Silveira, Paulo Enes da

Programação directa e iterativa : métodos para criar bons programas : tributo a Jacques Arsac

http://hdl.handle.net/11067/3410
https://doi.org/10.34628/mdmj-y603

Metadados

Data de Publicação 2016

Resumo A resolução de um Problema em Programação de computadores, apresenta dificuldades que se manifestam na escolha dos caminhos a seguir para se atingirem as soluções desejadas. O presente artigo apresenta métodos de resolução de Problemas em Programação, nas técnicas de Programação Directa e de Programação Iterativa, exemplificando a sua aplicação, no paradigma de Programação Imperativa. O autor dedica ao Professor Jacques Arsac um especial tributo, pela importância que este cientista e pedagogo te...

Palavras Chave Programação de computadores, Arsac, Jacques, 1929-2014 - Crítica e interpretação

Tipo article

Revisão de Pares Não

Coleções [ULL-FCEE] LEE, n. 21 (2016)

PROGRAMAÇÃO DIRECTA E ITERATIVA
Métodos para criar bons programas
Tributo a Jacques Arsac

Paulo Enes da Silveira
Universidade Lusíada, Lisboa
Universidade Autónoma de Lisboa
paulosilveira@edu.ulusiada.pt

Resumo: A resolução de um Problema em Programação de computadores, apresenta dificuldades que se manifestam na escolha dos caminhos a seguir para se atingirem as soluções desejadas. O presente artigo apresenta métodos de resolução de Problemas em Programação, nas técnicas de Programação Directa e de Programação Iterativa, exemplificando a sua aplicação, no paradigma de Programação Imperativa. O autor dedica ao Professor Jacques Arsac um especial tributo, pela importância que este cientista e pedagogo teve no estudo e difusão da Programação, na criação de programas correctos e na concepção de métodos de Programação.

Palavras-Chave: Programação Directa; Programação Iterativa; Paradigma Imperativo; Métodos de Programação; Algoritmos.

Abstract: The resolution of a problem in computer programming presents difficulties that are shown in the choice of the ways that have to be followed to achieve the required solutions. This article presents methods to solve problems in programming, on Direct Programming and Iterative Programming techniques, exemplifying their application in the Imperative Programming paradigm. The author dedicates to Professor Jacques Arsac a special tribute, because of the importance this scientist and pedagogue has had in the study and dissemination of Programming, creation of correct programs and designing Programming methods.

Keywords: Direct Programming; Iterative Programming; Imperative Paradigm; Programming methods; Algorithms.

1. Introdução

Há mais de trinta anos que o autor do presente artigo ensina a programar computadores, explicando como se concebem os algoritmos que dão solução aos problemas que desejamos resolver, a fim de os codificar em linguagens de programação, com as suas instruções fundamentais, as quais serão, posteriormente, interpretadas e executadas pelos computadores.

A sua experiência de ensino da programação a vários milhares de alunos, ao nível dos primeiros anos do ensino universitário, permite-lhe afirmar que a maior dificuldade não reside na aprendizagem das linguagens de programação propriamente ditas, mesmo considerando os vários paradigmas de programação a que estas podem pertencer, mas sim na forma de se conceber uma solução algorítmica para um dado problema.

Neste percurso, o contributo do Professor Jacques Arsac foi fundamental no sentido de proporcionar métodos de construção de algoritmos, em particular, na programação iterativa (com ciclos) e na programação recursiva (com funções).

O autor teve a oportunidade de estudar, no início da década de 80, com este exímio Mestre da Programação, Jacques Arsac, que nos deixou as suas lições em vários livros citados ao longo deste artigo. Na sua obra "Programação Criativa" (1993a), tradução para português de *Préceptes de Programmation* (1991), Arsac expõe métodos de construção de ciclos e de funções recursivas que, ao longo do tempo, o autor sistematizou e ensinou.

O ensino da Programação tem evoluído no decurso dos seus cerca de setenta anos de história. Inicialmente seguiam-se os exemplos de quem sabia programar. Programar era uma arte de acesso difícil que se aprendia com os "gurus" da matéria.

Entretanto, especialistas foram avançando com o estudo da Programação. Em 1968, Donald Knuth aborda algoritmos fundamentais em *The Art of Computer Programming*, Vol. 1 (1968), seguido de uma série de outros volumes sob o mesmo título. Em 1976, Edsger Dijkstra vê publicado o seu livro *A Discipline of Programming* (1976). Em 1981, é publicado *The Science of Programming* (1981), da autoria de David Greis. Estes três cientistas de Programação, dos mais destacados a nível mundial, muito galardoados a nível internacional, nomeadamente com vários prémios do ACM - Association for Computing Machinery e IEEE - Institute of Electrical and Electronics Engineers, apresentam, através dos títulos dos seus livros, uma curiosa e natural evolução da Programação: primeiro é

"Arte", depois "Disciplina" e, finalmente, "Ciência". O próprio Donald Knuth, em 1995, expressou no prefácio do livro *A=B* (Petkovšek, Wilf & Zeilberger, 1996): "Science is what we understand well enough to explain to a computer. Art is everything else we do. During the past several years an important part of mathematics has been transformed from an Art to a Science".

Observar bons algoritmos com a sua beleza e eficiência pode ser um excelente incentivo para a aprendizagem da Programação. Mas quando se deseja conceber um novo algoritmo para resolver um problema, podemos ter vários caminhos a seguir. No primeiro, procura-se um algoritmo existente que resolva um problema semelhante e, inspirado por este, constrói-se uma solução. No segundo, se se for iluminado por uma boa inspiração e se se vislumbrar a solução, há que implementá-la. No terceiro, avança-se para uma solução, eventualmente baseado na experiência adquirida e, muitas vezes por tentativas, conclui-se o algoritmo. No quarto caminho, aplica-se um método sistemático que nos proporciona uma boa solução, sempre que seja possível.

Embora os três primeiros caminhos de concepção de um algoritmo, ou uma combinação dos mesmos, possam resultar numa solução, é o quarto caminho, o da aplicação de um método de concepção que oferece maior fiabilidade, sobretudo quando, de outro modo, não se consegue obter uma boa solução.

Devemos esclarecer que o algoritmo aqui em causa é o conjunto de acções que dá solução a um problema ou, desejavelmente, a uma categoria de problemas, no sentido de serem codificadas numa linguagem de programação de computadores. Mas convém estudar a brilhante abordagem à noção de algoritmo exposta por Donald Knuth (1968, pp. 1-9) na qual evidencia "Besides merely being a finite set of rules that gives a sequence of operations for solving a specific type of problem, an algorithm has five importante features:

1) Finiteness…
2) Definiteness…
3) Input…
4) Output…
5) Effectiveness…".

No presente artigo, o autor apresenta um método de abordagem à resolução problemas que conduz à concepção de algoritmos que integram algumas propostas enunciadas por Jacques Arsac, com o principal objectivo de auxiliar os estudantes de Programação a obterem uma disciplina de concepção de algoritmos para resolver problemas.

No ponto 2., em resposta à questão "como resolver um problema em programação?" propõe-se um método e as suas fases a resolução de problemas e justifica-se a escolha do paradigma Imperativo, para a aplicação deste método. No ponto 3., analisa-se que algoritmo será necessário para resolver um problema, usando as ferramentas de programação do modelo Imperativo,

as quais se definem e exemplificam através de formalismos algorítmicos e sua implementação em Linguagens de Programação. No ponto 4. apresenta-se a técnica da Programação Directa para a resolução de uma categoria de problemas, com um método específico detalhado e um exemplo da sua aplicação. Outra técnica, a Programação Iterativa, é ilustrada no ponto 5., com a exposição e a aplicação de um método de resolução para a categoria de problemas em que as iterações são necessárias. Após as conclusões, o autor realiza um tributo ao cientista e professor Jacques Arsac, salientando os seus contributos científicos e pedagógicos.

2. Como resolver um Problema em Programação?

Dado um Problema, para se encontrar a sua solução através da Programação, uma das dificuldades iniciais que habitualmente se revela é a da correcta interpretação do mesmo. Para obviar a esta dificuldade e disciplinar a abordagem propomos o método "Caixa Preta" (Fig. 1) de resolução de problemas em programação, com três fases:

I. Interpretar
II. Formalizar
III. Resolver

Fig. 1 Método da "Caixa Preta"

Na primeira fase, Interpretar o Problema, significa identificar o seu domínio, os resultados a obter e o que é necessário conhecer como entradas para a sua resolução.

A segunda fase, para Formalizar o Problema utilizamos a noção de "Caixa Preta" (Fig. 1), a qual se imagina conter uma resolução do Problema. Em primeiro lugar, reunimos os Resultados do Problema, a sair da Caixa Preta. Em segundo lugar, reúnem-se as Entradas necessárias para a resolução do Problema, a entrar para a Caixa Preta.

Uma vez formalizado, agora na terceira fase, podemos Resolver o Problema, construindo o Algoritmo (A) que a Caixa Preta encerra, utilizando as suas Entradas e produzindo os seus Resultados. A construção deste algoritmo tem como objectivo a sua codificação num programa de uma linguagem de programação, a fim de poder vir a ser executado pelo processador de um computador.

Porém, resolver um Problema em Programação implica saber com que paradigma da Programação se vai lidar. Dos anos 50 aos anos 80, surgiram vários paradigmas que enquadraram linguagens de programação de alto nível[1]. O mais antigo é o paradigma Imperativo (com as linguagens de programação Fortran, Cobol, Basic, Pascal, Modula, C e Ada, entre muitas outras). Seguiram-se os paradigmas Funcional (Lisp, Scheme, Haskell), Lógico (Prolog), Orientado por Objetos (Simula, Smalltalk, Eiffel) e Paralelo (OCCAM, Erlang), entre outros.

Programadores com conhecimento e domínio dos vários paradigmas da Programação podem analisar um problema e procurar a sua solução de Programação com o paradigma mais adequado. Neste contexto, fazemos a opção de considerar o paradigma Imperativo, por ter uma abordagem mais simples, acessível a um iniciado na Programação, pois neste paradigma, o programador indica a sequência de instruções que resolvem o problema, na mesma ordem que será executada pelo processador da máquina que vier a executar o programa concebido, utilizando uma memória modificável para construir resultados.

Devemos a Alan Turing, grande matemático e cientista da computação, bem estudado nas 914 páginas de "AlanTuring HIS WORK AND IMPACT" (Cooper & Leeuwen (Ed.) 2013), a definição, com apenas 24 anos de idade, de "Universal Computing Machine", a conhecida "máquina de Turing", que executa qualquer sequência computável e assim postula os princípios do paradigma Imperativo (Turing 1936).

Assim, na terceira fase, a de se saber como resolver um problema em Programação, dever-se-á Resolver o Algoritmo da Caixa Preta, tendo em conta que o mesmo será codificado numa linguagem do Paradigma Imperativo.

3. O Algoritmo da Caixa Preta

3.1. Que Algoritmo?

A construção do Algoritmo que dará solução ao Problema em causa, utilizando as Entradas e produzindo os Resultados da Caixa Preta, depende da natureza do Problema e da forma como identificarmos o seu processo de resolução.

[1] Uma Linguagem de Programação de alto nível, é uma linguagem próxima das pessoas, longe e independente do processador do computador, constituída por um subconjunto de vocábulos de uma língua falada (inglês, na maioria dos casos).

Que acções iremos utilizar no Algoritmo? Devem ser acções passíveis de ser codificadas numa linguagem de programação.

Antes de partirmos para o processo de resolução, é necessário conhecermos as ferramentas do paradigma Imperativo de Programação, pois será a partir destas que poderemos elaborar o Algoritmo.

3.2. Ferramentas de programação do paradigma Imperativo

3.2.1. Noção de Variável

No paradigma Imperativo, as soluções são construídas através de sucessivas modificações de valores de variáveis armazenadas na memória do computador, realizadas por uma sequência de instruções da linguagem de programação. Importa então saber, neste contexto, o que é uma variável e quais são as instruções.

Uma variável (Var) destina-se a receber valores e é definida[2] por cinco componentes (1), os quais a caracterizam univocamente: Nome, Género, Tipo, Valor e Localização.

```
<Var> :: = (<Nome>, <Género>, <Tipo>, <Valor>, <Localização>)
<Nome> :: = <carácter permitido> {<carácter permitido>}
<Género> :: = escalar | <tabela> | <outros>
<Tipo> :: = inteiro | real | Booleano | carácter | <outros>
<Valor> :: = <valor pertencente ao domínio do Tipo>
<Localização> :: = <endereço de memória onde está armazenada>          (1)
```

3.2.2. Instruções fundamentais do paradigma Imperativo

3.2.2.1. A Atribuição

São três as instruções fundamentais do paradigma imperativo: a "Atribuição", a "Selecção" e o "Ciclo". São ainda necessárias outras duas que permitem a leitura da entrada de dados e a escrita da saída de resultados.

A Atribuição permite a uma variável receber um valor coerente com o seu Tipo. Uma vez que lhe é atribuído um valor, este destrói qualquer outro eventual valor anterior, sendo uma instrução que se deve utilizar com responsabilidade. A sua definição sintáxica apresentada em (2) contém três partes, na maioria dos casos com a seguinte organização: o "nome da variável" à esquerda; o "símbolo de atribuição" ao centro; uma "expressão" à direita. Estas partes têm uma ordem de execução: em primeiro lugar, é calculado o valor da "expressão"; em segundo lugar, esse valor é atribuído à "variável", destruindo qualquer eventual valor que esta "variável" tenha tido anteriormente.

[2] A definição é feita com recurso à EBNF (Extended Backus-Naür Form)

<atribuição> ::= <nome de variável> <símbolo de atribuição> <expressão>
<nome de variável> ::= <carácter permitido> {<carácter permitido>}
<símbolo de atribuição> ::= ¬ | := | =
<expressão> ::= <constante> | <variável> | <função> | <expressão binária>
<expressão unária> | <expressão composta> **(2)**

Segundo esta definição, observemos exemplos de atribuições (3), seja em simbologia algorítmica (Pseudocódigo, EXEL e Fluxograma), seja codificadas em linguagens de programação (Python, Ada, C, APL).

kilos := 4	(Pseudocódigo)	kilos = 4	(Python)	
x := y	(Pseudocódigo)	x := y	(Ada)	
m ← max(a,b)	(EXEL)	m = max(a,b)	(C)	
total := total + 1	(Fluxograma)	total ← total + 1	(APL)	**(3)**

À luz das definições de Variável (1) e de Atribuição (2), após execução da Atribuição em Python, kilos = 4 (3), a variável de nome kilos fica assim definida: (kilos, escalar, inteiro, 4, <refMem>), em que <refMem> é a referência à célula de memória que arquiva o valor 4.

3.2.2.2. A Selecção

A Selecção é uma instrução que permite formular uma questão e seleccionar a resposta. Aparece geralmente nas linguagens de programação sob a designação em inglês **IF** (**SE** em português). A Selecção permite testar uma condição e decidir, face ao valor lógico desta, Verdade ou Falso, qual o conjunto de instruções a executar, segundo a definição sintáxica considerada em Pseudocódigo:

<Selecção> ::= **SE** <condição> **ENTÃO** <instruçõesV se Verdade>
SENÃO <instruçõesF se Falso>
ES **(4)**

O símbolo **ES** significa "Fim de SE".

A Selecção pode também aparecer com a possibilidade de escolha múltipla, mediante o valor de um parâmetro de opção, caso este que não trataremos neste contexto.

Seguem-se exemplos em simbologia algorítmica (Pseudocódigo, EXEL e Fluxograma) e codificação em linguagem de programação (C, Ada, Python).

Pseudocódigo:

SE nota>=10 **ENTAO AFIXAR** "Aprovado"
SENAO AFIXAR "Reprovado"
ES **(5)**

EXEL:

```
nota>=10 ? AFIXAR "Aprovado" | AFIXAR "Reprovado" ¿
```
(6)

Fluxograma:

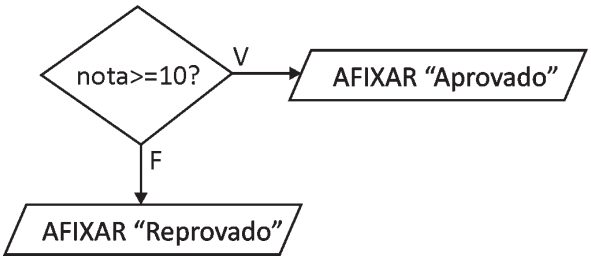

Fig. 2 Selecção SE... em Fluxograma

Linguagem C:

```
if (nota>=10)
        printf("Aprovado");
else
        printf("Reprovado");
```
(7)

Linguagem Ada:

```
if nota>=10 then
        Put_Line("Aprovado");
else
        Put_Line("Reprovado");
end if;
```
(8)

Linguagem Python:

```
if nota >= 10:
        print("Aprovado")
else:
        print("Reprovado")
```
(9)

3.2.2.3. O Ciclo

O Ciclo é uma instrução que permite fazer ou refazer um conjunto de instruções, até se atingir uma situação final, que finaliza o referido Ciclo. O número de vezes que se refaz o conjunto de instruções do ciclo deve ser sempre finito. A saída de um ciclo é feita através de uma condição, a testar de cada vez que se executa o conjunto de instruções do ciclo.

A condição a testar deve garantir que o objectivo do ciclo foi atingido, tendo por isso chegado ao fim. Uma passagem pelo conjunto de instruções a fazer ou refazer num Ciclo, designa-se de Iteração. Iterar é fazer ou refazer as instruções de um Ciclo.

O Ciclo pode apresentar variadas formas. Em (10) definimos quatro destas formas, em que o conjunto de instruções a iterar aparece diferentemente condicionado por uma condição de saída do Ciclo: FAZER, ENQUANTO, REPETIR e PARA.

```
<Ciclo> ::=  <instruções_a_refazer>

<instruções_a_refazer>::=
            <início> FAZER
                        <conj. Instruções>
                    REFAZER |

             <início> ENQUANTO <cond.Execução> FAZER
                        <conj. InstruçõesEnquanto>
                      REFAZER |

            <início> FAZER
                        <conj. InstruçõesAtéQue>
                    ATÉ_QUE <cond.Saída> REFAZER |

            PARA <iníc.ParâmetroControlo> ATÉ <cond.Saída> PASSO <intervalo>
                    FAZER
                        <conj. InstruçõesPara>
                    REFAZER                                          (10)
```

A primeira forma do Ciclo, FAZER...REFAZER, faz e refaz o <conj. Instruções> e cabe ao programador incluir neste conjunto de instruções uma condição de saída do Ciclo, para que este seja finito. A segunda forma, ENQUANTO...FAZER...REFAZER, testa à entrada do Ciclo uma condição de permanência no mesmo, a <cond. Execução>, e enquanto esta for verdadeira, continua a fazer ou refazer o <conj. InstruçõesEnquanto>. No caso contrário sai do Ciclo. A terceira forma, FAZER... ATÉ_QUE...REFAZER, faz ou refaz o <conj. InstruçõesAtéQue> até que seja Verdadeira a condição de saída, terminando assim o Ciclo. A quarta forma, PARA...FAZER... REFAZER, utiliza um parâmetro iterativo que controla o número de iterações no Ciclo, a partir dum valor inicial <iníc.ParâmetroControlo> e faz o conjunto <conj. InstruçõesPara> ATÉ ao valor desse parâmetro que na condição <cond.Saída>, por ser Verdadeira, provocar a saída do Ciclo. Se esta condição de saída resultar Falso, então o parâmetro iterativo é incrementado do valor do PASSO <intervalo> e o Ciclo é de novo refeito. Em algumas linguagens de programação, em vez

da <cond.Saída> está uma <cond.Execução> que, ao ser Verdadeira, conduz à permanência no Ciclo.

Seguem-se exemplos do Ciclo PARA...FAZER...REFAZER em notação algorítmica e em linguagens de programação.

Pseudocódigo:

```
contador=0
PARA i=0 ATÉ  i>10 PASSO 1
FAZER
        contador=contador+1
REFAZER                                                    (11)
```

EXEL:

$$\text{contador} \leftarrow 0;\ \left\{ {}^{10,1}_{i:\,0}\ \text{contador} \leftarrow \text{contador}+1 \right\} \qquad \textbf{(12)}$$

Fluxograma:

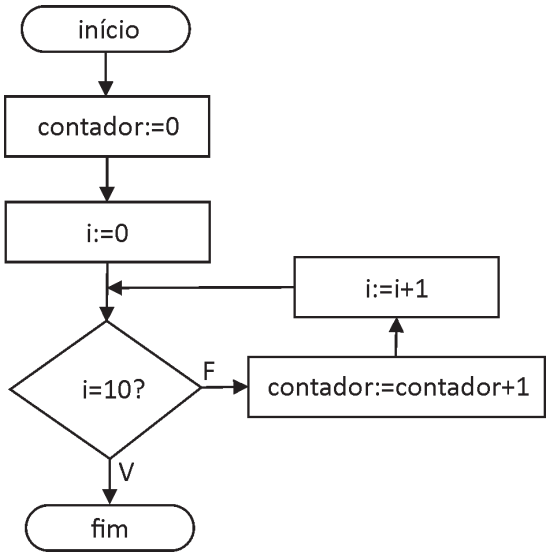

Fig. 3 Ciclo PARA... em Fluxograma

Linguagem C:

```
int contador=0;
for (int i=0; i<=10; i++)
contador=contador+1;                                       (13)
```

Linguagem Ada:

```
contador:  integer:=0;
for i in 0 .. 10 loop
        contador:=contador+1;
```
(14)

Linguagem Python:

```
contador=0
for i in range(0,10)
        contador=contador+1
```
(15)

3.3. Possíveis soluções algorítmicas

Conhecidas as ferramentas de programação que iremos utilizar, o conceito de variável e as instruções fundamentais do paradigma Imperativo, Atribuição, Selecção e Ciclo, em conjunto com as instruções de leitura de dados e afixação de resultados, observemos como estas são suficientes nas soluções algorítmicas que podemos propor, para a generalidade dos problemas que se procuram resolver.

As fases da interpretação e formulação de um Problema permitem vislumbrar qual combinação de ferramentas de Programação utilizar para a sua resolução. Nas secções seguintes, iremos abordar duas técnicas típicas de Programação para a resolução de problemas: Programação Directa e Programação Iterativa.

4. Programação Directa

Observemos uma categoria de problemas cuja solução algorítmica é conseguida com um conjunto seriado de questões, que se traduzem em Selecções ou a utilização de variáveis para se construírem os resultados, com Atribuições. Uma solução algorítmica com base em Selecções ou Atribuições dá origem à Programação Directa, na qual os resultados são obtidos directamente de uma sequência de perguntas ou de variáveis atribuídas com valores. Um excelente exemplo desta categoria de problemas é o da validação de uma data composta por dia, mês e ano, caso que é apresentado e resolvido no livro *Précettes pour programmer* de Jacques Arsac (1991, pp. 21-29), ou na tradução em português "Programação criativa" (1993a, pp. 27-36). Pretende saber-se se a data é "valida" ou "inválida".

Então apliquemos o método da Caixa Preta:

Fase I – Interpretar.

O resultado que se pretende obter na validação de uma data é, tão somente, saber-se se é "Data válida" ou "Data inválida". Para tal, partimos dos valores do dia,

do mês e do ano. Conhecemos que o dia estará compreendido entre 1 e 31, mas os dias 28, 29, 30 e 31, estão dependentes do mês, o qual por sua vez terá valores entre 1 e 12 (Janeiro e Dezembro). Em particular, os dias 28 e 29, como número de dias possíveis no mês de Fevereiro, será 29 nos anos bissextos e 28 em todos os outros;

Para saber se um ano é bissexto é necessário observar uma regra com duas excepções. É bissexto se a divisão inteira do ano por 4, tiver resto zero; a primeira excepção da divisão inteira do mesmo ano por 100, com resto zero, indica que este ano já não é bissexto; mas se na segunda excepção, a divisão inteira deste ano por 400, apresentar resto zero, afinal ele é bissexto;

Ainda há que questionar se todos os anos serão válidos. E aqui aplica-se uma afirmação que o autor ouviu o Professor Jacques Arsac proferir, variadas vezes, nas suas conferências e seminários: "Quanto mais culto se é, melhor programador se pode ser". Sabendo que o papa Gregório XIII, com a bula papal *Inter Gravissimas* modificou o calendário Juliano para o Gregoriano, em 1582. Os países do sul e centro da Europa, em 1600, já tinham adoptado este calendário, mas a Grã-Bretanha e os países nórdicos só o adoptaram entre 1700 e 1800. Assim, um programador com este conhecimento, apontará para que os anos anteriores a 1800 não sejam considerados válidos.

Fase II – Formular.

A Fig. 4 apresenta a formulação do Problema. Em primeiro lugar identificamos os Resultados: "Data válida" ou "Data inválida". Em segundo lugar as Entradas: dia, mês e ano. Por último o Algoritmo (A) que será desenvolvido na Fase seguinte.

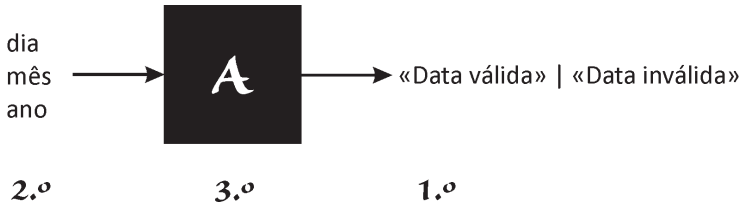

Fig. 4 Formulação do Problema "validação de uma data"

Fase III – Resolver.

Jacques Arsac (1991, 1993a) apresenta uma discussão detalhada desta resolução, onde expõe um método de resolução de problemas inspirado no "Discurso do método" de René Descartes, (com o título original "Discours de la méthode pour bien conduire sa raison, et chercher la verité dans les sciences" em 1637). O autor escolheu uma edição desta obra traduzida em português e disponível *on-line* (Descartes, 1996) em que Descartes enuncia, na pág. 23, quatro preceitos do "verdadeiro método para chegar ao conhecimento de todas as coisas".

No primeiro preceito, salientamos: "nunca aceitar alguma coisa como verdadeira sem que a conhecesse evidentemente como tal"; O segundo aponta: "dividir cada uma das dificuldades que examinasse em tantas parcelas quantas fosse possível e necessário para melhor resolvê-las"; O terceiro indica "conduzir por ordem os meus pensamentos, começando pelos objectos mais simples e mais fáceis de conhecer, para subir pouco a pouco, como degraus, até ao conhecimento dos mais compostos"; O quarto conclui "fazer em tudo enumerações tão completas, e revisões tão gerais, que eu tivesse certeza de nada omitir".

Este método enunciado no séc. XVII, é na realidade, uma excelente orientação para a resolução de Problemas em Programação. Consideramos um Problema como um todo, dividimos o todo em partes e começamos a resolução pela parte pais simples e acessível. Procedemos do mesmo modo para as restantes partes, até que o todo esteja resolvido. Verifiquemos então a aplicação do método de Descartes, na resolução do Problema enunciado da validação de uma data.

Segundo a Fase Formular, para obtermos o resultado de uma data ser válida ou inválida, temos de ter em conta as entradas do dia, mês e ano. Consideremos o problema da data como um todo que contém as partes dia, mês e ano. Qual das partes é a mais simples de abordar? Será a do ano, porque apenas temos de considerar os válidos após 1800, conforme a interpretação feita. Das outras duas, dia e mês, o mês será o mais fácil de abordar, já que bastará saber se o seu valor está entre 1 e 12, enquanto para o dia terá de se verificar o mês a que pertence e se o ano é ou não bissexto, caso o mês seja 2 (Fevereiro). Assim fica determinada a ordem da resolução das partes (Fig. 5).

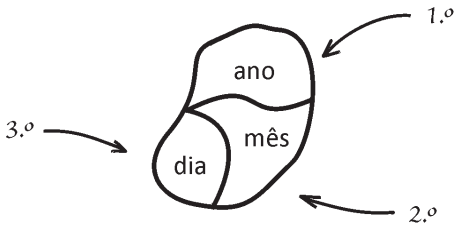

Fig. 5 As três partes do Problema "Validação de uma data"

Estamos perante um conjunto de questões para validar ou não a data.
Primeira parte: tratar o ano (a). A questão pode colocar-se de duas formas:

i. **SE** a >= 1800 **ENTAO** <verificar mês e dia> **ES**

ii. **SE** a < 1800 **ENTAO AFIXAR** "Data Inválida" **FIM ES**

Na forma i. a questão a >= 1800, coloca o ano em situação válida, mas dependente da verificação do mês e do ano, para se avaliar a possível validade da data, i. e., este começo em nada simplifica a resolução. A forma ii. coloca a questão

a < 1800 de forma a poder resolver o problema e respondendo que a data é inválida, não havendo mais que fazer e com a instrução **FIM**, finalizar o programa.

Perante estas duas opções escolhe-se a forma ii. na qual a questão conduz a uma resposta final, não a fazendo depender de outras. Esta será uma estratégia a utilizar em Programação Directa, sempre que possível. Vamos aplicá-la às três partes: ano (a), mês (m) e dia (d).

SE a < 1800 **ENTAO AFIXAR** "Data Inválida" **FIM ES**
[ano é válido]
SE m < 1 OU m > 12 **ENTAO AFIXAR** "Data Inválida" **FIM ES**
[mês é válido]
SE d < 1 OU d > durMês () **ENTAO AFIXAR** "Data Inválida" **FIM ES**
[dia é válido]
AFIXAR "Data Válida" **(16)**

Simples e muito eficaz. Eis como, com três questões, parece estar resolvido este Problema. No entanto, na questão do dia, o seu limite superior poderá ser 28, 29, 30 ou 31. Nenhum destes, *de per si*, satisfaz a questão, pois dependem do mês, e se o mês for o de Fevereiro, dependem do ano ser ou não bissexto. Por isso, como há muito a verificar, considerou-se um novo Sub-problema, uma nova Caixa Preta, intitulada duração do mês, durMês(), ao qual vamos aplicar a mesma abordagem.

Fase I – Interpretar o Sub-problema.

O Resultado será um dos dias 28, 29, 30 ou 31, sendo necessário em Entrada conhecer o mês e o ano. Os dias 30 ou 31 conhecem-se conforme os meses a que pertencerem, à excepção de Fevereiro, cujos dias 28 ou 29 dependem do ano ser ou não bissexto.

Fase II – Formular o Sub-problema.

Em primeiro lugar identificámos o Resultado, com uma das possíveis durações do mês: 28, 29, 30 ou 31. Em segundo lugar, as Entradas mês e ano, são necessárias para calcular o Resultado. O Algoritmo da Caixa Preta intitula-se durMês() (Fig. 6).

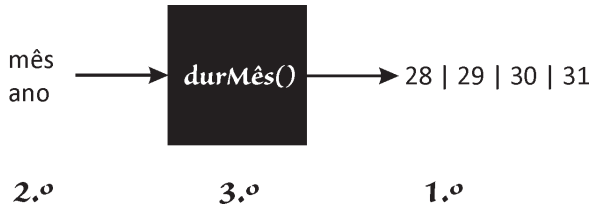

Fig. 6 Formulação do Sub-problema "duração do mês"

Fase III – Resolver o Sub-problema.

Na Fig. 7, identificamos quatro partes relativas aos dias 28, 29, 30 e 31.

A mais simples é a do dia 30, já que há menos meses dos que os de 31 para se verificar e muito menos verificações do que os de 28 e 29. Segue-se a parte da verificação dos meses com 31 dias. Por último, estamos perante o mês de Fevereiro (2), e poderemos tentar começar por identificar se o ano é não bissexto, sendo 28 a duração do mês, mas temos também que lidar com as excepções identificadas na interpretação do Problema.

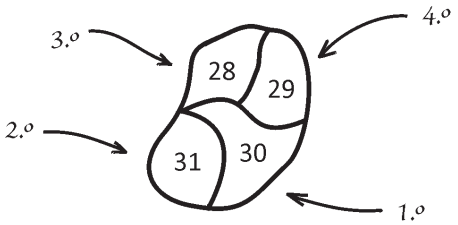

Fig. 7 As quatro partes do Sub-problema durMês()

Apliquemos o método de Descartes e a estratégia das questões que resolvem o problema sem as fazer depender de outras (17). O módulo função mod(a,b) com os parâmetros de entrada a e b, significa resto da divisão inteira de a por b.

SE m = 4 **OU** m = 6 **OU** m = 9 **OU** m = 11 **ENTAO** durMês() := 30 **FIM ES**
SE m ≠ 2 **ENTAO** durMês() := 31 **FIM ES**
SE mod(a,4) ≠ 0 **ENTAO** durMês () := 28 **FIM ES**
SE mod(a,100) ≠ 0 **ENTAO** durMês () := 29 **FIM ES**
SE mod(a,400) = 0 **ENTAO** durMês () := 29 **FIM ES**
durMês () := 28 **(17)**

Por fim, obtemos a resolução completa em **(18)**.

LER d, a, m
SE a < 1800 **ENTAO AFIXAR** "Data Inválida" **FIM ES**
SE m < 1 **OU** m > 12 **ENTAO AFIXAR** "Data Inválida" **FIM ES**
SE d < 1 **OU** d > durMês () **ENTAO AFIXAR** "Data Inválida" **FIM ES**
AFIXAR "Data Válida"

```
durMês()
INICIO_durMês
    SE m = 4 OU m = 6 OU m = 9 OU m = 11 ENTAO durMês() := 30 FIM ES
    SE m ≠ 2 ENTAO durMês() := 31 FIM ES
    SE mod(a,4) ≠  0 ENTAO durMês () := 28 FIM ES
    SE mod(a,100) ≠  0 ENTAO durMês () := 29 FIM ES
    SE mod(a,400) =  0 ENTAO durMês () := 29 FIM ES
    durMês () := 28
FIM_durMês                                                    (18)
```

Revendo a formulação do problema na Fig. 4 e a sua correspondência com **(18)**, temos: as Entradas na Caixa Preta traduzidas pela instrução **LER** d, a, m; o Algoritmo do Problema resolvido com três Selecções **SE** para tratar por ordem o ano, o mês e o dia, conforme o método de Descartes e a estratégia da escolha da Selecção que não a faz depender de outras para obter solução; e finalmente a afixação do Resultado "Data válida" ou "Data inválida" nos casos apropriados.

No Sub-problema identificado (Fig. 6), o módulo durMês() é tratado à parte, com a mesma abordagem do Problema principal, dando origem a cinco Selecções **SE** que determinam a duração do mês. No total, são oito Selecções **SE** todas independentes e seriadas de forma a se ir "peneirando" o Resultado desejado.

5. Programação Iterativa

5.1 Como construir um Ciclo?

Nos problemas em que identificamos a necessidade de recorrer variadas vezes a um mesmo conjunto de instruções para construirmos uma solução, estamos perante a possibilidade de ter uma das opções: ou utilizamos um ciclo que refaz esse conjunto de instruções até uma condição de saída, ou recorremos a uma função recursiva que se chama a si própria executando as suas próprias instruções até que uma condição de paragem da recursividade se verifique. Nesta secção, vamos abordar a programação iterativa, revendo um método de construção de ciclos que é apresentado por Jacques Arsac em (1991), ou na tradução em português "Programação criativa" (1993a, pp. 65-80).

Consideremos um problema cuja resolução irá necessitar de um ciclo.

Problema:

Um certo número de facturas devem ser pagas. Dispõe-se de um montante em euros para pagar essas facturas ou parte delas. Quantas facturas podem ser pagas com este montante?

Utilizemos o método da Caixa Preta com as três fases para a resolução deste problema.

Fase I – Interpretar.

Nesta interpretação vamos considerando o nome das variáveis que forem necessárias. Cada factura tem um valor de factura (valor_fact) em euros. A leitura destes valores faz-se a partir da factura com data mais antiga, sendo o utilizador do programa que faz essa selecção. É indicado o número máximo de facturas existentes a pagar (max_fact). O montante disponível (montante) permitirá liquidar o número de facturas a serem pagas (num_fact).

Fase II – Formalizar.

Em primeiro lugar identificamos o Resultado: o número de facturas a serem pagas (num_fact). Em segundo lugar, as Entradas: o número máximo de facturas existentes a pagar (max_fact); o montante disponível (montante); o valor em euros a ser pago, da próxima factura (valor_fact), valor este que vai sendo solicitado até que já não exista montante disponível para o pagar. O Algoritmo da Caixa Preta que trata do pagamento de facturas intitula-se pagFact() (Fig. 8).

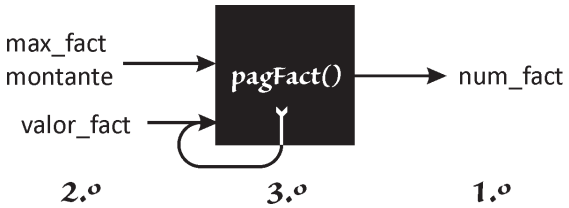

Fig. 8 Formulação do Problema "pagamento de facturas"

Fase III – Resolver.

Para calcular o num_fact, para além de inicialmente conhecermos o montante e o max_fact, necessitamos de ir lendo o valor_fact de cada factura e verificar se esta pode ser paga com o montante ainda disponível. Este processo repetitivo que é feito para cada valor de factura lido, é adequado para ser implementado com recurso a um ciclo.

Vamos construir o ciclo seguindo o referido método proposto por Jacques Arsac. Utilizaremos a linguagem algorítmica EXEL, originalmente proposta, também, por Jacques Arsac (Nolin & Ruggiu, 1973) (Silveira, 1984).

Representamos o trabalho do ciclo pelo esquema da Fig. 9.

Fig. 9 Esquema 1 da construção do ciclo – Situação geral

Caracterizamos uma situação geral que acontece no ciclo:

"Já li i-1 valores de facturas e o número de facturas que podem ser pagas está em num_fact". A partir desta situação prosseguimos em direcção ao resultado. Começamos pela questão mais simples, perguntando se chegou ao fim (Fig. 10). Se sim, terminamos o ciclo e temos o resultado em num_fact (19).

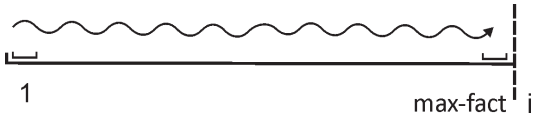

Fig. 10 Esquema 2 da construção do ciclo – Chegou ao fim?

i > max-fact **? ! | ¿** **(19)**

A Selecção **SE** em EXEL (19), contém o símbolo **!** (ponto de exclamação) que significa **FIM** do ciclo. No caso de não se ter chegado ao fim, deve avançar-se com a leitura do valor da factura i (Fig. 9) e verificar-se se ainda existe montante suficiente para a pagar. Surge a necessidade de termos uma variável total que vai acumulando os valores das facturas lidas (20), a fim de o comparar com o montante e se poder deduzir se o pagamento é viável ou não.

Ler valor_fact; total ← total + valor_fact; **(20)**

A comparação do total com montante leva a três situações:

i. Se total < montante, deve incrementar-se o valor_fact de mais uma factura que pode ser paga e continuar-se o processo de leitura;
ii. Se total = montante, deve incrementar-se o valor_fact de mais uma factura que pode ser paga e terminar-se o processo de leitura, saindo-se do ciclo, uma vez que não há mais montante disponível;
iii. Se total > montante, não pode haver mais pagamentos e sai-se do ciclo.

Como organizar estas três questões? Utilizando o método de Descartes, já descrito. Das três partes (i. ii. e iii.), começa-se pela mais simples, até à mais elaborada: primeiro iii., pois posta a questão, termina-se saindo do ciclo; segundo ii., uma vez que se incrementa o núm_fact e termina-se saindo do ciclo; terceiro i., pois além de incrementar o núm_fact, continua-se o processo de leitura incrementando o parâmetro i e refazendo o ciclo (21).

total > montante **? ! | ¿;**
total = montante **?** num_fact ← num_fact + 1; **! | ¿;**
total < montante **?** num_fact ← num_fact + 1 **| ¿;** i ← i + 1 **}** **(21)**

Reunindo as partes do algoritmo (19) (20) e (21) de forma a que no início do ciclo tenhamos a situação geral como um comentário, temos o corpo do ciclo construído (22) (o comentário é apenas informativo, não executável em programação, estando, em EXEL, indicado entre os símbolos **/*** e ***/**).

{ /*Já li i-1 valores de facturas e o número de facturas que podem ser pagas está em num_fact***/**
i > max_fact **? ! | ¿;**
Ler valor_fact; total ← total + valor_fact**;**
total > montante **? ! | ¿;**
total = montante **?** num_fact ← num_fact + 1**; ! | ¿;**
total < montante **?** num_fact ← num_fact + 1 **| ¿;** i ← i + 1 **}** **(22)**

Existem variáveis que necessitam de ser inicializadas, dado que estas não devem ser usadas pela primeira vez, sem terem valor atribuído. Como saber quais as variáveis que necessitam de ser inicializadas, i. e., atribuídas com um valor inicial? Verificamos os seguintes casos:

a) Numa Atribuição, considerando que a variável a ser atribuída (que está à esquerda do símbolo de atribuição, tal como na maioria das linguagens de programação) não necessita de ser inicializada, pois qualquer possível valor que tenha é destruído pelo novo valor que lhe vai ser atribuído, é à direita do símbolo de atribuição que se deve verificar a existência de alguma variável que esteja pela primeira vez a ser usada, tendo esta de ser inicializada;
b) Na condição de uma Selecção (instrução **SE**), se existir uma variável que esteja pela primeira vez a ser usada, esta deve ser inicializada.

Observamos então em (22) que a variável i (caso b)) e que as variáveis total e num_fact (caso a)) necessitam de ser inicializadas. As outras variáveis max_fact e montante são dadas com valores em Entrada (na Caixa Preta, Fig. 8), e o valor_fact é lido, logo não necessitam de ser inicializadas.

Por estas razões, só no fim da construção de um ciclo, conhecido o tratamento das variáveis, é que deve proceder-se à inicialização de variáveis que realmente dela necessitem, com os seus valores apropriados. Estamos, então, na posse do conhecimento necessário para realizarmos as inicializações.

Segundo o esquema 3 da construção do ciclo (Fig. 11), podemos confirmar que i deve receber o valor 1 e que num_fact, deve receber o valor 0, pois no início, pela primeira vez, ainda não foi lida factura alguma, o que pode constatar-se

Fig. 11 Esquema 3 da construção do ciclo – Início

pela veracidade da Situação Geral, com i=1, i-1= 0:

/*Já li zero (i-1=0) valores de facturas e o número de facturas que podem ser pagas está em num_fact***/**

O total deve ser atribuído com 0 por ser, pela primeira vez, o valor neutro da adição na expressão à direita da atribuição: total ← total + valor_fact. Está, então, concluída a construção do ciclo, com o algoritmo do módulo pagFact(), apresentado como um procedimento com parâmetros de entrada (↓) e saída (↑) (23), correspondentes às Entradas e Resultados da Caixa Preta da Fig. 8.

```
pagFact(↓max_fact, ↓montante, ↑num_fact):
⟦
i ← 1; num_fact ← 0; total ← 0;
{ /*Já li i-1 valores de facturas e o número de facturas que podem ser pagas está
    em num_fact*/
i >  max_fact ? ! | ¿
Ler valor_fact; total ← total + valor_fact;
total >  montante ? ! | ¿;
total =  montante ? num_fact ← num_fact + 1;  ! | ¿;
total <  montante ? num_fact ← num_fact + 1 | ¿;  i ← i + 1 }
⟧                                                                    (23)
```

Um programa principal, tem como missão, neste caso, ler os parâmetros de entrada e afixar o parâmetro de saída de pagFact() que contém o ciclo (24).

```
Programa_principal:
⟦
LER max_fact, montante; pagFact(↓max_fact, ↓montante, ↑num_fact);
AFIXAR num_fact
⟧
                                                                     (24)
```

5.2 Método para a construção de Ciclos

No ponto anterior 5.1, observámos, passo a passo, como construir um Ciclo. Sistematizemos, agora, os caminhos seguidos, em quatro passos, de um método para construção de ciclos. A abordagem a este método considera três situações

ou estados no Ciclo que representamos com um grafo, na Fig. 12:

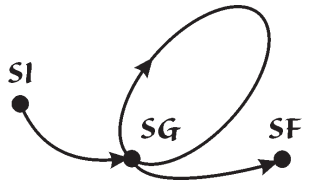

Fig. 12 Grafo com as três situações ou estados do Ciclo

- Uma Situação Geral (**SG**) que se caracteriza com uma frase sempre verdadeira no início do Ciclo, que constitui uma invariante do Ciclo, sempre válida em qualquer circunstância e que pode ser utilizada para verificação da Situação Final e da Situação Inicial;
- A Situação Final (**SF**), que se atinge quando uma condição de saída do Ciclo é verdadeira;
- A Situação Inicial (**SF**) onde se inicializam as variáveis cujos valores são necessários.

Os esquemas da Fig. 13 auxiliam a visualizar a implementação dos quatro passos deste método.

PASSO 1 – Caracterizar uma Situação Geral (SG)

Caracteriza-se uma **SG**, em três partes:

P1.1 Considerar já feita uma parte do processo de construção dos Resultados a obter com o Ciclo (na Fig. 13 (1)), representada pela flexa ondulada;

P1.2 Identificar o que foi feito e onde se está, através de uma frase descritiva (na Fig. 13 (1), a flecha ondulada representa as i-1 passagens pelo Ciclo);

P1.3 Indicar como estão os Resultados a construir (na Fig. 13 (1), representada por R).

PASSO 2 – Verificar se já se chegou ao fim (SF)

Sendo n o número de items a considerar no Ciclo, testar se o parâmetro i que conta as iterações ou passagens Ciclo ultrapassou n. No caso verdadeiro, termina-se o ciclo (na Fig. 13 (2), a flecha ondulada representa todas as passagens pelo Ciclo, com i>n). Poderá ser necessário uma outra verificação equivalente a esta, se a natureza do problema em causa não permitir conhecer o número n de items.

PASSO 3 – Aproximar a Situação Final (SF) e refazer a SG

Trata-se primeiramente das instruções da nova iteração do Ciclo, de ordem i, incrementa-se i do passo do Ciclo necessário e refaz-se de novo o Ciclo (na Fig. 13 (3)).

PASSO 4 – Inicializar variáveis na Situação Inicial (SI)

Inicializam-se as necessárias variáveis (na Fig. 13 (4), representa-se a **SI**, onde ainda não foram feitas iterações).

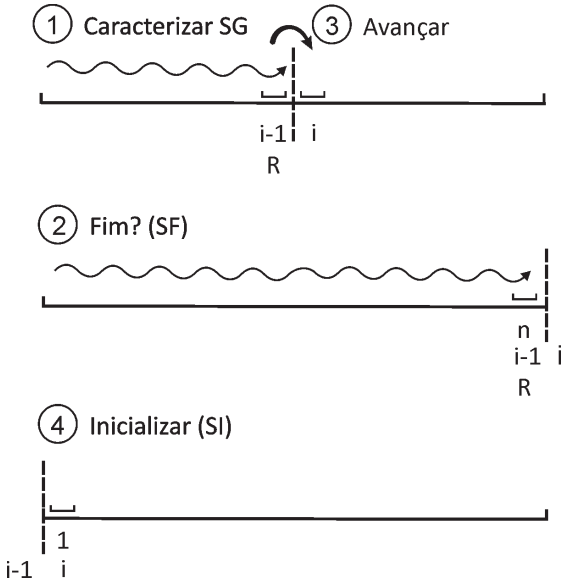

Fig. 13 Os quatro passos do método de construção de Ciclos

Está assim concluída a construção do Ciclo em quatro pontos, os quais devem agora ser revistos, observando-se a coerência deste mesmo Ciclo.

Nos casos em que, no passo 3, o desenvolvimento das instruções da nova iteração do Ciclo, de ordem i, revelar a necessidade de um novo processo iterativo, este deve considerar-se como uma nova "Caixa Preta" contendo a resolução do mesmo, prosseguindo-se de imediato para o passo 4, a fim de se concluir o actual Ciclo. A nova "Caixa Preta" identificada, será então resolvida como um sub-problema, com solução iterativa, pelo mesmo método de construção de Ciclos. Estamos perante uma solução em que um Ciclo contém outro Ciclo, um caso típico existente no tratamento de matrizes.

6. Conclusões

Pode considerar-se que a Ciência da Programação - acumulada a experiência de mais de seis décadas de programação de computadores - trouxe uma enorme contribuição para a resolução de Problemas através da Programação. No entanto, mantêm-se dificuldades da Programação, quando se trata de se

encontrar uma solução para um Problema, mais ainda, particularmente, quando se está a aprender a programar e a experiência não abunda. Estas dificuldades manifestam-se logo ao início, quando se aborda a resolução de um Problema. Que fazer? Que caminhos seguir? Este artigo veio dar um contributo no sentido de reunir e propor métodos que propocionam, de forma faseada, a construção de algoritmos que permitem implementar as soluções desejadas.

De entre os vários paradigmas da Programação, justificou-se a escolha do Imperativo, no qual se definiram as suas ferramentas fundamentais e se apresentaram métodos para resolver categorias de Problemas com diferentes técnicas: Programação Directa, com Atribuições e Selecções e Programação Iterativa, com ciclos. O autor propôs o método da Caixa Preta, o qual estrutura a análise do Problema para a construção do Algoritmo que o resolve. Foram descritos: o método de Descartes para resolução de um Problema, a partir das suas partes; e o método de construção de ciclos para a Programação Iterativa. Foi ilustrada a aplicação destes métodos, nas técnicas abordadas de Programação Directa e Iterativa, mostrando-se, passo a passo, exemplos da aplicação dos referidos métodos.

Tendo o Professor Jacques Arsac desempenhado um papel fundamental na criação de alguns dos métodos expostos no presente artigo, o autor dedica-lhe, na secção final, um especial tributo, onde refere as suas contribuições científicas, académicas e, em particular, as pedagógicas.

Tributo a Jacques Arsac

O autor conheceu o Professor Jacques Arsac (n. 1 de Fevereiro de 1929 – f. 14 de Janeiro de 2014) nas décadas de 80 e 90 e com ele conviveu, enquanto este ministrou cursos e seminários de Programação, inicialmente a convite do Professor Rui Agonia Pereira, no ISLA de Lisboa e de Santarém e, mais tarde, a convite do próprio autor, na Universidade dos Açores, e ainda quando realizou estudos em Paris, onde teve a honra de o ter como presidente do júri da sua tese de doutoramento, apresentada na Université Pierre et Marie Curie, Paris 6, (Silveira, 1987).

Jacques Arsac transmitia sempre a sua preocupação em encontrar as melhores formas de ensinar Programação. Em 1984, organizou uma conferência patrocinada pela IFIP - International Federation for Information Processing, sobre o tema *The role of Programming in Teaching Informatics*, durante a qual, na sua intervenção como orador principal, *Teaching programming* (Arsac, 1984), manifestou esse seu sentir, ao afirmar que a principal actividade em programação seria a da concepção de um programa correcto para resolver um problema. Podemos observar estas suas convicções num artigo intitulado *Enseigner l'Informatique* (Arsac, 2001), disponível *online*.

Essa sua preocupação foi bem acolhida pelo governo francês que, em 1980, criou a opção de Informática nos liceus e o nomeou responsável pela missão de inspecção geral dessa opção, entre 1987 e 1991, e onde ele cuidou de aplicar estes princípios no início do ensino da programação.

Nos seus cursos e seminários, Jacques Arsac fazia notar as dificuldades da Programação na resolução de problemas e proporcionava abordagens possíveis para soluções, procurando indicar caminhos típicos de resolução. Neste aspecto, foi inovador, propondo várias reflexões e soluções em Programação, em particular, métodos de Programação para diferentes técnicas de solução, Programação Directa, Programação Iterativa, Programação Recursiva e transformação de programas recursivos em iterativos, que foi apresentando em vários livros (Arsac, 1977, 1980, 1983, 1985a, 1991, 1993a). Nas suas aulas, tinha o cuidado de aliar a parte expositiva teórica dos algoritmos que construía para resolver problemas, à implementação imediata em linguagem de programação,

nomeadamente com LSE – *Langage Symbolique pour l'Enseignement*, com vocabulário e sintaxe em francês, cujo sistema o próprio Arsac reescreveu em 1983 (Arsac, 1985b) e cujo historial foi recentemente elaborado por Jacques Baudé (2015). O autor, em colaboração com Jacques Arsac, contribuiu para a edição, com vocabulário e sintaxe em português, desta linguagem, a LSE93 (Silveira & Arsac, 1993) (Silveira, 1993).

Jacques Arsac era um apaixonado da interpretação das coisas, da exploração do sentido delas. O autor acompanhou o Professor Arsac à Universidade dos Açores e, nessa viagem, teve a oportunidade de com ele abordar este tema da interpretação, não só na Programação, mas também noutras áreas, nomeadamente, na música de Ravel, a qual ambos apreciavam, bem como de outros aspectos do sentido das coisas que desenvolveu no seu livro "La science et le sens de la vie" (1993b).

Interessava-se pela epistemologia da informática. Publicou *Les machines à penser* (1987), onde defende a tese de que o homem nunca será substituído por uma inteligência artificial. Era um homem de profunda fé. Em Jaques *Arsac, un informaticien. Entretien avec Jaques Vauthier* (Arsac & Vauthier, 1989) expressa que é a experiência da sua fé que lhe permite afirmar que há qualquer coisa para lá da ciência, ideia que desenvolve, após ter-se dedicado ao estudo da filosofia, que resultou na sua última obra *Y a-t-il une vérité hors de la science? Un scientifique s'aventure en philosophie* (2002).

Jacques Arsac, físico de formação, desenvolveu estudos de radioastronomia, nomeadamente aplicando "transformadas de Fourier" nas imagens deformadas dos anéis de Saturno, obtidas por telescópio. Em 1959, conseguiu que comprassem um computador IBM 650 para o Centro de Cálculo do Observatório de Paris-Meudon, que fundou, nesse ano, e que dirigiu até 1965. Arsac salientou que até 1958 era muito difícil trabalhar em radio-interferometria, onde aplicava as "transformadas de Fourier" (1961), as quais eram calculadas à mão, durante dias. Com a chegada do computador, a programação destes cálculos constituiu uma franca evolução.

Desde 1956, começou a orientar a sua investigação para a Informática. Em 1963, é convidado a ensinar Programação no então novo Institut de Programmation em Paris e em 1965, é nomeado responsável pela cadeira de Programação na Faculdade de Ciências de Paris onde, em 1966, participa na criação de cursos de Informática - *Maîtrises d'Informatique* - e, com Jean Claude Simon, cria o "troisième cycle" de Informática na Université Pierre et Marie Curie permitindo, deste modo, as teses de doutoramento especificamente em Informática. Em 1967, é nomeado director do Institut de Programmation, no qual, durante quatro anos, contribui para a formação de milhares de informáticos, através de cursos conferentes de diploma. Publicou a obra *La science informatique* (1970).

Jacques Arsac foi professor convidado na Universidade de Montréal, em 1970-71. Em 1974, criou na Universidade de Paris 6, a *Equipe de Recherche Associée au* C.N.R.S.- Centre National de la Recherche Scientifique, de *Systèmes Informatiques* e, em 1976, foi, juntamente com Maurice Nivat, co-responsável

do LIPT- Laboratoire d'Informatique Théorique et Programmation, associado ao C.R.N.S. e comum às Universidades Paris 6 e 7. Foi eleito *correspondant* de l'Académie des Sciences de l'Institut de France, em 1980. Em 1991, veio a ser *Professeur émérite* na Université Pierre et Marie Curie, Paris 6.

Recebeu numerosas distinções e prémios pela sua carreira, nomeadamente, foi *Commandeur de l'Ordre national de mérite e Officier des palmes académiques*. O LIP6 – Laboratoire d'Informatique de Paris 6, no seu site, termina uma nota que dedica ao Professor Jacques Arsac, salientando: "De nombreux ingénieurs, docteurs, enseignants et chercheurs doivent beaucoup à la qualité de ses enseignements, à la profondeur des vues sur l'informatique et à sa mobilisation efficace pour l'essor de cette science."

O autor deixa este tributo[3] público ao cientista e ao amigo e extraordinário Professor Jacques Arsac, de quem recebeu um valioso "testemunho" - que vem passando a milhares de alunos - nos últimos trinta anos, ao longo dos quais tem lecionado em Portugal e em França.

Agradecimento

O autor agradece ao Professor Rui Agonia Pereira as oportunidades que providenciou no acesso ao conhecimento do Professor Jacques Arsac, através dos cursos e seminários que este ministrou. Está também reconhecido pelas inúmeras conversas científicas que, ao longo de muitos anos, ambos vêm dedicando ao tema da Programação, incluindo o presente artigo na sua fase de *working paper*.

Bibliografia

Academie des Sciences de l'Institut de France. (s.d.). Notice biographique de Jacques Arsac, Correspondant de l'Académie des sciences. Obtido a 8/07/2016, de http://www.academie-sciences.fr/pdf/membre/ArsacJ_bio0809.pdf.

Arsac, J. (1961). *Transformation de Fourier et théorie des distributions*. Paris: Dunod. Tradução em inglês (Prentice Hall, 1966).

Arsac, J. (1970) *La science informatique*. Paris: Dunod

Arsac, J. (1977). *La construction de programmes structurés*. Paris: Dunod.

Arsac, J. (1980). *Premières leçons de programmation*. Paris: Cedic.

Arsac, J. (1983). *Les bases de la programmation*. Paris: Dunod.

Arsac, J. (1984). Teaching programming. In *The role of Programming in Teaching Informatics, IFIP Working Conference* (organizada por J. Arsac). Elsevier Science Publishers, North Holland, 3-6.

[3] No presente tributo são utilizadas informações do site Academie des Sciences de l'Intitut de France (s.d.) e do site LIP6-Laboratoire d'Informatique de Paris 6 (s.d.).

Arsac, J. (1985a). *Jeux et casse-tête à programmer*. Paris: Dunod.
Arsac, J. (1985b). LSE 83. In *Bulletin de l'EPI*, no 38, Junho 1985, 116–137. Obtido a 8/07/2016, de http://www.epi.asso.fr/fic_pdf/b38p116.pdf.
Arsac, J. (1987). *Les machines à penser*. Le Seuil.
Arsac, J., & Vauthier, J. (1989). *Jacques Arsac, un informaticien: Entretien avec Jacques Vauthier*. Paris: Beauchesne.
Arsac, J. (1991). *Préceptes pour Programmer*. Paris: Dunod.
Arsac, J. (1993a). *Programação Criativa, Preceitos e Instrumentos*. T. L. Castro (Ed.), Mem Martins, Portugal: CETOP. Prólogo de P. E. Silveira.
Arsac, J. (1993b). *La science et le sens de la vie*. Fayard.
Arsac, J. (2001). Enseigner l'Informatique. In *Revue de l'EPI n° 104*, Septembre 2001, 175-185. Obtido a 8/07/2016, de http://www.epi.asso.fr/fic_pdf/ba4p175.pdf.
Arsac, J. (2002). *Y a-t-il une vérité hors de la science*? Un scientifique s'aventure en philosophie. L'Harmattan.
Baudé, J. (2015). Le système LSE par Jacques Baudé, In *Bulletin de la Société Informatique de France 1024 n°7* Novembro 2015, 41-56. Obtido a 2016/07/08, de http://www.societe-informatique-de-france.fr/wp-content/uploads /2015 /12/1024-no7-Baude.pdf.
Cooper, S. B. & Leeuwen, . B. (Ed.). (2013). *Alan Turing HIS WORK AND IMPACT*. Elsevier. Obtido em 17/07/2016, de http://oecdinsights.org/wp-content/uploads/2012/06/Alan-Turing.pdf
Descartes, R. (1996). Discurso do método. (2.ª ed.) (M. E. Galvão, Trad.) São Paulo: Martins Fontes. (Obra original de 1637). Obtido a 9/06/2016, de http://www.josenorberto.com.br/DESCARTES_Discurso_do_m%C3%A9todo_Completo.pdf.
Dijkstra, E. (1976). *A Discipline of Programming*. Prentice Hall.
Greis, D. (1981). *The Science of Programming*. New York: Springer Verlag.
Knuth, D. (1968). *The art of computer programming: Fundamental Algorithms* (1st. Ed. 1968, 3rd. Ed. 1997) vol. 1, Addison Wesley.
LIP6 Laboratoire d'Informatique de Paris 6. (s.d.). Décès de Jacques Arsac. Obtido a 8/07/2016, de https://www.lip6.fr/actualite/information-fiche.php?ident=OL79.
Nolin, L. & Ruggiu, G. (1973). Formalization of EXEL. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages* (POPL '73). ACM, New York, NY, USA, 108-119. DOI=http://dx.doi.org/10.1145/512927.512937
Petkovšek, M., Wilf, H. S. & Zeilberger, D. (1996). *A=B*. Wellesley, Massachusetts: A K Peters. Prefácio de D. Knuth.
Silveira, P. E. (1984). EXEL - *Uma linguagem algorítmica de elevado nível no domínio Brainware*. Universidade dos Açores. Ponta Delgada. p. 42
Silveira, P. E. (1987). *Structuration Dynamique des Connaissances: PLAN/ PROJET/ ACTION*. Thèse de doctorat de 1'Université Pierre et Marie Curie, Paris 6, p. 443.

Silveira, P. E. (1993). *Lições de LSE93, com exercícios resolvidos*. Tradução do original LSE de J. Arsac, CIC-ISLA, Lisboa: CIC-Centro de Investigação e Cálculo do ISLA, p. 94.

Silveira, P. E. & Arsac, J. (1993). LSE93 - *Linguagem Simples para o Ensino, sistema em versão portuguesa*, suporte disquete 3.5. Lisboa: CIC-Centro de Investigação e Cálculo do ISLA.

Turing, A. (1936). On Computable Numbers, with an application to the Entscheidungsproblem. In *Proc. Lond. Math. Soc.* (2) 42 230-265. Obtido a 17/07/2016, de https://www.cs.virginia.edu/~robins/Turing_Paper_1936.pdf; Correction ibid. 43, 544-546 (1937).