Computação

Computação

- Computação significa: processamento automático de informação.
- É a atividade realizada pelos computadores.
- O objetivo da informática é estudar a computação e formas úteis de tirar partido dela para resolver problemas importantes.

Facetas da computação

- **Faceta interna:** a informação é codificada usando símbolos (e.g. bytes), sendo a computação uma atividade de manipulação e transformação automática de símbolos.
- **Faceta externa:** ... mas, geralmente, as computações também estabelecem um diálogo interativo com o ambiente exterior (constituído por humanos e por outras máquinas).

Automatismos e sua especificação

- Qualquer computação é realizada de acordo com regras estabelecidas antes desse processamento se iniciar. É isso que significa o processamento ser automático.
- Daí a necessidade que especificar as computações de forma rigorosa e exaustiva, prevendo todas as eventualidades.

Algoritmos, Programas e Linguagens de Programação

Algoritmo

- Um algoritmo é um conjunto de regras abstratas que determinam, passo a passo, como uma computação vai decorrer.
- Em geral um problema pode ser resolvido usando diversos algoritmos.
- Um algoritmo é independente de qualquer linguagem de programação particular. Podemos imaginá-lo como se fosse para ser executado num computador ideal com memória infinita.

A palavra "algoritmo" deriva da palavra **Algoritmi** que por sua vez corresponde à Latinização do nome do matemático Persa Muhammad ibn Musa al-Khwarizmi, nascido por volta de 780DC. Este cientista, que trabalhou quase toda a sua vida em Bagdad, deu importantes contribuições para a Álgebra, Trigonometria e Aritmética.

Muhammad ibn Musa al-Khwarizmi

Programa

- Um programa é a expressão concreta dum algoritmo.
- Um programa implementa um algoritmo numa linguagem de programação concreta.

Linguagem de programação

- É uma notação para escrever programas.

Exemplo de algoritmo

Algoritmo de Euclides - mdc(m,n)

- Usar dois contadores x e y e inicializar x com m e y com n.
- Se x > y então x recebe x - y
- Se x < y então y recebe y - x
- Repetir os dois passos anteriores até que os valores de x e y fiquem iguais. Quando isso acontecer, esse é o resultado final.

Implementação em C do algoritmo anterior

```
#include <stdio.h>

int main() /* Implementação do algoritmo de Euclides */
{
    int m, n, x, y ;
    printf(">> ") ;
    scanf("%d %d", &m, &n) ;
    x = m ;
    y = n ;
    do {
        if( x > y ) x = x - y ;
        else if( x < y ) y = y - x ;
    } while( x != y ) ;
    printf("mdc(%d,%d) = %d\n", m, n, x) ;
    return 0 ;
}
```

Antiguidade dos Algoritmos

- Os algoritmos mais antigos que se conhecem devem-se aos Babilónios (3000AC-1500AC). Os seus livros de Matemática eram acima de tudo receitas sobre como efetuar os cálculos para resolver determinados problemas. Esses algoritmos eram para executar "à mão".
- A principal motivação para se usarem máquinas para executar algoritmos é a grande velocidade de execução que elas permitem.

Duas questões importantes sobre computação e linguagens

- Existem limites para a computação?
  - Sim, realmente existem problemas com solução para os quais não se consegue inventar qualquer algoritmo que descubra essa solução.
- Existe alguma linguagem de programação que permita implementar todos os algoritmos que se possam imaginar?
  - Sim, qualquer linguagem "normal" permite isso. Do ponto de vista do poder computacional, todas as linguagens normais são equivalentes.

Paradigmas de Programação

Já sabemos que computação significa processamento automático de informação, mas esta noção é demasiado vaga. Podemos interrogar-nos sobre quais os mecanismos concretos através dos quais esse processamento se efetua.

Na verdade, uma linguagem não pode deixar de se comprometer com algum conjunto mecanismos primitivos para processar informação. Ao efetuar essa escolha, a linguagem adere a um paradigma de programação particular.

É surpreendente a grande variedade de paradigma de programação que têm sido inventados.

Exemplos de paradigmas de programação e respetivos conceitos de base

- **Paradigma imperativo**
  - Conceitos: estado, atribuição, sequenciação
  - Linguagens: Basic, Pascal, C, Assembler.
- **Paradigma funcional**
  - Conceitos: função, aplicação, avaliação
  - Linguagens: Lisp, ML, OCaml, Haskell.
- **Paradigma lógico**
  - Conceitos: relação, dedução
  - Linguagens: Prolog.
- **Paradigma orientado pelos objetos**
  - Conceitos: objetos, mensagem
  - Linguagens: C++, Java, Eiffel.
- **Paradigma concorrente**
  - Conceitos: processo, comunicação (síncrona ou assíncrona)
  - Linguagens: Occam, Ada, Java.

Aspetos práticos

- Cada paradigma de programação determina uma forma particular de abordar os problemas e de formular as respetivas soluções. De facto, diferentes paradigmas de programação representam muitas vezes visões irreconciliáveis do processo de resolução de problemas.
- O grau de sucesso dum programador depende em parte da coleção de paradigmas que domina e da sua arte em escolher o modelo conceptual (paradigma) mais indicado para analisar e resolver cada problema.

Uma linguagem de programação pode combinar mais do que um paradigma

- C++ --- Paradigma imperativo + paradigma orientado pelos objetos.
- Java --- Paradigma imperativo + paradigma orientado pelos objetos + paradigma concorrente.
- OCaml --- Paradigma funcional + Paradigma imperativo + paradigma orientado pelos objetos.
- Ada --- Paradigma imperativo + paradigma concorrente.

Paradigma imperativo *versus* paradigma funcional

Paradigma imperativo

- A computação baseia-se na existência de um "estado" (conjunto de variáveis mutáveis) que vai evoluindo ao longo do tempo.
- O estado é modificado usando um "comando de atribuição" e existe um "operador de sequenciação".
- Linguagens: Basic, Pascal, C, etc., mas o Java também tem uma base imperativa, visto suportar variáveis mutáveis.

Paradigma funcional

- A computação consiste na avaliação de "expressões" nas quais podem ocorrer chamadas de funções.
- O objetivo da avaliação duma expressão é a produção dum "resultado" ou "resposta".
- Não há "atribuição" nem variáveis cujo valor possa ser alterado.
- Cada função limita-se a produzir um resultado a partir dos seus argumentos.
- Linguagens: Lisp, ML, Haskell, Scheme.

Plataformas computacionais

É demasiado simplista dizer que os programas correm sobre hardware. É mais rigoroso dizer que os programas correm sobre uma **plataforma computacional**.

Uma plataforma é constituída pelos seguintes elementos:

- **Hardware** - Inclui: processador, memória, canais de entrada e saída;
- **Sistema operativo** - Software que faz a gestão dos recursos do computador e que fornece aos programas uma interface para aceder a esses recursos.
- **Bibliotecas** - Sã componentes de software opcionais que fornecem aos programas interfaces especializadas e de mais alto-nível para o sistema operativo.

Exemplos de plataformas:

- Intel-32/Microsoft Windows
- Intel-32/Mac OS
- PowerPC/Mac OS
- Intel-32/Linux
- Mips/Linux
- Sparc/Linux
- Java (é independente do hardware)
- .Net (é independente do hardware)

Implementação de linguagens de programação: Compilação e Interpretação

A generalidade das linguagens de programação suporta conceitos e abstrações mais sofisticados do que os mecanismos suportados pelas plataformas computacionais usuais. Assim, para conseguir executar numa plataforma computacional programas escritos numa linguagem de alto-nível é preciso recorrer a uma das duas seguintes técnicas, ou a uma mistura das duas:

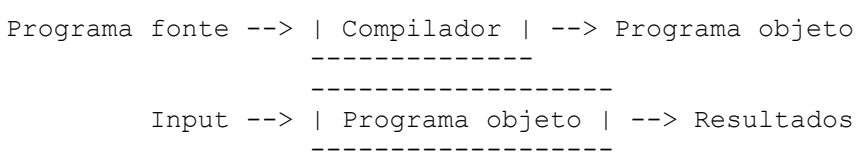
- Compilação
- Interpretação

Compilação

Um **compilador** é um programa tradutor com as seguintes características:

- Converte programas duma linguagem de programação de alto-nível para programas equivalentes escritos numa linguagem de mais baixo nível.
- A linguagem de mais baixo nível é geralmente linguagem-máquina, o que já permite que cada programa possa ser executado diretamente na plataforma computacional.
- O programa executável, gerado pelo compilador, pode ser corrido as vezes que se quiser, sem ser preciso voltar a usar o compilador.

O seguinte diagrama descreve a situação:



Alguns exemplos. Na plataforma Intel/Linux usada nas aulas práticas estão disponíveis os seguintes compiladores:

- Compilador de OCaml chamado **ocamlopt**.
- Compilador de C/C++ chamado **gcc**.
- Compilador de Java chamado **gcj**.

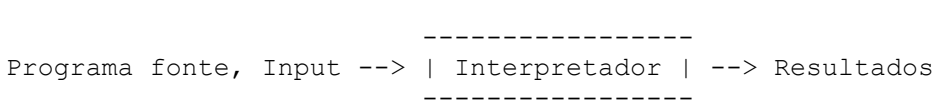
Todos estes compiladores geram código nativo.

Interpretação

Um **interpretador** é um programa "executor" com as seguintes características:

- Executa diretamente o programa fonte.
- Para correr o programa novamente é necessário voltar a usar o interpretador.

O seguinte diagrama descreve a situação:



Alguns exemplos. Na plataforma usada nas aulas práticas está pelo menos disponível um interpretador:

- Um interpretador de OCaml chamado **ocaml**.

Vantagens/Desvantagens

Vantagens de usar um compilador:

- Grande velocidade de execução (depois da compilação ter terminado).
- Os programas podem ser distribuídos sob a forma executável, sem ser necessário fornecer uma cópia do compilador.

Os ganhos de eficiência dos compiladores devem-se à seguinte razão:

- O compilador toma muitas decisões ao processar o programa fonte (considere por exemplo o acesso a variáveis). O ganho está no facto dessas decisões já não terem de ser tomadas novamente durante a execução de programa. Além disso, se o programa contiver ciclos ou funções recursivas, esta questão torna-se ainda mais importante - um interpretador estaria sempre, repetidamente, a tomar as mesmas decisões.

Vantagens de usar um interpretador:

- Maior rapidez na escrita e teste de programas, pois podem correr-se os programas diretamente, sem esperar pela compilação.
- Melhores diagnósticos de erros.
- Maior potencial para escrever *source-level debuggers* de qualidade.
- Maior facilidade em suportar linguagem reflexivas, ou seja linguagens em que os programas se podem observar a eles mesmos e auto-modificarem-se durante a execução. Exemplos de linguagens destas: Prolog, Lisp.

Técnicas de implementação mistas

São bem claras as diferenças entre os conceitos de compilação e interpretação. Contudo a implementação de muitas linguagens de programação acaba por baseada numa mistura de compilação e de interpretação.

Vejamos três situações diferentes:

Coexistência de código compilado e de código interpretado

Algumas implementações da linguagem Prolog permitem a coexistência de código compilado com código interpretado.

Predicados declarados como estáticos (i.e. que não podem ser alterados em tempo de execução) são normalmente compilados; predicados declarados como dinâmicos (i.e. que podem ser modificados em tempo de execução) são obrigatoriamente interpretados.

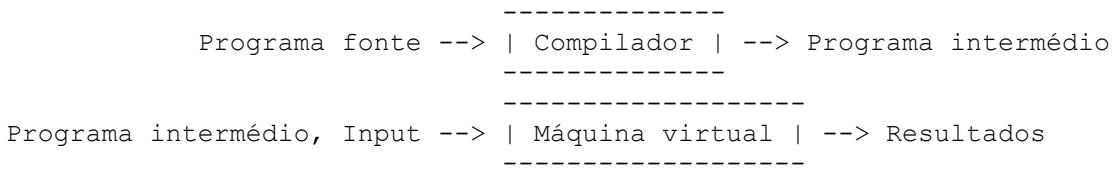
A vantagem desta técnica é que permite obter velocidade de execução nas partes estáticas do programa, e a flexibilidade necessária nas partes dinamicamente modificáveis do programa.

Máquinas virtuais

Outra forma de implementação mista de linguagens de programação envolve a invenção duma linguagem intermédia para a qual se escreve um interpretador. O termo **máquina virtual** é muitas vezes usado, tanto para designar a linguagem intermédia, como o seu interpretador.

Agora basta escrever um compilador para traduzir programas da linguagem de alto-nível para código intermédio. O resultado da tradução pode depois ser executado na máquina virtual.

O seguinte diagrama descreve a situação:



É muito importante que a máquina virtual seja criada com três objetivos em mente:

- As suas particularidades devem facilitar a tradução dos programas fonte para código intermédio.
- Deve ser simples escrever um interpretador razoavelmente eficiente para ela.
- Deve favorecer a compacidade do código intermédio.

Vantagens da técnica da máquina virtual:

- O compilador torna-se mais simples de escrever.
- A velocidade de execução é mediana: melhor do que num interpretador puro mas pior do que num compilador nativo.
- Os programas intermédios podem ser executados em qualquer plataforma onde a máquina virtual esteja disponível.
- Os programas intermédios são geralmente bastante compactos pelo que podem ser transferidos através da WEB mais rapidamente.

Eis alguns exemplos de implementações baseadas em máquinas virtuais:

- Compilador de OCaml chamado **ocamlc** que gera código para uma máquina virtual chamada **CAML** (Categorical Abstract Machine Language).

- Compilador de Java chamado **javac** que gera código para uma máquina virtual chamada **JVM** (Java Virtual Machine).
- Compilador de C# chamado **mcs** que gera código para uma máquina virtual chamada **CLR** (Common Language Runtime).

Máquinas virtuais just-in-time

Quando é muito importante tornar a implementação duma máquina virtual tão rápida quanto possível, torna-se necessário implementar o respetivo interpretador usando a técnica **just-in-time** (JIT), também conhecida por **tradução dinâmica**.

A ideia da técnica é simples, embora a implementação seja complicada de fazer. Durante a execução do programa intermédio, este vai sendo dinamicamente traduzido em código-máquina que é imediatamente executado. A tradução dinâmica é aplicada a unidades a unidades de código, tais como métodos ou função, e a implementação gere uma cache de unidades de código já processadas. Ao fim de algum tempo de execução, quando a maior parte do código já foi traduzido, consegue-se atingir uma velocidade de execução que ronda os 70% da velocidade de execução dum programa compilado nativamente. Alguns exemplos de máquinas virtuais para as quais existem versões JIT:

- **JVM** - Java Virtual Machine.
- **CLR** - Common Language Runtime.

Ponto de vista externo e interno

As técnicas de implementação mistas, fazem esbater a fronteira entre as noções de compilação e de interpretação. Por vezes, para caracterizar a implementação duma linguagem com rigor, temos de considerar dois pontos de vista:

- Um **ponto de vista externo** que considera a implementação dos compiladores e interpretadores como caixas negras.
- Um **ponto de vista interno** que se interessa pela implementação interna dos compiladores e interpretadores.

Para discutir os dois pontos de vista, vamos considerar uma máquina virtual just-in-time, por exemplo a JVM:

- Do ponto de vista externo, uma máquina virtual just-in-time é um simples interpretador, pois constitui um programa que permite executar diretamente programas (neste caso escritos na chamada linguagem intermédia).
- Do ponto de vista interno, uma máquina virtual just-in-time é um complexo sistema que intercala a execução dum compilador nativo com a execução de porções de código máquina diretamente pelo hardware.

Consideremos agora o interpretador de OCaml **ocaml**, muito usado nas nossas aulas práticas.

- Do ponto de vista externo, trata-se dum interpretador, pois é visível que ele permite executar diretamente programas escritos em OCaml.
- No entanto, consultando a documentação, aprende-se que, do ponto de vista interno, é um **compilador/interpretador em duas fases**, pois ele compila internamente os programas fonte para código da máquina virtual CAML, sendo o código gerado internamente imediatamente executado pela máquina virtual.

Consideremos agora a arquitetura de hardware Intel-32. Claramente um processador da Intel implementa uma linguagem que se chama linguagem máquina. Será que deve ser encarado como um compilador ou um interpretador.

- Do ponto de vista externo, trata-se dum interpretador, pois é notório que ele permite executar diretamente programas escritos em linguagem máquina.
- No entanto, o processador compila internamente a linguagem máquina para uma linguagem de mais baixo nível chamada de micro-código. Depois, é o micro-código que efetua o trabalho útil do programa, ao ser

executado por um interpretador implementado em hardware. São as instruções de micro-código que atuam sobre os circuitos físicos do processador.

Níveis de interpretação

Vejamos os três níveis de interpretação envolvidos na execução dum programa em Java sobre uma JVM implementada em hardware da Intel:

1. Interpretador de micro-código implementado dentro do processador físico.
2. Interpretador de código-máquina - é o processador físico.
3. Interpretador da JVM - implementado por software.

Podem existir níveis de interpretação ainda menos elevados:

- A arquitetura da Motorola MC68000 inclui nano-código, a mais baixo nível do que o micro-código.

E podem existir níveis de interpretação ainda mais elevados:

- Uma forma popular de implementar a linguagem Prolog é através da escrita dum interpretador em Java (porque isso facilita a interoperabilidade das duas linguagens).
- O nosso CLIP está implementado numa linguagem chamada Compass. Esta linguagem está implementada como um interpretador sobre a linguagem em Prolog.

Detalhes da implementação de interpretadores

Já sabemos que um interpretador é um programa que executa diretamente programas fonte.

Código interpretado é geralmente mais lento (por vezes, 10 vezes mais lento!) do que código compilado. Algumas razões:

- O interpretador tem de analisar cada parte do programa antes de a executar, e tem de a voltar a analisar sempre que a quiser voltar a executar (por exemplo, num ciclo). Por seu lado o compilador executa o código diretamente, sem perdas de tempo, pois a analise do programa já foi efetuada anteriormente, em tempo de compilação.
- Outra razão pela qual um interpretador é mais lento prende-se com o acesso às variáveis do programa. Um interpretador gasta imenso tempo a mapear dinamicamente os nomes das variáveis do programa em células de memória (geralmente com a ajuda duma hash table). Por seu lado um compilador associa endereços fixos às variáveis do programa, pelo que o acesso às variáveis é praticamente instantâneo em tempo de execução.

Um aspeto importante da interpretação é a forma como o interpretador lida com o texto do programa fonte. Há varias técnicas que têm sido usadas historicamente.

Interpretação direta

Nos interpretadores mais rudimentares, o texto é carregado em memória e é usado diretamente como matéria prima para a execução dos programas.

Os primeiros interpretadores de Basic, em meados dos anos 60, foram feitos desta forma. O manual da linguagem recomendava que não se escrevessem muitos comentários para os programas correrem mais rapidamente. Com efeito,

os mesmos comentários eram lidos (e ignorados) sempre que uma dada parte do programa era executada (por exemplo, num ciclo).

Também convinha escolher variáveis com nomes curtos, para acelerar a execução dos programas.

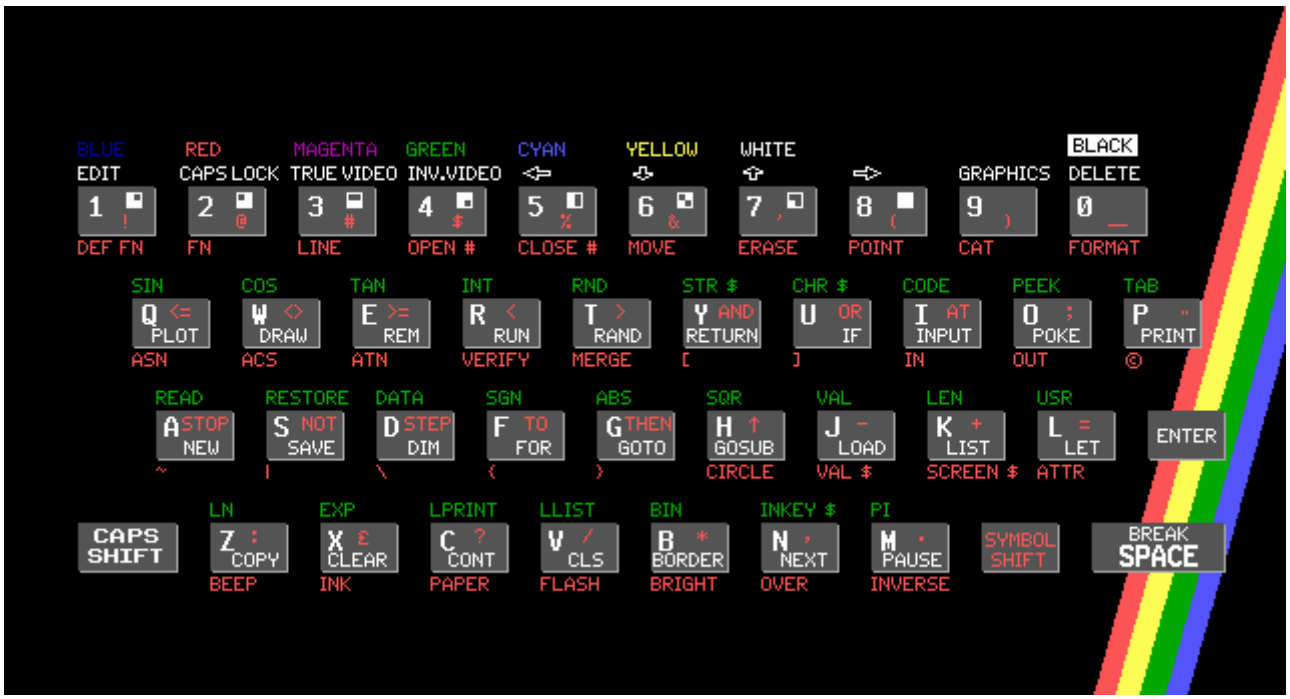
Preprocessamento com identificação de tokens

A maioria dos interpretadores faz algum processamento do ficheiro fonte no momento do carregamento. Tipicamente, os comentários e espaços em branco são removidos, e todas as sequências de caracteres significativas são agrupadas em **tokens**, ou seja em símbolos que representam palavras reservadas, números, nomes de variáveis, etc.

Durante a interpretação, a matéria prima para a execução dos programas já não são sequências de caracteres, mas sim sequências de tokens. Consegue-se assim obter maior eficiência

O Basic do ZX Spectrum

Uma variante curiosa desta técnica foi usada na implementação do Basic do ZX Spectrum. O ZX Spectrum foi um computador pessoal muito popular na Europa durante os anos 80. Os programas podiam ser carregados a partir duma cassete áudio ou ser metidos à mão. Quando os programas eram metidos à mão, o editor de texto obrigava o utilizador a introduzir as palavras reservadas como tokens, o que era possível devido ao curioso teclado.



Preprocessamento com identificação de árvore sintática

Muitos interpretadores modernos efetuam um processamento bastante sofisticado do ficheiro fonte, no momento do carregamento. Além de identificarem todos os tokens, também identificam a estrutura sintática do programa de entrada e constroem em memória a correspondente **árvore sintática** (*parse tree*).

Durante a interpretação, a matéria prima para a execução dos programas é a árvore sintática. Consegue-se assim obter ainda maior eficiência.

Interpretador ou compilador?

Note que a identificação dos tokens e a construção da árvore sintática correspondem às duas primeiras fases de processamento dos compiladores tradicionais. Realmente, para aumentar a eficiência dum interpretador é preciso integrar nele alguma funcionalidade típica dos compiladores. Prosseguindo nesta linha de integrar mais e mais funcionalidade dos compiladores num interpretador, em breve se chega à técnica mista das máquinas virtuais,

estudada da aula anterior. Do ponto de vista interno, já deixámos de ter um interpretador, e passámos a ter um sistema misto constituído por um compilador que gera código intermédio e um interpretador duma máquina virtual.

Detalhes da implementação de compiladores

Num compilador típico, a compilação evolui ao longo duma série de fases. Cada fase descobre informação que é necessária nas fases seguintes, ou então transforma o programa numa forma que é requerida pela fase seguinte.

1. Leitura de caracteres do ficheiro fonte

   sequência de caracteres ->

2. Análise lexical (scanner)

   -> sequência de tokens ->

3. Análise sintática (parser)

   -> árvore de parsing ->

4. Análise semântica

   -> árvore Abstract ->

5. Melhoramento e completação da árvore Abstract

   -> árvore Abstract melhorada ->

6. Geração de código intermédio

   -> código intermédio ->

7. Otimização de código intermédio

   -> código intermédio otimizado ->

8. Geração de código máquina

   -> código máquina executável

Ligação

Para simplificar as discussões anteriores relativas a compiladores, temos vindo a omitir qualquer referência à questão da **ligação de ficheiros objeto**. Vamos tratar agora dessa questão que não pode ser ignorada quando se discute o tema da implementação de linguagens de programação.

Código fonte e código objeto

Um programa é muitas vezes constituído por diversos **ficheiros fonte**, sendo o conjunto habitualmente designado por **código fonte** do programa. Exemplos:

```
a.c b.c    --- código fonte dum programa em C
a.ml b.ml  --- código fonte dum programa em ML
```

Quando o compilador processa o código fonte dum programa, o compilador gera um **ficheiro objeto** distinto por cada ficheiro fonte. Exemplos:

```
gcc -c a.c b.c      --> a.o b.o
ocamlc -c a.ml b.ml --> a.cmo b.cmo
```

O conjunto dos ficheiros objeto chama-se **código objeto**.

Ligador

Depois de compilado o código fonte, para se obter o programa executável é preciso usar um programa **ligador** que junta os diversos ficheiros objeto num único programa executável. Na altura de ligar um programa é preciso também indicar quais são as bibliotecas (arquivos de ficheiros objetos predefinidos) que interessa juntar ao programa.

Em Linux, o ligador chama-se `ld` e pode ser invocado diretamente pelo utilizador, se tal for desejado. Mas geralmente é mais prático invocar indiretamente o ligador através do comando de compilação. Em todo o caso, veja o que diz o início do manual do comando `ld`:

```
$ man ld

LD(1)                       GNU Development Tools                       LD(1)

NAME
       ld - The GNU linker

SYNOPSIS
       ld [options] objfile ...

DESCRIPTION
       ld  combines a number of object and archive files, relocates their data
       and ties up symbol references. Usually the last  step  in  compiling  a
       program is to run ld.
```

No Linux usa-se a opção "-o" na linha de comando dos compiladores serve para invocar indiretamente o ligador como último passo da compilação. Exemplos:

```
ocaml -o prog a.cmo b.cmo  --> prog
gcc -o prog a.o b.o        --> prog
```

Outra missão do ligador é resolver as referências cruzadas de nomes globais que podem ocorrem nos diversos ficheiros object. Para perceber o que está em causa vamos observar a tabela de símbolos e informação de relocação que é guardada dentro do ficheiro object correspondente ao seguinte ficheiro fonte "a.c":

```
/* File a.c */
/* This is a C source file */

#include <stdio.h>

extern int g(int) ;
extern int global_e ;

int global_x, global_y, global_z ;
int global_a = 123, global_b = 123, global_c = 124 ;

int f(int x)
{
    return x + global_a ;
}

int main()
{
    printf("%d", f(2) + g(3)) ;
    return 0 ;
}
```

O comando `nm` do Linux mostra os símbolos e informação de relocação de ficheiros-objeto individuais. Aplicando `nm` ao ficheiro object "a.o", obtém-se o seguinte:

```
amd@sunra2:z$ nm a.o

00000000 T f
         U g
00000000 D global_a
00000004 D global_b
00000008 D global_c
00000004 C global_x
00000004 C global_y
00000004 C global_z
0000000d T main
         U printf
```

Carregamento

O **carregador** é a componente do núcleo do sistema operativo que trata de carregar em memória os programas para depois os executar.

No Linux, `execve` é o nome da chamada ao sistema que permite usar o carregador. Leia o que diz o início do manual do comando `execve`:

```
$ man execve

EXECVE(2)                  Linux Programmer's Manual                 EXECVE(2)

NAME
       execve - execute program

SYNOPSIS
       #include <unistd.h>

       int execve(const char *filename, char *const argv[],
                  char *const envp[]);

DESCRIPTION
       execve() executes the program pointed to by filename.  filename must be
       either a binary executable, or a script starting with  a  line  of  the
       form:

            #! interpreter [optional-arg]

       For details of the latter case, see "Interpreter scripts" below.

       argv  is  an array of argument strings passed to the new program.  envp
       is an array of strings, conventionally of the form key=value, which are
       passed  as  environment to the new program.  Both argv and envp must be
       terminated by a null pointer.  The argument vector and environment  can
       be  accessed  by the called program's main function, when it is defined
       as:

            int main(int argc, char *argv[], char *envp[]).

       execve() does not return on success, and the text, data, bss, and stack
       of the calling process are overwritten by that of the program loaded.
```

Em algumas plataformas, e.g. System/360 da IBM, o carregador tem a tarefa de efetuar relocação de endereços, porque o hardware só suporta endereçamento absoluto.

Nas plataformas modernas está geralmente disponível um segundo carregador - um **carregador dinâmico** - que permite carregar bibliotecas dinâmicas a meio da execução dum programa. Em Windows as bibliotecas dinâmicas são guardadas em ficheiros com extensão "dll". No Linux a extensão das bibliotecas dinâmicas é "so".

Ligações (bindings)

Uma ligação consiste numa associação entre duas entidades. As **entidades** podem ser nomes, localizações de memória, tipos, objetos, etc.

Uma ligação é sempre unidirecional e associa uma entidade mais simples a outra entidade mais complexa. O caso das duas entidades terem complexidade semelhante também ocorre por vezes. As ligações facilitam o trabalho do programador pois ele pode usar a entidade mais simples para representar a entidade mais complexa.

Por exemplo:

- Ligação dum nome a um valor fixo, que pode ver um inteiro, uma função, etc. (constante).
- Ligação dum nome a uma localização de memória (variável mutável).
- Ligação duma localização de memória a outra localização de memória (apontador).
- Ligação dinâmica dum nome a um valor (variável mutável).

Diferentes ligações duma entidade

Para uma mesma entidade, digamos um nome, podem estar definidas diferentes ligações. Por exemplo, em C o nome duma variável global tem as seguintes ligações:

- Ligação a um tipo fixo (estabelecida em tempo de compilação).
- Ligação a uma localização de memória fixa (estabelecida em tempo de carregamento).
- Ligação a um valor (estabelecida em tempo de execução).

Semântica e ligações

A semântica de qualquer linguagem de programação é determinada de forma essencial a partir:

- O conjunto de formas de ligação que se podem estabelecer;
- Do momento em que essas ligações se estabelecem.

Momento da ligação

Eis diversos momentos em que uma ligação pode ser estabelecida:

- Tempo de conceção da linguagem. Exemplo em OCaml: operador "+".
- Tempo de implementação da linguagem Exemplo em OCaml: constante "max_int".
- Tempo de compilação. Exemplo em C: constante "#define A 2"
- Tempo de ligação. Exemplo em C: função externa "extern int f(void) ;".
- Tempo de carregamento. Exemplo em C: variável global "int x ;".
- Tempo de execução. Exemplo em OCaml: constante "let a = 2 + x in ...".

Em geral, quanto mais tarde se estabelecem as ligações mais flexível é a linguagem.

Em geral, quanto mais cedo se estabelecem as ligações, mais rápida é a linguagem.

Por exemplo, é mais eficiente invocar um procedimento em C (ligação em tempo de ligação) do que enviar uma mensagem para um objeto em C++ (ligação em tempo de execução). Em todo o caso, justifica-se o preço a pagar em C++: é o facto da ligação entre a mensagem e o método ser estabelecida muito tarde (late binding) que permite o paradigma orientado pelos objetos.

Classificação das ligações

Classificação das ligações em função do momento de ligação:

- **Estáticas**: efetuadas antes da execução do programa.
- **Semi-dinâmicas**: efetuadas em tempo de execução mas determinadas em grande parte antes de o programa começar a correr.
- **Dinâmicas**: efetuadas completamente em tempo de execução.

Exemplos de ligações estáticas:

- nome->valor: constantes "#define" em C;
- nome->tipo: tipos, variáveis e constantes em ML e C; (ML->inferência de tipos)
- nome->localização: variáveis globais em C.

Exemplos de ligações semi-dinâmicas:

- nome->localização: variáveis locais em C e constantes locais em ML;
- nome->método: mensagens em C++ e Java.

Discussão do caso das variáveis locais em C, um caso de ligação semi-dinâmica: Essas variáveis residem na chamada *pilha de execução* do C, e as respetivas localizações de memória são determinadas em tempo de execução. No entanto, é em tempo de compilação que se determina o *offset* dentro do registo de cativação da função onde a variável reside. As constantes locais do ML "let a = 2 + x in ..." também são implementadas da mesma forma.

Exemplos de ligações dinâmicas:

- localização->valor: variáveis mutáveis em C;
- localização->localização: apontadores em C;
- nome->valor: variáveis mutáveis em qualquer linguagem;
- nome->tipo: variáveis em Smalltalk e Ruby.

Discussão do caso das variáveis em Smalltalk e Ruby: Estas variáveis não têm tipos estáticos associados. Aceitam valores de qualquer tipo e portanto, sempre que há uma atribuição muda o valor e o tipo dessas variáveis.

Tempo de vida duma ligação

O tempo de vida duma ligação é o período de tempo da execução dum programa durante a qual essa ligação persiste. As ligações estáticas persistem durante a execução de topo o programa. As ligações semi-dinâmicas e dinâmicas persistem geralmente apenas durante parte da execução do programa.

Ambiente (conjunto de ligações para nomes)

O conceito de ambiente tem a ver com um tipo particular de ligações: as ligações de nomes.

Chama-se **ambiente** a um conjunto de ligações que associam nomes (identificadores) a entidades. Matematicamente um ambiente é uma função de nomes para entidades, ou seja uma função com o seguinte tipo:

```
Nomes -> Entidades
```

O seguinte pequeno programa em ML define quatro ambientes diferentes, consoante o ponto do programa que for considerado:

```
let f x = x + 1 ;;
let rec g x = f x + 1 ;;
```

1. O ambiente antes da função f inclui apenas as ligações dos nomes predefinidos na linguagem ML. São exemplos desses nomes: max, "^", "+", int, float.

2. O ambiente no interior da função f inclui as ligações dos nomes predefinidos, mais a a ligação do nome "x" que representa o argumento de f. Repare que o nome "f" não tem ligação dentro da função f, porque esta função não é recursiva.
3. O ambiente no interior da função g inclui as ligações dos nomes predefinidos, mais a ligação do nome "x" que representa o argumento de g, mais os nomes "f" e "g" que representam funções.
4. O ambiente após a função g inclui as ligações dos nomes predefinidos, mais os nomes "f" e "g" que representam funções.

Âmbito (escopo) duma ligação

Âmbito (escopo) duma ligação é a região do programa na qual esse nome tem os atributos estabelecidos pela declaração que introduz a ligação.

Na maior parte das linguagens de programação, o âmbito duma ligação é determinado pela estrutura sintática do programa (ver "Escopo estático", mais abaixo).

Há exemplos de âmbitos na secção "Blocos", um pouco mais abaixo.

Construções definidoras de âmbitos

A generalidade das linguagens de programação possuem construções que têm implicações nos âmbitos das ligações que se estabelecem nos programas. Eis alguns exemplos dessas construções:

- Blocos (C, C++, Java, Pascal, Ada)
- Let-in (OCaml)
- Módulos (Módula-2, OCaml)
- Classes (C++, Java, C#)
- Packages (Java)
- Namespaces (C++, C#)
- Espaço global (duma forma ou de outra, todas as linguagens dispõem dum espaço global de nomes)

Blocos

As linguagens que descendem do antigo Algol-60 possuem uma construção sintética chamada **bloco**. Um bloco tem duas utilidades:

- Serve para introduzir um novo ambiente no qual todas as novas ligações têm aproximadamente o mesmo âmbito. (Esta é a parte que nos interessa aqui.)
- Serve para agregar uma sequência de comandos num comando naquilo que, tecnicamente, é um comando único, dito composto.

O seguinte bloco, em C, determina ligações para os nomes `i`, `j` e `k`. Todas essas ligações têm como âmbito aproximadamente todo o interior do bloco. "Aproximadamente", porque realmente o âmbito de `j` é ligeiramente mais pequeno do que o âmbito de `i`, e o âmbito de `k` é ligeiramente mais pequeno do que o âmbito de `j`. Onde é que começa exatamente o âmbito de cada uma das três ligações?

```
{
    int i = 0
    int j = i + 2
    int k = i + j ;
    printf("%d %d %d\n", i, j, k) ;
}
```

O seguinte exemplo, também em C, é mais interessante e ilustra um **bloco aninhado** dentro de outro bloco.

```
{
    int i ;
    int b = 5 ;
```

```
        i = a + b ;
        {
            int i = 0
            int j = i + 2
            int k = i + j ;
            printf("%d %d %d\n", i, j, k) ;
        }
        printf("%d %d\n", i, b) ;
    }
```

Repare que o âmbito da variável `b`, introduzida no bloco exterior, abrange aproximadamente todo o bloco externo, o que inclui o bloco interno. No entanto o âmbito da variável `i` introduzida no bloco exterior abrange o bloco exterior *menos* o bloco interior, porque a variável `i` é redefinida no bloco interior.

Este exemplo mostra que numa linguagem onde as construções definidores de âmbitos podem ser aninhadas, o âmbito duma ligação pode não corresponder a uma zona contígua de programa. Por outras palavras, pode ter "buracos" âmbito duma ligação!

Resolução de nomes

Chama-se **resolução de nomes** ao processo de descoberta do significado (ou seja, da ligação) de alguns nomes num ponto do programa onde esses nomes são usados.

Escopo estático

Escopo estático é o nome da regra de resolução de nomes usada na maioria das linguagem modernas, incluindo o OCaml, C, C++ e Java.

A regra é muito simples e diz apenas o seguinte:

- Um uso dum nome refere-se sempre à ligação sintaticamente envolvente, mais próxima.

Portanto, para saber o que o significado dum nome num dado ponto do programa, basta olhar para o código "à volta" (de acordo com as regras da sintaxe da linguagem), e procurar aí a declaração mais próxima desse nome. Um caso particular: se o nome estiver definido localmente, então é a declaração local que vale para o nome em causa.

Quem aprendeu a programar numa linguagem moderna, está tão habituado a esta regra que geralmente nem se apercebe dele. Bem, nos exemplos da secção "Blocos", atrás, nós já usámos esta regra "sem dar por isso"...

Mais um exemplo, agora em ML:

```
let z = 5 in
    let f x = x + z in
        let z = 6 in
            f 0
```

Neste exemplo aparece uma utilização duma variável `z`, dentro da função `f`, que pode criar dúvidas. Será que uso do nome `z` se refere à declaração de `z` exterior, ou à declaração de `z` interior?

Como a linguagens ML usa a regra de escopo estático, a resposta correta é: o uso do nome `z` refere-se à declaração de `z` exterior (ou seja, envolvente).

Pergunta: Quando o valor da expressão do exemplo, 5 ou 6?
Resposta: 5.

Escopo dinâmico

Escopo dinâmico é o nome da regra de resolução de nomes atualmente em desuso, mas que importa conhecer. É usado em algumas versões da linguagem Lisp e também na linguagem APL, por exemplo.

A regra também é muito simples e diz apenas o seguinte:

- Um uso dum nome refere-se sempre à ligação mais recentemente estabelecida para esse nome, durante a execução do programa.

Regressemos ao exemplo:

```
let z = 5 in
    let f x = x + z in
        let z = 6 in
            f 0
```

Pergunta: Usando a regra de escopo dinâmico, qual o valor da expressão do exemplo, 5 ou 6?
Resposta: 6.

A resposta é 6 porque, repare, quando a função `f` é chamada, a ligação mais recente para `z` é que foi estabelecida na declaração de `z` interior.

Efeitos das regras

Em muitas situações, como por exemplo quando estão em causa nomes declarados localmente, as duas regras de escopo acabam por dar os mesmos resultados, ou seja, resolvem os nomes da mesma forma. Só perante situações semelhantes à do exemplo anterior é que os efeitos são diferentes.

Em rigor, os efeitos das regras só diferem quando estão em causa acessos a nomes não-locais a partir do interior de funções. Por isso as duas regras de escopo podem ser apresentadas da seguinte forma alternativa:

- **Escopo estático** - As funções são chamadas no ambiente da sua definição.
- **Escopo dinâmico** - As funções são chamadas no ambiente de quem os invoca.

Comparação

O escopo estático é usado em praticamente em todas as linguagens modernas pois faz com que a estrutura estática de um programa se aproxime do seu comportamento dinâmico. Isto simplifica imenso a compreensão dos programas.

A regra de escopo dinâmico tem ainda mais estas desvantagens:

- Não permite fazer verificação de tipos estática;
- Os programas ficam sensíveis aos nomes escolhidos para as variáveis e procedimentos.

Estado (conjunto de ligações para localizações)

O conceito de estado tem a ver com um tipo particular de ligações: as ligações de localizações.

Chama-se **estado** a um conjunto de ligações que associam localizações de memória a entidades. Matematicamente um estado é uma função de localizações para entidades, ou seja uma função com o seguinte tipo:

```
Localizações -> Entidades
```

As linguagens funcionais puras não possuem estado. Este facto tem a desvantagem de reduzir a variedade de ligações que se podem estabelecer. Mas tem a vantagem de simplificar a linguagem; outra vantagem é o facto da linguagem ficar mais segura pois sabe-se que a maioria dos dos bugs dos programas estão relacionados com variáveis mutáveis ou com apontadores.

Modelo de execução para diversos tipos de linguagens de programação com escopo estático

Na implementação da maioria das linguagens de programação, é necessário considerar diversos aspetos para garantir que os efeitos da execução dos programas são os desejados. Algumas das questões mais importantes a considerar são as seguintes:

- Avaliação de expressões
- Acesso a variáveis
- Chamada e retorno de funções
- Passagem de funções como argumento
- Funções retornadas por outras funções

Vamos estudar estas questões com a ajuda dum modelo de implementação. Note que o objectivo de qualquer modelo é simplificar o estudo uma realidade, omitindo deliberadamente pormenores não essenciais e outras complicações.

Avaliação de expressões

No nosso modelo usamos uma pilha para avaliar expressões. Para cada operação a executar, primeiro os seus argumentos são empilhados e depois a operação é activada. A operação retira os argumentos do topo da pilha, calcula o resultado e deixa-o no topo da pilha.

Acesso a variáveis

1 - Numa linguagem sem recursividade nem aninhamento de funções ou procedimentos, como é o caso do Fortran 77, cada procedimento é dono duma zona de memória própria onde as suas variáveis locais são guardadas. Todas as variáveis são estáticas, ou seja têm localizações de memória fixas. Diversas chamadas do mesmo procedimento reutilizam a mesma zona de memória... não há problema porque, como foi dito, a recursividade não é permitida.

2 - Numa linguagem com recursividade mas sem aninhamento de funções, como é o caso do C, há dois casos a considerar. As variáveis globais, ou seja as variáveis declaradas fora das funções, são consideradas variáveis estáticas e têm localizações de memória fixas. Já as variáveis locais são criadas dinamicamente no topo duma pilha sempre que a sua função-mãe é activada, e são removidas do topo da pilha quando a execução da função-mãe termina. A partir do interior duma função em C, as variáveis acessíveis são as seguintes: as variáveis locais da activação mais recente dessa função e as variáveis globais que não tenham sido redefinidas dentro da função.

3 - Numa linguagem com recursividade e com aninhamento de funções, como são os casos do Pascal, do OCaml e do GCC, a implementação fica mais complicada porque surgem variáveis declaradas nas funções envolventes, as chamadas **variáveis intermédias**. Para aceder às variáveis intermédias é preciso guardar na activação de cada função o respectivo ambiente de definição ou seja um apontador para a activação mais recente da função envolvente (**static link**). Para aceder a uma variável intermédia é preciso percorrer parte duma lista ligada constituída por static links. Foi Dijkstra quem, em 1960, inventou a técnica dos static links. [Nunca linguagem com escopo dinâmico a implementação seria mais simples por não ser necessario usar static links.]

Nota: GCC é um nome dum compilador que suporta uma linguagem um pouco mais vasta do que o C de base. Mas o termo GCC também pode ser usado para designar a linguagem suportada por esse compilador.

Considere o seguinte exemplo, em GCC. Do ponto de vista do interior da função h, as duas variáveis locais da função f são consideradas variáveis intermédias.

```
#include <stdio.h>

int i = 1, j = 1, k = 1, l = 1 ;    // variáveis globais

void f(void) {
    int i = 2, j = 2 ;
    void h(void) {        // função aninhada
        int i = 3 ;
        l = i + j + k ;
    }
    void g(void) {        // outra função aninhada
        int i = 7, j = 7, k = 7, l = 7 ;
        h() ;
    }
    g() ;
}

int main(void)
{
    f() ;
    printf("%d\n", l) ;
    return 0 ;
}
```

Chamada e retorno de funções

Nas linguagem que suportam recursividade, o estado da invocação duma função é capturado numa estrutura chamada **registo de activação (frame)**. Num registo de activação guarda-se a seguinte informação: os argumentos e as variáveis locais da função, o endereço de retorno da função, e um apontador para o registo de activação da função que fez a chamada (**dinamic link**).

Se a linguagem também permitir aninhamento de funções, ainda é preciso guardar no registo de activação o ambiente de definição da função ou seja um apontador para o registo de activação mais recente da função envolvente (**static link**). Repare que a função envolvente pode não ser a mesma função que fez a chamada.

Os registos de activação organizam-se de forma natural numa pilha, a chamada **pilha de execução**. A chamada duma função causa o empilhamento dum novo registo de activação; o retorno duma função causa o desempilhar do registo de activação mais recente.

Passagem de funções como argumento - closures

No caso duma linguagem sem aninhamento de funções, como o C, a passagem duma função como argumento é trivial: basta passar o endereço da função.

No caso duma linguagem com aninhamento de funções, a passagem duma função como argumento é mais complicada. A função passada pode ter necessidade de aceder a variáveis intermédias, e por isso, juntamente com o endereço da função tem também de ser passado o respectivo ambiente de definição (static link).

Portanto a implementação da passagem duma função como argumento envolve tecnicamente a passagem dum par ordenado a que se chama **closure** (ou **fecho** em Português):

closure = (endereço da função, static link)

A razão de ser do nome closure é a seguinte: Em geral, uma função é uma entidade "aberta" (incompleta) pois o seu corpo pode referir entidades desconhecidas localmente. Mas o par *função + respectivo ambiente de definição* já é uma entidade fechada pois o ambiente de definição "fecha" (completa) o significado da função.

No seguinte exemplo, em GCC, ocorre uma passagem de função como argumento. Medite no código para confirmar a necessidade das closures.

```
#include <stdio.h>
```

```
int i = 1, j = 1, k = 1, l = 1 ;

void f(void) {
    int i = 2, j = 2 ;
    void h(void) {        // função aninhada
        int i = 3 ;
        l = i + j + k ;
    }
    void g(void (*p)(void)) {     // outra função aninhada com par. funcional
        int i = 7, j = 7, k = 7, l = 7 ;
        (*p)() ;
    }
    g(&h) ;
}

int main(void)
{
    f() ;
    printf("%d\n", l) ;
    return 0 ;
}
```

Funções retornadas por outras funções

No caso duma linguagem sem aninhamento de funções, como o C, implementar o retorno duma função a partir de outra função é trivial: basta retornar o endereço da função-resultado.

No caso duma linguagem com aninhamento de funções, como o OCaml, a implementação desse mecanismo é mais complicada. Quando se retorna uma função, o que tecnicamente precisa de ser retornado é uma closure.

Mas aqui pode haver problema. A closure retornada pode depender de entidades locais à função que produz o resultado. Assim quando a função retorna a closure e termina, o seu registo de activação não pode ser destruído. Uma solução é transferir o registo de activação para o heap, ou seja para a zona das variáveis dinâmicas. Esta é a solução adoptada na linguagem funcional Lua. Outra solução, mais radical, consiste em abandonar completamente o modelo da pilha de execução e passar a criar todos os registos de activação no heap (deixando que seja o gestor de memória a descobrir o momento em que eles podem ser removidos com segurança).

Esta complicação explica o facto do retorno de funções ser um mecanismo não suportado pela maioria das linguagens de programação. Praticamente só as linguagens funcionais é que suportam este mecanismo.

No seguinte exemplo, em OCaml, a função f retorna uma closure que depende do x local ao f. Quando a closure retornada é depois chamada, o x tem de continuar disponível, apesar da execução de f já ter terminado.

```
let f x =
    let g y = x + y in
        g
;;

(f 5) 4 ;;
```

Modelo de execução detalhado para linguagem com escopo estático e aninhamento de funções

Para concretizar melhor as ideias anteriores, vamos desenvolver um modelo de execução ainda mais detalhado para o caso das linguagens com escopo estático e aninhamento de funções.

Usaremos a linguagem C como veículo de formalização, para concretizar melhor as ideias. O código C que vai aparecer tem grandes semelhanças com o código duma máquina virtual típica.

Áreas de memória e registos

O nosso modelo tem três áreas de memória:

- **stack** - Guarda os registos de activação e também como memória de trabalho para avaliar expressões.
- **code** - Guarda o código do programa que está a correr.
- **heap** - Guarda as variáveis dinâmicas do programa.

Pouco será dito sobre a zona de código e sobre o heap. O stack vai receber quase toda a nossa atenção.

Os registos do modelo são os seguintes:

- **SP** - Indica o topo da pilha de execução.
- **FP** - Indica o registo de execução correntemente activo, ou seja para o registo de execução que se encontra no topo da pilha de execução.
- **PC** - Indica qual a instrução corrente, na zona de código.

Eis uma concretização em C destas ideias:

```
#define STACK_SIZE   20000
#define CODE_SIZE    20000
#define HEAP_SIZE    20000

typedef unsigned Word ;
typedef Word *Pt ;

#define Push(v)      (*SP++ = (Word)(v))
#define Pop()        (*--SP)

Word stack[STACK_SIZE] ;
Word code[CODE_SIZE] ;
Word heap[HEAP_SIZE] ;

Pt SP = stack;       // Stack pointer
Pt FP = stack ;      // Frame pointer
Pt PC = code ;       // Program counter
```

Avaliação de expressões

Como já foi dito, usamos a pilha de execução como auxiliar da avaliação de expressões. Os argumentos das operações são empilhados e depois a operação encontra os seus argumentos no topo da pilha.

Eis uma concretização em C destas ideias:

```
void PushConst(Word c) {
    Push(c) ;              // push the constant
}

void Add(void) {
    Word arg1 = Pop() ;    // pop the first argument
    Word arg2 = Pop() ;    // pop the second argument
    Push(arg1 + arg2) ;    // push the result
}

...
```

Acesso a variáveis

Em tempo de execução os nomes das funções não precisam de estar disponíveis. Para referenciar uma variável em tempo de execução basta usar as chamadas **coordenadas da variável**, constituídas por dois valores:

- **Diferença de nível (nesting)** - Diferença de nível de aninhamento entre o ponto da referência e o ponto da declaração.
- **Deslocamento (offset)** - Deslocamento da variável dentro do registo de activação a que pertence.

A diferença de nível representa o número de vezes que é necessário descer pelos static links para se atingir o registo de activação onde se encontra a variável pretendida. No caso do acesso a uma variável local, a diferença de nível é zero.

O compilador da linguagem consegue determinar com muita facilidade as coordenadas que permitem referenciar cada variável. Um ser humano também o faz facilmente. Por exemplo, no primeiro exemplo desta aula, qual é a diferença de nível das utilizações das variáveis i, j, k e l dentro da função f.

Concretização em C:

```
void LoadVarAddr(int nesting, int offset) {
    for( Pt pt = FP ; nesting-- ; pt = SL(pt) ) ;  // find the frame of the var
    Push(pt + offset) ;             // push the var address
}

void Value() {
    Pt pt = Pop() ;         // pop the var address
    Push(*pt) ;             // push the value of the variable
}

void Assign() {
    Pt pt = Pop() ;         // pop the var address
    Pt value = Pop() ;      // pop the value to store
    *pt = value ;           // do the assignement
}
```

Chamada e retorno de funções

No nosso modelo, o formato dum registo de activação é o que se apresenta abaixo. Convenciona-se que registo FP aponta para a célula onde é guardado o dynamic link do registo de activação mais recente. Nos acessos aos argumentos e variáveis da função, usa-se o registo FP como ponto de referência. Assim, os argumentos da função têm offset negativo, a começar em -1, e as variáveis locais têm offset positivo, a começar em 3. No diagrama assume-se que a pilha cresce para cima.

```
          <- SP
Variável local n
...
Variável local 1
Variável local 0
PC
SL
DL        <- FP
Argumento m
...
Argumento 1
Argumento 0
```

A principal complicação envolvida na activação duma função e na criação do seu registo de activação é o calculo do static link (que representa o ambiente de definição dessa função). O static link tem de ser guardado no registo de activação da função para permitir o subsequente acesso a variáveis não-locais. Para determinar o static link é preciso conhecer a **diferença de nível (nesting)** entre o ponto da declaração da função e o ponto da sua chamada. A diferença de nível representa o número de vezes que é necessário descer pelos static links para se atingir o registo de activação onde se encontra o static link a copiar. Numa chamada recursiva a diferença de nível é zero. Na chamada duma função filha, a diferença de nível é -1.

No nosso modelo a chamada e retorno de funções estão organizados da seguinte forma:

- Antes da função ser chamada, os seus argumentos são empilhados;
- No momento da chamada são empilhados o dinamic link, o static link e o endereço de retorno;
- À entrada da função, esta reserva espaço para as suas variáveis locais.
- Quando a função termina ela desempilha o registo de activação completo, incluindo os argumentos.
- A função deixa no topo da pilha o seu resultado.

Concretização em C:

```
#define DL(fp)      ((fp)[0])
#define SL(fp)      ((fp)[1])
#define PC(fp)      ((fp)[2])

void Call(int nesting, Pt addr) {
    Push(FP) ;          // push dynamic link
    if( nesting == -1 )
        Push(FP) ;      // push static link
    else {
        for( Pt pt = FP ; nesting-- ; pt = SL(pt) ) ;
        Push(SL(pt)) ;  // push static link
    }
    Push(PC) ;          // push return address
    PC = addr ;         // jump to the function
}

void Enter(int locSpace) {
    if( STACK_SIZE - (SP - stack) < locSpace + 20 ) {
        fprintf(stderr, "Stack overflow at PC = %d\n", PC) ;
        exit(1) ;
    }
    FP = SP - 3 ;    // FP points to the dynamic link inside the frame
    SP += locSpace ;
}

void Exit(int argSpace) {
    Word res = Pop() ;    // save the result here
    SP = FP - paramSpace; // discard the entire frame, including the arguments
    PC = PC(FP) ;         // restore the program counter
    FP = DL(FP) ;         // restore the previous frame pointer
    Push(res) ;           // push the result
}
```

Passagem de funções como argumento - closures

Já sabemos que a passagem duma função como argumento requer efectivamente a passagem duma closure como argumento. A principal complicação envolvida na criação da closure é o calculo do static link. Mas agora não há qualquer novidade: ele calcula-se exactamente da mesma forma que na chamada duma função normal.

A activação duma função através duma closure é mais simples do que a activação directa duma função pois o static link já se encontra calculado na closure.

Concretização em C:

```
void PushClosureArg(int nesting, Pt addr) {
    if( nesting == -1 )
        Push(FP) ;      // push static link
    else {
        for( Pt pt = FP ; nesting-- ; pt = SL(pt) ) ;
        Push(SL(pt)) ;  // push static link
    }
    Push(addr) ;
}
```

```
void CallClosure(int nesting, int offset) {
    for( Pt pt = FP ; nesting-- ; pt = SL(pt) ) ;  // find the frame of the closure-
argument
    Pt sl = pt[offset] ;       // get the static link from the closure
    Pt addr = pt[offset + 1] ; // get the function address from the closure
    Push(FP) ;                 // push dynamic link
    Push(sl) ;                 // push static link
    Push(PC) ;                 // push return address
    PC = addr ;                // jump to the function
}
```

Funções retornadas por outras funções

O estudo mais detalhado desta questão é um tópico avançado que está fora do âmbito da cadeira de LAP.

Exercícios

- Relativamente ao primeiro exemplo desta aula, mostre o estado da pilha de execução à entrada da função h.
- Relativamente ao segundo exemplo desta aula, mostre o estado da pilha de execução à entrada da função h.

Para efeitos da criação do registo de activação inicial, convém imaginar que cada programa em C está embebido numa função chamada start, sem argumentos. É importante tratar todas as entidades globais do programa como sendo locais à função imaginária start.

Assuma também que a primeira célula da pilha de execução é identificada como posição 0, a segunda célula da pilha de execução é identificada como posição 1, etc.

```
Resolução do primeiro exercício:

              <- SP
25: i  3 --h--
    PC ?
    SL 10
    DL 15     <- FP
    l  7 --g--
20: k  7
    j  7
    i  7
    PC ?
    SL 10
15: DL 10
    j  2 --f--
    i  2
    PC ?
    SL 0
10: DL 7
    PC ? --main--
    SL 0
    DL 0
    l  1 --start--
 5: k  1
    j  1
    i  1
    PC ?
    SL ?
 0: DL ?
```

Tipos

Um **tipo** representa uma coleção de elementos de dados e tem associados um conjunto de *literais* mais um conjunto de *operações*.

Por exemplo, em ML o tipo `int` representa o conjunto do inteiros, tem associados os literais ..., -2, -1, 0, 1, 2, etc., e as operações `+`, `-`, `*`, `div`, `mod`, `succ`, etc.

Numa linguagem sem tipos, e.g. Assembler, uma operação pode ser aplicada a quaisquer dados sem que qualquer validação seja feita. Os dados são vistos como simples sequências de bits e cada operação interpreta uma sequências de bits da maneira que lhe convém.

Numa linguagem com tipos, cada elemento de dados tem um tipo associado. Ao tentar aplicar uma operação a valores que não têm o tipo esperado obtém-se um **erro de tipo**.

Géneros de tipos

Cada linguagem de programação oferece um conjunto de tipos simples e um conjunto de construtores de tipos. É assim possível distinguir dois géneros de tipos:

- **Tipos simples:** tipos pré-definidos cujos valores são escalares atómicos inteiros, caracteres, reais, booleanos, enumerados.
- **Tipos estruturados:** tipos compostos definidos pelo programador aplicando determinadas construções a tipos existentes: arrays, registos, listas, tuplos, classes, funções, apontadores.

Utilidade dos tipos

Para que servem os tipos? No caso das linguagens com tipificação estática:

- Permitem a deteção antecipada de erros, em tempo de compilação.
- São úteis na otimização de código gerado.
- Ajudam a documentar os programas, tornando mais claras as intenções do programador (exceto se for usada inferência de tipos).
- Suportam a definição de módulos que ocultam a representação interna dos valores.

No caso das linguagens com tipificação dinâmica:

- Suportam um modelo de execução no qual se aborta a execução do programa logo que seja detetado algum erro de tipo.

Sistemas de tipos

Um **sistema de tipos** é um conjunto de regras que:

- Associa tipos aos literais e às expressões duma linguagem. Por exemplo o literal `2`, e a expressão `2+5*3` são de tipo inteiro.
- Determina quais são as manipulações permitidas dos valores de cada tipo. Por exemplo, os inteiros podem ser manipulados usando apenas as operações associadas ao tipo inteiro.
- Define como os valores dos vários tipos interagem. Por exemplo, em C a expressão `2 + 5.2` provoca para o valor `2` uma conversão automática de tipo (uma *coerção*).

Para exemplificar, eis três regras que fazem parte do sistema de tipos da linguagem OCaml:

```
                                              exp : bool    exp' : bool
   --------------        -------------       ---------------------------
    false : bool           true : bool           exp&&exp' : bool
```
Nestas regras, por cima da barra são colocadas pré-condições e por baixo da barra são colocadas conclusões. As duas primeiras regras dizem que `false` e `true` são valores válidos de tipo `bool`; a terceira regra diz que se `exp` e `exp'` forem expressões válidas de tipo `bool`, então `exp&&exp'` é também uma expressão válida de tipo `bool`.

Não mostramos mais regras, pois a formalização de sistemas de tipo não é matéria de LAP. Mas, pelo menos fica uma ideia.

Estas regras são úteis por várias razões. Por exemplo, servem para provar que o sistema de tipos é consistente (ou seja, que a cada expressão é associado um tipo único), e também servem como base para a escrita duma parte importante do compilador.

Verificação de tipos (Type checking)

Chama-se **verificação de tipos** ao processo de validação dum programa face às regras dum sistema de tipos.

Tipificação estática

Diz-se que uma linguagem usa **tipificação estática** se a verificação de tipos for efetuada em tempo de compilação. Os erros de tipo são detetados antes do programa começar a correr. Nestas linguagens declaram-se os tipos das variáveis, dos argumentos e do resultado das funções para o compilador ter uma base para fazer as suas validações.

As seguintes linguagens usam tipificação estática: ML, C, C++ e Java.

Numa linguagem com tipificação estática, os tipos são associados às expressões que aparecem no texto dos programas e é o próprio texto que é validado. Mais tarde, durante a execução, os tipos dos valores são ignorados de forma geral, porque o compilador já garantiu a ausência de erros de tipo.

Mas atenção: Num programa validado, ou seja sem erros de tipo, podem mesmo assim ocorrer diversos problemas durante a execução:

1. Certos **valores particulares** podem gerar exceções: **zero** usado como denominador duma divisão, ou **null** usado como recetor de mensagem.
2. O programa pode entrar num **ciclo infinito**.
3. O programa pode estar **incorreto**, ou seja mal feito, e não fazer o que o programador pretendia.

Costuma dizer-se que os sistemas de tipos das linguagens com tipificação estática **são conservadores**. Quando se usa tipificação estática, os tipos representam informação incerta, correspondendo a "aproximações" dos valores que realmente serão usados em tempo de execução (e que não se podem prever). Para garantir segurança, o compilador tem sempre de assumir o pior caso, mesmo que o pior caso nem sempre ocorra. Por exemplo, no código C++ que se segue, a variável `a`, de tipo `Animal *` recebe um gato. Depois, mais adiante, tenta-se fazer miar o animal apontado por `a`. Mas o compilador tem considerar que, nessa altura, a variável `a` poderá já não apontar para um gato e, por isso, emite um erro de compilação; no entanto, em tempo de execução até poderia até não haver problema...
```
Animal *a = new Cat ;
...
a->Meou() ;    // ERRO DE TIPO
```

Tipificação dinâmica

Diz-se que uma linguagem usa **tipificação dinâmica** se a verificação de tipos for efetuada em tempo de execução. Os erros de tipo são detetados durante a execução dos programas. Nestas linguagens não de declaram os tipos das

variáveis, dos argumentos e do resultado das funções. Nestas linguagens uma mesma variável pode conter valores de tipos diferentes, em diferentes momentos da execução.

As seguintes linguagens usam tipificação dinâmica: JavaScript, Prolog, Lisp, Perl, Python, Ruby, APL e Smalltalk.

Numa linguagem com tipificação dinâmica, os tipos são associados aos valores que são usados em tempo de execução. Portanto, em tempo de execução tem de existir informação de tipo associada aos valores. Sempre que se aplica uma operação a alguns argumentos, o tipo dos argumentos é testado pelo sistema de execução.

Os sistemas de tipos das linguagens com tipificação dinâmica **não precisam de ser conservadores** pois exercem a sua influência durante a execução dos programas, quando os argumentos exatos das operações são conhecidos e estão disponíveis para teste.

Elementos de tipificação dinâmica dentro de linguagens com tipificação estática

Há linguagens de programação com tipificação estática que incluem alguns elementos de tipificação dinâmica.

Um exemplo em Java: a operação **instanceof** permite ao programador testar dinamicamente o tipo de qualquer objeto; a operação de cast aplicada a um objeto também executa um teste de tipo implícito em tempo de execução. Para isto funcionar, os objetos em Java precisam de registar a classe a que pertencem.

Outro exemplo em C++: a operação **dynamic_cast** também envolve um teste dinâmico de tipo. Por isso os objetos em C++ também precisam de registar a classe a que pertencem. (Na realidade, alguns de compiladores de C++ requerem a ativação duma flag chamada RTTI (Run-Time Type Information) para que o cast dinâmico funcione. No caso do g++, essa flag está cativada por omissão.)

Duas escolas de programação

Nos dias de hoje, entre as linguagens de programação mais usadas encontram-se linguagens de programação com tipificação estática e linguagens de programação com tipificação dinâmica. Existem adeptos enquadrados em cada uma das duas escolas de programação, os quais defendem renhidamente a sua "dama" (veja por exemplo esta discussão).

Por exemplo, os adeptos da tipificação estática acreditam que os seus programas são mais seguros depois de verificados, mas os adeptos da tipificação dinâmica argumentam que conseguem programar melhor as suas ideias sem constrangimentos artificiais e que apesar de tudo os seus programas têm provado ser robustos e conter poucos erros.

Na verdade é perfeitamente possível ser bons resultados dentro de cada uma das escolas de programação desde que se usem boas técnicas de desenvolvimento de software. Uma das técnica mais importante é certamente a técnica de escrever muitos tests unitários para validar sistematicamente todos os aspetos do software em desenvolvimento.

Só uma curiosidade: O CLIP, o sistema da informação da FCT, é um exemplo de software da escola da tipificação dinâmica - está escrito em Prolog.

Sistemas de tipos seguros

Um sistema de tipos diz-se **seguro** se conseguir garantir que não ocorrem erros de tipo durante a execução dos programas.

Os sistemas de tipos do C e C++ não são seguros porque:

- Alguns casts permitem violar o sistema de tipos de forma muito básica. Por exemplo é possível converter um inteiro num apontador para função.
- Mesmo sem usar um cast, usando aritmética de apontadores é fácil pôr um apontador para int a apontar para uma variável de tipo double e depois escrever nela (ou em parte dela) um inteiro.
- A gestão explícita de memória também é um problema. Suponhamos que temos um apontador para uma variável dinâmica inteira e que libertamos essa variável. Suponhamos que logo a seguir criarmos uma variável dinâmica de tipo double. Suponhamos ainda que, por acaso, a variável é criada na zona de memória anteriormente libertada. Agora a variável real pode ser acedida usando o apontador para inteiro original, pois apesar de ser considerar um apontador inválido, ele manteve o endereço original.

Exemplos de linguagens com sistemas de tipos seguros:

- ML
- Java
- Smalltalk
- Lisp

Repare que na lista anterior aparecem linguagens com tipificação estática e linguagens com tipificação dinâmica.

Ambientes de programação

Um **ambiente de programação** é uma coleção de ferramentas usadas no desenvolvimento de software. Um bom sistema de desenvolvimento aumenta a produtividade dos programadores e também contribui um pouco para aumentar a qualidade do software produzido.

Um ambiente minimal é tipicamente constituído por:

- um sistema de ficheiros,
- um editor de texto,
- um interpretador (ou, em alternativa, um compilador e um ligador)

Um ambiente sofisticado é tipicamente constituído por:

- um sistema de ficheiros,
- numerosas ferramentas, incluindo editor, (depurador) debugger, gerador de documentação, framework para testes unitários, etc.
- essas ferramentas estão bem integradas umas com as outras;
- são oferecidas ao utilizador duas interfaces uniformes: uma interfaces gráfica e uma interface baseada em linha de comando.

O ambiente de desenvolvimento JDK para Java

O ambiente de desenvolvimento não-visual JDK (Java Developer's Kit) consiste num conjunto de ferramentas que ajudam a **desenvolver**, **testar**, **documentar** e **executar** programas em Java. Estas ferramentas podem ser usadas na linha de comando, mas também podem ser integradas em ambientes de desenvolvimento gráficos, como por exemplo o Eclipse. O JDK é gratuitamente disponibilizado pela empresa Sun e existem versões para Windows, MacOS, Linux, etc.

Ferramentas do JDK

javac	Compilador
java	Executor de *programas independentes*
javadoc	Gerador de documentação
appletviewer	Executor de *applets*
jdb	Depurador
javap	Disassembler
javah	Gerador de *header files* para métodos nativos escritos em C

Como se compilam programas em Java?

- Os programas em Java compilam-se por meio do comando **javac**. Por exemplo: `javac Sets.java`. O comando **javac** permite compilar tanto **programas independentes** como **applets**.
- Os ficheiros compilados pelo comando **javac** têm obrigatoriamente a extensão ".java".
- Quando cativado, o compilador cria na diretoria corrente diversos ficheiros com a extensão ".class", um por cada classe ou interface definidas nos ficheiros fonte. Os ficheiros ".class" contêm código executável, num formato padrão chamado **bytecode**.

Como se executam programas em Java?

- Se se tratarem de **programas independentes**, executam-se usando o comando **java**. Por exemplo: `java TestSets`. Neste exemplo, "TestSets" refere-se ao ficheiro "TestSets.class" na diretoria corrente: repare que o comando `java` não admite a escrita da extensão ".class".
- Se se tratarem de **applets**, executam-se usando o comando **appletviewer** ou então por intermédio dum browser WWW.
- A componente de software que permite executar ficheiros de de *bytecode* chama-se **Java Virtual Machine**. Está incorporada no comando **java**, no comando **appletviewer** e na maioria dos browsers WWW modernos.

Como se documentam programas em Java?

- O gerador de documentação, **javadoc** converte ficheiros ".java" em ficheiros HTML, visualizáveis usando qualquer browser WWW. Estes ficheiros HTML incluem documentação sobre as classes, interfaces, métodos, variáveis e exceções definidos nos ficheiros fonte e ainda o conteúdo de comentários especiais, da forma /** ... */, escritos pelo programador. A vasta documentação sobre a plataforma Java que é fornecida pela Sun foi gerada pelo comando **javadoc**.

As ferramentas do OCaml

A distribuição do OCaml inclui uma vasto conjunto de ferramentas disponíveis no Windows, Linux e MacOS. Estas ferramentas podem ser usadas na linha de comando, mas também podem ser integradas em ambientes de desenvolvimento gráficos.

Repare que a lista de ferramentas abaixo inclui um profiler que serve para o programador descobrir quais as partes do programa onde se gasta mais tempo. Essas são as partes do programa que mais interessa optimizar.

Ferramentas do OCaml

ocaml	Interpretador
ocamlc	Compilador para a máquina virtual CAML
ocamlopt	Compilador de código nativo
ocamlrun	Executor da máquina abstrata CAML
ocamlbrowser	Permite navegar e inspecionar o conteúdo dos módulos de biblioteca
ocamldebug	Depurador de código fonte
ocamldoc	Gerador de documentação
ocamldep	Inspeciona um conjunto de ficheiros fonte e gera automaticamente informação de dependências para a ferramenta make
ocamlcp	Profiler, insere no código fonte a contagem de quantas vezes cada função é chamada, etc.

ocamlprof Interpreta o output da execução de programas processados usando ocamlcp
ocamllex Gera reconhecedores de expressões regulares diretamente a partir dessas expressões
ocamlyacc Gera parsers diretamente a partir de gramáticas LALR(1)
ocamlp4 Processador do código fonte de programas ocaml (para pretty-print, por exemplo)
ocamldumpobj Disassembler de ficheiros .cmo
ocamlobjinfo Inspeciona ficheiros .cmo, .cmi, .cma

Unix e Gnu/Linux

O Unix é um sistema operativo mas também um sistema de desenvolvimento e manutenção de software. Atualmente a sua versão open-source, chamada de Gnu/Linux, é muito popular e tem sido usada no desenvolvimento de software "open-source" e de software comercial.

O Unix disponibiliza uma grande quantidade de ferramentas de desenvolvimento, mas deve ser dito que essas ferramentas foram nascendo de forma caótica e que não há uma interface consistente entre elas. Algumas ferramentas são também muito complexas, embora cumpram os seus objetivos de forma admirável, na maior parte dos casos.

gnu tools

Os **gnu tools** são um conjunto de ferramentas que ajudam a produzir software portável, que funciona em diversas versões do Unix e noutros sistemas operativos.

autoconf Ferramenta para criar projetos de software portáveis e reconfiguráveis
automake Gerador automático de makefiles
libtool Ferramenta para criar bibliotecas de software

make

A utilitário **make** pode ser usado para gerir automaticamente pequenos projetos. A sua principal utilização é determinar automaticamente que partes do projeto precisam de ser recompiladas e produzir os comandos para concretizar a recompilação. O make considera as dependências entre os diversos ficheiros do projeto para saber o que é precisos fazer em cada momento.

Geralmente, cada projeto contém um ficheiro chamado **Makefile** onde se declaram as dependências entre os ficheiros, e onde se definem regras que dizem como os diversos ficheiros devem ser atualizados. Para além da recompilação, podem ser definidas outras ações, tais como apagar os ficheiros auxiliares do projeto, ou produzir o arquivo da distribuição do projeto.

O seguinte exemplo é um ficheiro Makefile que poderia ter sido usado no primeiro projeto de LAP (o projeto das árvores de decisão). O ficheiro declara as dependências entre ficheiros e tem regras que dizem como obter uns ficheiros a partir de outros (as regras começam obrigatoriamente por um tab).

```
APP = dt
VERSION = 0.1
COMP = ocamlc
INTERFS = DTree.mli
OBJS = DTree.cmo dt.cmo
GROUP = 123_456

$(APP): $(OBJS)
        $(COMP) -o $(APP) $(INTERFS) $(OBJS)

%.cmo: %.ml
        $(COMP) $(INTERFS) -c $<

clean:
        rm -f $(APP) *.cmo *.cmi
```

```
dist: clean
	mkdir $(GROUP)
	cp *.ml *.mli $(GROUP)
	zip -r $(GROUP).zip $(GROUP)
	rm -rf $(GROUP)
```

Eis uma pequena sessão interativa, que pressupõe a existência do ficheiro `Makefile` na diretoria corrente:

```
$ ls
dt.ml  DTree.ml  DTree.mli  Makefile
$ make DTree.cmo
ocamlc DTree.mli -c DTree.ml
$ make
ocamlc DTree.mli -c dt.ml
ocamlc -o dt DTree.mli DTree.cmo dt.cmo
$ make clean
rm -f dt *.cmo *.cmi
$ make dist
rm -f dt *.cmo *.cmi
mkdir 123_456
cp *.ml *.mli 123_456
zip -r 123_456.zip 123_456
  adding: 123_456/ (stored 0%)
  adding: 123_456/DTree.ml (deflated 72%)
  adding: 123_456/dt.ml (deflated 67%)
  adding: 123_456/DTree.mli (deflated 65%)
rm -rf 123_456
$ ls
123_456.zip  dt.ml  DTree.ml  DTree.mli  Makefile
```

gdb

O Gnu Debugger trabalha com linguagens implementadas nativamente e tem um funcionalidade extensíssima. Mostramos só como usar o `gdb` para descobrir qual a função onde um programa está a *rebentar*.

O seguinte programa está errado:

```
char str[128] ;

void f(void)
{
    char *pt = str ;
    int i ;
    for( i = 0 ; i < 10000 ; i++ ) <-- Provoca estoiro
        *pt++ = 'a' ;
}

void g(void) {
    f() ;
}

int main()
{
    g() ;
    return 0 ;
}
```

O `gdb` permite mostrar o conteúdo da pilha de execução no momento do estoiro, o que geralmente é muito útil para descobrir as razões do erro.

```
$ ./a
Segmentation fault (core dumped)
$ gdb ./a
GNU gdb 6.6-debian
Copyright (C) 2006 Free Software Foundation, Inc.
(gdb) run
Starting program: /media/EXTERN/amd1/z/a
```

```
Program received signal SIGSEGV, Segmentation fault.
0x0804835d in f ()                    <-- Estoiro na função f
(gdb) backtrace
#0  0x0804835d in f ()
#1  0x0804837b in g ()
#2  0x08048390 in main ()             <-- Pilha de execução no momento do estoiro
(gdb)
```

cvs

O **cvs** (Concurrent Versions System) um sistema de apoio ao desenvolvimento de projetos de software. Tem duas funcionalidades essenciais:

- Suporta o **registo cronológico** das alterações introduzidas nos ficheiros-fonte do projeto;
- Suporta a **colaboração** entre os diversos participantes de projeto, e sabe lidar com o problema da submissão de **alterações conflituantes**.

Atualmente, o CVS (ou o SVN) é usado na maioria dos projetos de *open source*. Podemos confirmar isso visitando o site Sourceforge: trata-se do principal site hospedeiro deste tipo de projetos. Poderá observar que, inclusivamente, projetos de software livre para o Windows (e.g. Dev-Pascal, Dev-C++) usam o CVS.

Uma ferramenta como o CVS é essencial para a viabilidade do software livre, pois este tipo de software é tipicamente desenvolvido coletivamente por um grande número de voluntários geograficamente espalhados e não pagos. Nenhum desses voluntários teria tempo nem paciência para processar manualmente as numerosas submissões de código novo ou corrigido.

Uma ferramenta do tipo do CVS também faz falta no desenvolvimento de software proprietário, mas no caso do software livre a questão é mais evidente.

Exemplos de software desenvolvido com a ajuda do CVS

- Apache web server
- X window system
- GNOME desktop
- Mozilla web browser
- OpenOffice
- GCC gnu compiler collection
- Debian linux distribuition
- Anjuta IDE for rapid application development

Alternativas ao CVS

Existem diversas alternativas ao CVS, o qual já começa a mostrar a sua longa idade:

- Alternativas *open source*: SVN - Subversion, GNU arch, Monotone, .
- Alternativas comerciais: BitKeeper.

Atualmente o uso do Subversion está em expansão muito rápida. Não admira: introduz melhorias significativas e foi exatamente criado para substituir o CVS. O eSvn é um excelente front-end gráfico para o Subversion (veja os screenshots).

Serviço de SNV do DI, para ser usado por alunos e docentes.

IDEs

Um IDE, ou ambiente de desenvolvimento integrado, é um programa único dentro do qual se processa todo o desenvolvimento do software. Geralmente a interface é gráfica.

Exemplos: Eclipse, Visual Studio .Net, Anjuta, Code::Blocks.

Os IDEs mais sofisticados são estendíveis através de plugins. Isso permite que o mesmo sistema possa ser usado para trabalhar em diversas linguagens.

Escolha duma linguagem de programação

A escolha duma linguagem de programação para usar num projeto de programação depende de muitas circunstâncias. Em contextos diferentes deve dar-se maior ou menor importância a diferentes aspetos. Em algumas situações é conveniente usar mais do que uma linguagem.

- Experiência prévia com a linguagem.
- Facilidade em aprender a linguagem.
- Facilidade em usar a linguagem.
- Suporte de boas práticas de programação.
- Suporte de tecnologias de desenvolvimento de software recentes.
- Qualidade das ferramentas de desenvolvimento disponíveis.
- Disponibilidade de grande número de programadores.
- Suporte para deteção atempada de bugs.
- Padronização da linguagem.
- Expressividade e produtividade.
- Riqueza das bibliotecas disponíveis.
- Portabilidade.
- Segurança.
- Velocidade de execução.
- Custo.
- Facilidade em escrever scripts.
- Adequação a programação de sistemas.
- Suporte para concorrência.
- Suporte para introspeção.

Discussão sobre as linguagens usadas na cadeira

OCaml

- Linguagem simples, fácil de usar, suporta abstrações muito poderosas.
- A programação pode ser de muito alto-nível e os programas tendem a ser compactos.
- Tem um conjunto de bibliotecas rico e há muitas ferramentas desenvolvidas para serem usadas com o OCaml.
- Suporta programação funcional, imperativa e orientada pelos objetos.

C

- Grande liberdade para explorar os recursos da máquina -> programação de sistemas.
- Linguagem insegura mas muito eficiente.
- Nível excelente de interoperabilidade - quase todas as linguagens suportam mecanismos de ligação ao C.
- Depuração difícil devido ao uso de apontadores e à necessidade de gerir a memória explicitamente. Os programas podem ser difíceis de ler.

C++

- C com objetos + templates + bibliotecas mais ricas.
- Tem muitas das características do C mas ajuda no desenvolvimento de projetos maiores.

Java

- Segurança em aplicações orientadas para a WEB.
- Portabilidade dos executáveis.
- Biblioteca imensamente rica.
- Suporta boas práticas de programação e tecnologias de desenvolvimento de software recentes: testes unitários, objetos, componentes (Java Beans), serviços WEB, frameworks de programação (Spring).
- Mais lenta do que o C++.

JavaScript

- Linguagem muito prática de usar, dá grande liberdade ao programador, mantendo um razoável nível de segurança quando o que está em causa é a escrita de pequenos scripts.
- Serve para programar páginas WEB interativas, mas também é usada como linguagens de scripting em muitas outras aplicações.