a qualquer ficheiro para descobrir o seu tipo, independentemente da sua extensão ou de ser um ficheiro de texto ou binário. O programa baseia-se num conjunto de regras que procuram inferir o tipo de ficheiro com base no seu conteúdo. No caso de ficheiros binários, cujo formato está devidamente especificado, o programa oferece indicações úteis. No entanto, no caso de alguns ficheiros de texto pode-se deixar enganar e considerar um ficheiro em **Java** como sendo um ficheiro em **C** ou **C++**. Na realidade, até o leitor pode ter algumas dúvidas em muitos casos. Para mais existem tantas linguagens de programação que se torna impossível encontrar regras que garantam a sua correcta e inequívoca identificação.

Por exemplo,

prompt$ file a.out
    a.out: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked (uses shared libs), not stripped

18.2 ldd

Este programa imprime as bibliotecas dinâmicas que o executável necessita para executar. Por exemplo,

prompt$ ldd a.out
    libc.so.6 => /lib/i686/libc.so.6 (0x4002e000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)

18.3 size

Este programa imprime a dimensão da área de código e dados de ficheiros objecto,

```
prompt$ size a.out
   text    data     bss     dec     hex filename
  22780     404   73932   97116   17b5c a.out
```

Notar que estes valores referem as dimensões de cada secção e não a dimensão do ficheiro (usar **ls -l**), nem incluem o cabeçalho dos ficheiros ou as áreas que contêm informação de ligação ou análise.

18.4 nm

Imprime os símbolos de ligação de um ficheiro objecto. No caso de ficheiros objecto relocatáveis os endereços são relativos ao início da secção a que pertencem, sendo reposicionados (relocatados) pelo editor de ligações. Embora existam muitos tipos de símbolos referiremos os mais comuns:

U indefinido, a entidade (função, variável ou vector constante) está declarada mas não definida. Assim, para ser possível gerar um executável, deve ser ligado com outro ficheiro que defina o símbolo em questão.

T função global, a função está definida e pode ser acedida de outros ficheiros.

t função local, a função está definida mas é local pelo que não pode ser acedida. Mesmo que utilize este ficheiro no processo de edição de ligações, este não permite eliminar um símbolo indefinido com o mesmo nome de outro ficheiro. Por outro lado, podem ser ligados diversos ficheiros com o mesmo símbolo local.

D dados iniciados globais.

d dados iniciados locais

B dados não iniciados globais.

...

18.5 strip

Remove a informação simbólica de um ficheiro objecto. Caso o ficheiro seja um objecto relocatável deixa de ser possível incluí-lo num executável. O executável fica com a dimensão mínima, mas deixa de ser possível utilizar um *debugger* simbólico.

prompt$ wc -c a.out
78010 a.out

prompt$ strip a.out

prompt$ wc -c a.out
25308 a.out

prompt$ file a.out
a.out: ELF 32-bit LSB executable, Intel 80386, version 1, dynamically linked (uses shared libs), stripped

prompt$ size a.out

text	data	bss	dec	hex	filename
22780	404	73932	97116	17b5c	a.out

prompt$ nm a.out
nm: a.out: no symbols

18.6 Outros comandos

Embora úteis, os comandos seguintes são muito simples e o seu funcionamento pode ser esclarecido através da leitura da sua página do manual, através de **man**`comando`.

strings: Procura num ficheiro sequências de caracteres que possam ser utilizados como cadeias de caracteres. Num ficheiro de texto equivale a imprimir todo o ficheiro, mas em ficheiros binários permite identificar possíveis sequências de caracteres.

readelf: Apresenta a informação existente num ficheiro **elf**.

od: Apresenta o conteúdo de um ficheiro, de texto ou binário, interpretando-o como uma sequência de: d2 (decimal shorts), d4 (decimal longs), c (ASCII), u2 (unsigned decimal shorts), x2 (hexadecimal shorts), o2 (octal shorts), *etc*. Por exemplo, **od -t c file**.

objdump: Apresenta o conteúdo de um ficheiro objecto, podendo selecionar as secções a ler e a arquitectura que deve ser assumida. É especialmente útil para verificar o código que efectivamente foi gerado, sem ter de entrar no *debugger* e inspeccionar a memória manual, nem cometer erros de alinhamento que podem comprometer a correcta interpretação do conteúdo.

19 gprof: analisador do desempenho de programas

A optimização de programas pelo utilizador, optimização de alto-nível, só se justifica em programas muito algorítmicos e com pouca interacção. Mesmo nestes casos, apenas uma pequena fracção do programa, tipicamente 10%, é responsável por quase todo o tempo gasto, tipicamente 90%. Desta forma, é de extrema importância saber onde é que o programa está efectivamente a perder o seu tempo. Notar que em muitos casos o tempo é gasto pelo sistema operativo, nomeadamente em operações sobre ficheiros (aberturas, leituras e escritas), sendo impossível melhorar significativamente o seu desempenho (sem mudar o sistema operativo...).

O comando **gprof** (designado **prof** na maioria dos sistemas UNIX) permite contabilizar o número de chamadas a cada rotina e o tempo médio e total gasto por cada uma delas. É também apresentada uma listagem com os tempos totais acumulados, ordenada da rotina que gastou mais tempo para a que gastou menos tempo. Assim, o utilizador tem a possibilidade de dedicar o seu esforço nas rotinas mais preponderantes em termos de tempo de execução. Notar que o tempo de execução depende da simulação efectuada, isto é, da dimensão ou número de ficheiros ou outros parâmetros usados. Ou seja, os tempos e a respectiva ordenação do tempo gasto pelas rotinas pode variar de caso para caso.

De referir, que alterações locais são normalmente resolvidas pelos optimizadores de código do compilador e são, em geral, muito específicas do processador em questão. Assim, exceptuando o interior de ciclos ou funções que executem muitas vezes, as optimizações de alto-nível são em geral determinantes quando são algorítmicas. Isto é, quando é possível substituir um algoritmo por outro mais eficiente, por exemplo, um *bubble-sort* por um *quick-sort*.

Para poder utilizar o **gprof** os programas necessitam ser compilados com a opção **-p**. O programa é executado normalmente, produzindo um ficheiro de monitorização, designado **gmon.out**. O **gprof** é então executado, indicando o nome do programa executado e o nome do ficheiro de monitorização, caso este último tenha entretanto sido renomeado. O **gprof** produz a tabela acima referida no terminal.

20 strace: interceptor de chamadas ao sistema operativo

Este programa é apresentado apenas como curiosidade, sendo a sua utilidade nesta cadeira reduzida.

Um interceptor de chamadas ao sistema, e de outro tipo de comunicação como interrupções, permite acompanhar a execução do programa, não linha a linha como um *debugger*, mas através da sua interacção com o mundo exterior. Este tipo de ferramenta é muito menos invasiva que um *debugger*. De facto, apenas a duração das chamadas ao sistema fica um pouco mais demorada, mas o mesmo aconteceria se o sistema se encontrasse um pouco mais carregado. Este programa permite apenas a observabilidade, não a controlabilidade, do executável em análise, mas permite obter muita informação útil.

De entre a informação útil que se obtém inclui-se os ficheiros que o programa tenta abrir, sendo indicado se tem ou não sucesso nessa operação. Notar que este tipo de informação é de pouca utilidade quando dispomos do programa em código fonte, mas especialmente útil se estivermos a fazer testes de caixa preta, ou seja, sem conhecer o conteúdo da ferramenta.

É possível escolher uma chamada ao sistema específica, por exemplo **open**,com a opção `-eopen`, sendo apenas impressas essas chamadas ao sistema com os seus argumentos e valor de retorno.

21 time: contabilizador dos recursos de execução

O programa **time** é muito simples, basta colocar **time** antes do comando a contabilizar. Por exemplo,

```
prompt$ time find $HOME -name "[Mm]akefile" -print
(resultado comando 'find')
0.210u 0.460s 0:35.38 1.8%      0+0k 0+0io 154pf+0w
prompt $
```

A contabilização é indicada após o fim da execução do comando e antes de retornar ao editor de comandos e apresenta, da esquerda para a direita:

tempo de utilizador: tempo gasto pelo programa a executar. Este tempo não inclui o tempo gasto pelo sistema operativo para executar pedidos do programa, como abrir ficheiros.

tempo de sistema: tempo gasto pelo sistema operativo em nome do programa. Isto é, o tempo real (não virtual) em que o sistema operativo esteve a executar os pedidos do programa.

tempo virtual de execução: o tempo que o utilizador aguardou pela execução do programa. Este tempo depende da fatia de tempo que foi atribuída ao programa. Este tempo corresponde à soma dos dois tempos anteriores se, numa situação ideal, o processador só serviu este programa e nenhum outro, incluindo tarefas de rotina do sistema operativo.

fatia de tempo: indica, em média, a percentagem de ocupação do processador pelo programa. Um valor de 100% representa a ocupação total (situação ideal). O valor será tanto mais baixo quanto maior for a carga da máquina.

utilização da memória: em valores médios durante a execução.

interacção com o disco: em termos de entradas e saídas. Notar que os ficheiros podem estar em *cache* não havendo tráfego com disco.

paginação: número de faltas de página (*page faults*) realizadas pelo processo e número de vezes que o processo foi retirado de memória (*swapped out*).

Notar que o comando permite, activando as devidas opções, fornecer informação mais específica sobre a execução do processo do ponto de vista do sistema operativo.

22 Os dez mandamentos do programador

por *Henry Spencer*[2]

1. Tu verificarás todos os avisos (*warnings*) que te sejam indicados pelo compilador e corrigi-lo-ás com o maior cuidado, pois certamente que a sua infinita percepção e capacidade de análise excedem a tua.

2. Tu não seguirás ponteiros nulos, pois o caos instalar-se-á e no fim só a loucura te espera.

3. Tu verificarás todos os parâmetros que a tua rotina recebe, mesmo que estejas convencido que tal não é necessário, pois eles vingar-se-ão cruelmente em ti quando menos esperas.

4. Se os teus ficheiros de declarações não incluirem os tipos dos parâmetros e de retorno das tuas funções globais, tu as declararás com o maior cuidado, pois se não o fizeres as maiores pragas cairão sobre o teu programa.

5. Tu verificarás os limites de todas as cadeias de caracteres (de facto, de todos os vectores), pois onde tu escreveste "foo" algum dia alguém escreverá "supercalifragilisticexpialidocious".

6. Se uma rotina afirma que pode devolver um erro, tu verificarás todos os valores devolvidos, mesmo os menos esperados, mesmo que isso triplique a dimensão do teu código e produza calos nos teus dedos, pois se tu pensas que "isto não me pode acontecer", os deuses certamente te punirão pela tua arrogância.

7. Tu estudarás as bibliotecas e firmemente te oporás a reinventá-las sem uma clara justificação, para que o teu código seja curto e legível e os teus dias agradáveis e produtivos.

8. Tu procurarás tornar claro o objectivo e simples a estrutura do teu programa, para o teu colega, usando muitos espaços, parenteses e chavetas, mesmo que disso não gostes, pois a tua criatividade é melhor utilizada a resolver problemas que a criar novos impedimentos aos outros e a ti mesmo.

9. Tu escolherás cuidadosamente os nomes dos teus identificadores, em especial dos globais, apesar desta tarefa ser fastidiosa e hárdua e os teus dias prolongarem-se sem fim, pois tu arrancarás os cabelos e enloquecerás no dia fatídico em que usares uma nova biblioteca ou mudares de sistema operativo.

[2]tradução e actualização por Pedro Reis dos Santos.

10. Tu renunciarás e abster-te-ás da vil heresia de afirmar que "Todos os computadores usam WINDOWS", e não te deixarás influenciar pelos intelectualmente fracos e pagãos que acreditam nesse bárbaro conceito, e os dias dos teus programas poderão ser longos apesar dos dias do teu computador actual poderem ser curtos.