Capítulo 2

Evolução das Principais Linguagens de Programação

Este capítulo descreve o desenvolvimento de uma coleção de linguagens de programação, explorando o ambiente no qual cada uma delas foi projetada e focando nas contribuições da linguagem e na motivação para seu desenvolvimento. Descrições gerais de linguagens não são incluídas; em vez disso, discutimos alguns dos novos recursos introduzidos por cada linguagem. De interesse, em particular, estão os recursos que mais influenciaram linguagens subsequentes ou o campo da ciência da computação.

Este capítulo não inclui uma discussão aprofundada de nenhum recurso ou conceito de linguagem; isso é deixado para os seguintes. Explanações breves e informais de recursos serão suficientes para nossa jornada pelo desenvolvimento dessas linguagens.

O capítulo discute uma ampla variedade de linguagens e conceitos que não serão familiares a muitos leitores. Aqueles que acharem isso um obstáculo podem preferir postergar a leitura deste capítulo até que o resto do livro tenha sido estudado.

A escolha sobre quais linguagens discutir foi subjetiva, e alguns leitores irão notar de maneira desapontada a ausência de uma ou mais de suas linguagens favoritas. Entretanto, para manter essa cobertura histórica em um tamanho razoável, foi necessário deixar de fora algumas linguagens que alguns apreciam muito. As escolhas foram feitas baseadas em nossas estimativas da importância de cada uma para o desenvolvimento das linguagens e para o mundo da computação como um todo. Também incluímos descrições breves de outras linguagens referenciadas mais tarde.

Ao longo do capítulo, as versões iniciais das linguagens são geralmente discutidas em ordem cronológica. Entretanto, versões subsequentes das linguagens aparecem com sua versão inicial, em vez de em seções posteriores. Por exemplo, o Fortran 2003 é discutido na seção com o Fortran I (1956). Além disso, em alguns casos, linguagens de importância secundária relacionadas a uma linguagem que tem sua própria seção aparecem naquela seção.

O capítulo inclui listagens de 14 programas de exemplo completos, cada um em uma linguagem diferente. Nenhum deles é descrito neste capítulo; o objetivo é simplesmente ilustrar a aparência de programas nessas linguagens. Leitores familiarizados com qualquer uma das linguagens imperativas comumente utilizadas devem ser capazes de ler e entender a maioria do código desses programas, exceto aqueles em LISP, COBOL e Smalltalk (o exemplo de LISP é discutido no Capítulo 15). O mesmo problema é resolvido pelos programas em Fortran, ALGOL 60, PL/I, BASIC, Pascal, C, Perl, Ada, Java, JavaScript e C#. Note que a maioria das linguagens contemporâneas nessa lista oferece suporte para vetores dinâmicos, mas devido à simplicidade do problema de exemplo, não as utilizamos nos programas de exemplo. Além disso, no programa do Fortran 95, evitamos usar os recursos que poderiam ter evitado o uso de laços completamente, em parte para manter o programa simples e legível e em parte apenas para ilustrar a estrutura básica de laços da linguagem. A Figura 2.1 é um gráfico da genealogia das linguagens de alto nível discutidas neste capítulo.

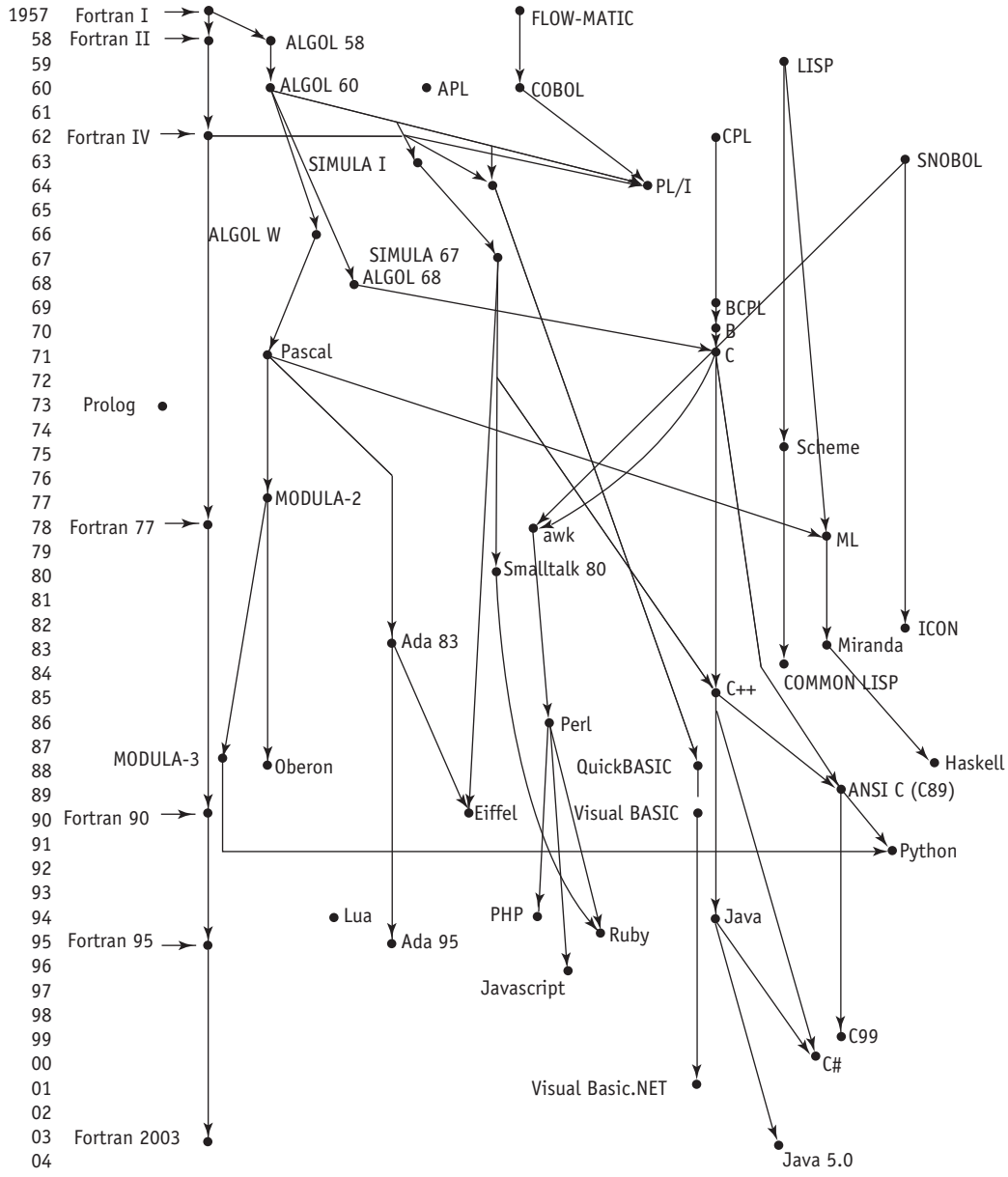

Figura 2.1 Genealogia das principais linguagens de programação de alto nível.

2.1 PLANKALKÜL DE ZUSE

A primeira linguagem de programação discutida neste capítulo é altamente não usual em diversos aspectos. Primeiro, ela nunca foi implementada. Segundo, apesar de ter sido desenvolvida em 1945, sua descrição não foi pu-

blicada até 1972. Como poucas pessoas estavam familiarizadas com a linguagem, algumas de suas capacidades não apareceram em outras até 15 anos após seu desenvolvimento.

2.1.1 Perspectiva histórica

Entre 1936 e 1945, o cientista alemão Konrad Zuse construiu uma série de computadores complexos e sofisticados a partir de relés eletromecânicos. No início de 1945, a guerra havia destruídos todos, exceto um de seus últimos modelos, o Z4, então ele se mudou para um vilarejo na Bavária chamado Hinterstein, e os membros de seu grupo de pesquisa se separaram.

Trabalhando sozinho, Zuse embarcou em uma jornada para desenvolver uma linguagem para expressar computações para o Z4, um projeto iniciado em 1943 como proposta de sua tese de doutorado. Ele chamou essa linguagem de Plankalkül, que significa cálculo de programas. Em um extenso manuscrito datado de 1945, mas não publicado até 1972 (Zuse, 1972), Zuse definiu Plankalkül e escreveu algoritmos na linguagem para uma ampla variedade de problemas.

2.1.2 Visão geral da linguagem

Plankalkül era extraordinariamente completa, com alguns de seus recursos mais avançados na área de estruturas de dados. O tipo de dados mais simples em Plankalkül era o bit. Tipos numéricos inteiros e de ponto flutuante eram construídos a partir do tipo bit. O tipo ponto flutuante usava a notação de complemento de dois e o esquema de "bit oculto" atualmente usado para evitar armazenar o bit mais significativo da parte normalizada da fração de um valor de ponto flutuante.

Além dos tipos escalares usuais, Plankalkül incluía vetores e registros. Os registros podiam incluir registros aninhados. Apesar de a linguagem não ter um comando de desvio incondicional (*goto*) explícito, ela incluía uma sentença iterativa similar ao `for` de Ada. Ela também tinha o comando `Fin` com um índice que especificava um salto a partir de um número de aninhamentos de iterações de um laço ou para o início de um novo ciclo iterativo. Plankalkül incluía uma sentença de seleção, mas não permitia o uso de uma cláusula do tipo senão (*else*).

Um dos recursos mais interessantes dos programas de Zuse era a inclusão de expressões matemáticas mostrando os relacionamentos atuais entre variáveis de programas. Essas expressões informavam o que deveria ser verdadeiro durante a execução nos pontos de código em que apareciam. Isso é bastante similar ao uso de asserções em Java e em semântica axiomática, discutida no Capítulo 3.

O manuscrito de Zuse continha programas de complexidade muito maior do que qualquer outro escrito antes de 1945. Estavam incluídos programas para ordenar vetores de números; testar a conectividade de um dado grafo; realizar operações de inteiros e de ponto flutuante, incluindo a raiz quadrada; e realizar análise sintática em fórmulas lógicas que tinham parên-

teses e operadores em seis níveis diferentes de precedência. Talvez o mais excepcional fossem suas 49 páginas de algoritmos para jogar xadrez, um jogo no qual ele não era um especialista.

Se um cientista da computação tivesse encontrado a descrição de Zuse de Plankalkül no início dos anos 50, o único aspecto da linguagem que poderia ser um obstáculo para sua implementação conforme a definição seria sua notação. Cada sentença consistia em duas ou três linhas de código. A primeira linha era bastante parecida com as sentenças das linguagens correntes. A segunda, opcional, continha os índices das referências a vetores na primeira. O mesmo método para indicar índices era usado por Charles Babbage em programas para sua Máquina Analítica no meio do século XIX. A última linha de cada sentença em Plankalkül continha os nomes dos tipos para as variáveis mencionadas na primeira linha. Essa notação é bastante intimidadora vista pela primeira vez.

A seguinte sentença de atribuição de exemplo, a qual atribui o valor da expressão `A[4] + 1` para `A[5]`, ilustra essa notação. A linha rotulada `V` é para os índices, e a `S` é para os tipos de dados. Nesse exemplo, `1.n` significa um inteiro de *n* bits:

```
  | A + 1 => A
V | 4       5
S | 1.n     1.n
```

Podemos apenas especular sobre a direção que o projeto de linguagens de programação teria tomado se o trabalho de Zuse tivesse sido bastante conhecido em 1945 ou até mesmo em 1950. É também interessante considerar como seu trabalho teria sido diferente se ele o tivesse feito em um ambiente pacífico, cercado por outros cientistas, em vez de na Alemanha em 1945, praticamente isolado.

2.2 PROGRAMAÇÃO DE HARDWARE MÍNIMA: PSEUDOCÓDIGOS

Primeiro, note que a palavra *pseudocódigo* é usada neste capítulo com um sentido diferente de seu significado contemporâneo. Chamamos as linguagens discutidas nesta seção de pseudocódigos porque esse era o seu nome na época em que foram desenvolvidas e usadas (final dos anos 1940 e início dos anos 1950). Entretanto, elas não são pseudocódigos no sentido atual da palavra.

Os computadores que se tornaram disponíveis no final dos anos 1940 e no início dos anos 1950 eram muito menos usáveis do que os de atualmente. Além de lentos, não confiáveis, caros e com memórias extremamente pequenas, as máquinas daquela época eram difíceis de programar por causa da falta de software de suporte.

Não existiam linguagens de programação de alto nível, nem mesmo linguagens de montagem, então a programação era feita em código de máquina, o que era tanto tedioso quanto passível de erros. Dentre os proble-

mas, existia o uso de códigos numéricos para especificar instruções. Por exemplo, uma instrução ADD poderia ser especificada pelo código 14 em vez de por um nome textual conotativo, mesmo se fosse composto por apenas uma única letra. Isso faz com que os programas sejam de difícil leitura. Um problema mais sério era o endereçamento absoluto, que tornava as modificações de programas tediosas e passíveis de erros. Por exemplo, suponha que tenhamos um programa em linguagem de máquina armazenado na memória. Muitas das instruções se referem a outras posições dentro do programa, normalmente para referenciar dados ou indicar os alvos de instruções de desvio. Inserir uma instrução em qualquer posição do programa em outra que não o final desse invalida a corretude de todas as instruções que referenciam endereços além do ponto de inserção, pois esses endereços devem ser incrementados para que exista espaço para a nova instrução. Para fazer a adição corretamente, todas as instruções que referenciam endereços após a adição devem ser encontradas e modificadas. Um problema similar ocorre com a exclusão de uma instrução. Nesse caso, entretanto, as linguagens de máquina geralmente incluem uma instrução "sem operação" que pode substituir instruções excluídas, evitando o problema.

Esses são os problemas padrão com as linguagens de máquina e serviram de motivação para a invenção de montadores e de linguagens de montagem. Além disso, a maioria dos problemas de programação da época era numérica e requeria operações aritméticas de ponto flutuante e de indexação de algum tipo para permitir a conveniência do uso de vetores. Nenhuma dessas capacidades, entretanto, estava incluída na arquitetura dos computadores do final dos anos 1940 e início dos anos 1950. Essas deficiências levaram ao desenvolvimento de linguagens de um nível mais alto.

2.2.1 Short code

A primeira de tais novas linguagens, chamada de Short Code, foi desenvolvida por John Mauchly em 1949 para o BINAC, um dos primeiros computadores eletrônicos com programas armazenados bem-sucedidos. Short Code foi posteriormente transferida para um UNIVAC I (o primeiro computador eletrônico comercial vendido nos Estados Unidos) e, por diversos anos, era uma das principais maneiras de programar essas máquinas. Apesar de pouco ser conhecido sobre a linguagem Short Code original, já que sua descrição completa nunca foi publicada, um manual de programação para o UNIVAC I sobreviveu (Remington-Rand, 1952). É seguro assumir que as duas versões eram bastante similares.

As palavras da memória do UNIVAC I tinham 72 bits, agrupados como 12 bytes de seis bits cada. A linguagem Short Code era composta de versões codificadas de expressões matemáticas que seriam avaliadas. Os códigos eram valores de pares de bytes e muitas equações podiam ser codificadas em uma palavra. Alguns dos códigos de operação eram

```
01  -     06  abs value    1n  (n+2)nd power
02  )     07  +            2n  (n+2)nd root
```

```
03  =     08  pause        4n  if <= n
04  /     09  (            58  print and tab
```

As variáveis eram nomeadas com códigos de pares de bytes, assim como os locais a serem usados como constantes. Por exemplo, `X0` e `Y0` poderiam ser variáveis. A sentença

```
X0 = SQRT(ABS(Y0))
```

seria codificada em uma palavra como `00 X0 03 20 06 Y0`. O `00` inicial era usado como um espaçamento para preencher a palavra. Um fato interessante é que não existia um código para a multiplicação; ela era indicada apenas pela simples colocação de dois operandos um ao lado do outro, como na álgebra.

Os programas em Short Code não eram traduzidos para código de máquina. Em vez disso, a linguagem era implementada com um interpretador puro. Na época, esse processo era chamado de *programação automática*. Ele simplificou a programação, mas ao custo do tempo de execução – a interpretação de programas em Short Code era aproximadamente 50 vezes mais lenta do que a execução de código de máquina.

2.2.2 Speedcoding

Em outros lugares, sistemas de interpretação estavam sendo desenvolvidos para estender linguagens de máquina para incluir operações de ponto flutuante. O sistema Speedcoding desenvolvido por John Backus para o IBM 701 é um exemplo (Backus, 1954). O interpretador Speedcoding efetivamente convertia o 701 para uma calculadora virtual de ponto flutuante de três endereços. O sistema incluía pseudoinstruções para as quatro operações aritméticas em dados de ponto flutuante, assim como operações como a raiz quadrada, seno, arco tangente, exponenciação e logarítmo. Desvios condicionais e incondicionais e conversões de entrada e saída também faziam parte da arquitetura virtual. Para se ter uma ideia das limitações de tais sistemas, considere que a memória usável restante após carregar o interpretador era de apenas 700 palavras e que a instrução de adição levava 4,2 milissegundos para ser executada. Em contrapartida, a linguagem Speedcoding incluía a inédita facilidade para incrementar os registradores de endereço automaticamente. Essa facilidade não apareceu em hardware até os computadores UNIVAC 1107 em 1962. Por causa desses recursos, a multiplicação de matrizes poderia ser feita em 12 instruções Speedcoding. Backus afirmava que problemas que levariam duas semanas para serem programados em código de máquina poderiam ser programados em poucas horas usando Speedcoding.

2.2.3 O sistema de "compilação" da UNIVAC

Entre 1951 e 1953, uma equipe liderada por Grace Hopper na UNIVAC desenvolveu uma série de sistemas de "compilação" nomeados A-0, A-1 e A-2 que expandiam um pseudocódigo em subprogramas em código de máquina da

mesma maneira que as macros são expandidas em linguagem de montagem. O código fonte do pseudocódigo para esses "compiladores" era ainda muito primitivo, apesar de mesmo isso ser uma grande melhoria em relação ao código de máquina, pois fazia com que os programas fonte fossem muito menores. Wilkes (1952), independentemente, sugeriu um processo similar.

2.2.4 Trabalhos relacionados

Outras maneiras de facilitar a tarefa de programação estavam sendo desenvolvidas mais ou menos na mesma época. Na Universidade de Cambridge, David J. Wheeler (1950) desenvolveu um método de usar blocos de endereços realocáveis para resolver parcialmente o problema do endereçamento absoluto, e posteriormente, Maurice V. Wilkes (também em Cambridge) estendeu a ideia de projetar um programa em linguagem de montagem que poderia combinar sub-rotinas escolhidas e alocar armazenamento (Wilkes et al., 1951, 1957). Esse era, na verdade, um avanço importante e fundamental.

Devemos também mencionar que as linguagens de montagem, bastante diferentes dos pseudocódigos mencionados, evoluíram durante o início dos anos 1950. Entretanto, tiveram pouco impacto no projeto de linguagens de alto nível.

2.3 O IBM 704 E FORTRAN

Certamente um dos maiores avanços na computação veio com a introdução do IBM 704 em 1954, em grande parte porque suas capacidades levaram ao desenvolvimento do Fortran. Pode-se argumentar que, se não fosse a IBM com o 704 e o Fortran, logo seria outra organização com um computador similar e uma linguagem de alto nível relacionada. Entretanto, a IBM foi a primeira a ter tanto a visão quanto os recursos para bancar tais avanços.

2.3.1 Perspectiva histórica

Uma das principais razões pelas quais a lentidão dos sistemas de interpretação era tolerada no final da década de 1940 e até meados da década de 1950 era a falta de hardware de ponto flutuante nos computadores disponíveis. Todas as operações de ponto flutuante teriam de ser simuladas em software, um processo que consumia muito tempo. Como muito tempo do processador era gasto no processamento de software para ponto flutuante, a sobrecarga da interpretação e a simulação de indexação eram relativamente insignificantes. Enquanto as operações de ponto flutuante tivessem de ser feitas via software, a interpretação era uma despesa aceitável. Entretanto, muitos programadores da época nunca usaram sistemas de interpretação, preferindo a eficiência do código de linguagem máquina (ou de montagem) escrito à mão. O anúncio do sistema IBM 704, contendo tanto indexação quanto instruções de ponto flutuante em hardware, decretaram o fim da era de interpretação, ao menos para a computação científica. A inclusão de hardware de ponto flutuante removeu o esconderijo para o custo da interpretação.

Apesar de o Fortran levar o crédito de ser a primeira linguagem de alto nível compilada, a questão sobre quem merece o crédito por implementar a primeira linguagem desse tipo permanece aberta. Knuth e Pardo (1977) creditam a Alick E. Glennie, por seu compilador Autocode para o computador Manchester Mark I. Glennie desenvolveu o compilador em Fort Halstead, no Royal Armaments Research Establishment, na Inglaterra. O compilador ficou operacional em setembro de 1952. Entretanto, de acordo com John Backus (Wexelblat, 1981, p. 26), o Autocode de Glennie era tão baixo nível e orientado à máquina que ele não poderia ser considerado um sistema compilado. Backus dá o crédito a Laning e Zierler do Instituto de Tecnologia de Massachusetts (MIT).

O sistema de Laning e Zierler (Laning and Zierler, 1954) foi o primeiro de tradução algébrica a ser implementado. Por algébrica, queremos dizer que o sistema traduzia expressões aritméticas, usava subprogramas codificados separadamente para computar funções transcendentais (por exemplo, seno e logarítmo) e incluía vetores. O sistema foi implementado no computador Whirlwind do MIT, como um protótipo experimental, no verão de 1952, e em uma forma mais usável em maio de 1953. O tradutor gerava uma chamada a sub-rotina para codificar cada fórmula, ou expressão, no programa. A linguagem fonte era fácil de ler, e as instruções de máquina que foram incluídas eram aquelas para desvios. Apesar de esse trabalho preceder o trabalho no Fortran, ele nunca saiu do MIT.

Apesar desses trabalhos anteriores, a primeira linguagem de alto nível compilada de ampla aceitação foi o Fortran. As subseções seguintes descrevem esse importante avanço.

2.3.2 Processo do projeto

Mesmo antes de o sistema 704 ser anunciado em maio de 1954, já haviam sido iniciados os planos para o Fortran. Em novembro de 1954, John Backus e seu grupo da IBM produziram um relatório intitulado "The IBM Mathematical FORmula TRANslating System: FORTRAN" (IBM, 1954). Esse documento descrevia a primeira versão do Fortran, a qual nos referimos como Fortran 0, antes de sua implementação. O documento também afirmava que forneceria a eficiência de programas codificados manualmente com a facilidade de programação dos sistemas de interpretação de pseudocódigo. Em outra rajada de otimismo, o documento afirmava que o Fortran eliminaria os erros de codificação e o processo de depuração. Baseado nessa premissa, o primeiro compilador Fortran incluía pouca verificação de erros de sintaxe.

O ambiente no qual o Fortran foi desenvolvido era o seguinte: (1) os computadores tinham memórias pequenas, eram lentos e relativamente não confiáveis; (2) o uso primário dos computadores era para computações científicas; (3) não existiam maneiras eficientes e efetivas de programar computadores; (4) devido ao alto custo dos computadores comparados com o dos programadores, a velocidade do código objeto gerado era o objetivo principal

dos primeiros compiladores Fortran. As características das primeiras versões do Fortran são diretamente oriundas desse ambiente.

2.3.3 Visão geral do Fortran I

O Fortran 0 foi modificado durante o período de implementação, que começou em janeiro de 1955 e continuou até o lançamento do compilador em abril de 1957. A linguagem implementada, que chamamos de Fortran I, é descrita no primeiro *Manual de Referência do Programador* Fortran, publicado em outubro de 1956 (IBM, 1956). O Fortran I incluía formatação de entrada e saída, nomes de variáveis até seis caracteres (eram apenas dois no Fortran 0), sub-rotinas definidas pelos usuários, apesar de elas não poderem ser compiladas separadamente, a sentença de seleção `If` e a sentença de repetição `Do`.

Todas as sentenças de controle do Fortran I eram baseadas em instruções do 704. Não fica claro se os projetistas do 704 ditaram o projeto das sentenças de controle do Fortran I ou se os do Fortran I sugeriram essas instruções para os do 704.

Não existiam sentenças para tipagem de dados na linguagem Fortran I. Variáveis cujos nomes começassem com `I`, `J`, `K`, `L`, `M` e `N` eram implicitamente do tipo inteiro e todas as outras eram de ponto flutuante. A escolha das letras para essa convenção foi baseada no fato de que, na época, os cientistas e engenheiros usavam inteiros como índices, normalmente *i*, *j* e *k*. Em um momento de generosidade, os projetistas do Fortran inseriram mais três letras.

A afirmação mais audaciosa feita pelo grupo de desenvolvimento do Fortran durante o projeto da linguagem era que o código de máquina produzido pelo compilador teria cerca da metade da eficiência que poderia ser produzida à mão[1]. Isso, mais do que qualquer coisa, fez os usuários potenciais ficarem céticos a seu respeito e frustrou muito do interesse no Fortran antes de seu lançamento. Para a surpresa de muitos, entretanto, o grupo de desenvolvimento do Fortran quase atingiu sua meta de eficiência. A grande parte do esforço de 18 trabalhadores-ano usada para construir o primeiro compilador foi gasta em otimização, e os resultados foram extraordinariamente efetivos.

O rápido sucesso do Fortran é mostrado pelos resultados de uma pesquisa feita em abril de 1958. Na época, aproximadamente metade do código que estava sendo escrito para o 704 era em Fortran, apesar do ceticismo do mundo da programação apenas um ano antes.

2.3.4 Fortran II

O compilador Fortran II foi distribuído na primavera de 1958. Ele corrigiu diversos problemas do sistema de compilação do Fortran I e adicionou recursos

[1] Na verdade, a equipe do Fortran acreditava que o código gerado pelo seu compilador não poderia ter menos do que a metade da rapidez com que o código de máquina escrito manualmente rodava, senão a linguagem não seria adotada pelos usuários.

significativos à linguagem, cujo mais importante foi a compilação independente de sub-rotinas. Sem a compilação independente, quaisquer mudanças em um programa requeriam que ele todo fosse recompilado. A falta de compilação independente no Fortran I, agregada à pobre confiabilidade do 704, colocou uma restrição prática no tamanho máximo dos programas de cerca de 300 a 400 linhas (Wexelblat, 1981, p. 68). Programas mais extensos têm uma pequena chance de serem compilados completamente antes da ocorrência de uma falha de máquina. A capacidade de incluir versões pré-compiladas em linguagem de máquina dos subprogramas diminuiu o processo de compilação consideravelmente.

2.3.5 Fortrans IV, 77, 90, 95 e 2003

Um Fortran III foi desenvolvido, mas nunca amplamente distribuído. O Fortran IV, entretanto, tornou-se uma das linguagens de programação mais utilizadas de seu tempo. Ele evoluiu no período de 1960 até 1962 e foi padronizado como Fortran 66 (ANSI, 1966), apesar de esse nome ser raramente usado. O Fortran IV era uma melhoria ao Fortran II em muitos pontos. Dentre suas adições mais importantes estavam as declarações de tipo explícitas para variáveis, uma construção `If` lógica e a capacidade de passar subprogramas como parâmetros para outros subprogramas.

O Fortran IV foi substituído pelo Fortran 77, que se tornou o novo padrão em 1978 (ANSI, 1987a). Ele manteve a maioria dos recursos do Fortran IV e adicionou manipulação de caracteres de cadeias, sentenças de controle de laços lógicos e um `If` com uma cláusula opcional `Else`.

O Fortran 90 (ANSI, 1992) era drasticamente diferente do Fortran 77. As adições mais significativas eram os vetores dinâmicos, os registros, os ponteiros, uma sentença de seleção múltipla e os módulos. Além disso, os subprogramas Fortran 90 poderiam ser chamados recursivamente.

Um novo conceito incluído na definição do Fortran 90 era o de remover recursos da linguagem de versões anteriores. Embora o Fortran 90 tivesse todos os recursos do Fortran 77, a definição da linguagem incluía uma lista de construções recomendadas para remoção na próxima versão da linguagem.

O Fortran 90 incluía duas mudanças sintáticas simples que alteravam a aparência tanto de programas quanto da literatura de descrição da linguagem. Primeiro, o formato fixo obrigatório do código, que requeria o uso de posições específicas dos caracteres para partes específicas das sentenças, foi abandonado. Por exemplo, rótulos poderiam aparecer apenas nas primeiras cinco posições e as sentenças não poderiam começar antes da sétima posição. Essa formatação rígida de código era projetada para o uso com cartões perfurados. A segunda mudança foi seu nome, de FORTRAN para Fortran, acompanhada pela mudança na convenção de usar letras maiúsculas para palavras-chave e para identificadores em programas Fortran. A nova convenção era que apenas a primeira letra das palavras-chave e dos identificadores deveria ter letra maiúscula.

O Fortran 95 (INCITS/ISO/IEC, 1997) continuou a evolução da linguagem, mas apenas algumas mudanças foram feitas. Dentre outras, uma nova construção de iteração, `Forall`, foi adicionada para facilitar a tarefa de paralelizar os programas Fortran.

A última versão do Fortran, Fortran 2003 (Metcalf et al., 2004), adiciona suporte à programação orientada a objetos, tipos derivados parametrizados, ponteiros para procedimentos e interoperabilidade com a linguagem de programação C.

2.3.6 Avaliação

A equipe de desenvolvimento original do Fortran pensou no projeto da linguagem apenas como um prelúdio necessário para a tarefa crítica de projetar o tradutor. Além disso, a equipe nunca havia pensado que o Fortran seria usado em computadores não fabricados pela IBM. Na verdade, eles foram forçados a considerar a construção de compiladores Fortran para outras máquinas IBM apenas porque o sucessor do 704, o 709, foi anunciado antes que o compilador Fortran para o 704 fosse lançado. O efeito do Fortran no uso de computadores, com o fato de que todas as linguagens de programação subsequentes devem algo a ele, é impressionante, considerando os modestos objetivos de seus projetistas.

Um dos recursos do Fortran I, e de todos os seus sucessores antes de 1990, que permite compiladores altamente otimizados é o fato de os tipos e o armazenamento para todas as variáveis serem fixados antes da execução. Nenhuma nova variável ou espaço pode ser alocado em tempo de execução. Esse é um sacrifício à flexibilidade em benefício da simplicidade e da eficiência, já que elimina a possibilidade de subprogramas recursivos e torna difícil a implementação de estruturas de dados que crescem ou mudam de forma dinamicamente. É claro, os tipos de programas construídos na época do desenvolvimento das primeiras versões do Fortran eram primariamente numéricos em sua natureza e simples comparados com os projetos de software recentes. Assim, o sacrifício não era muito grande.

O sucesso geral do Fortran é difícil de ser exagerado: ele mudou drasticamente a maneira como os computadores são usados. Isso, é claro, em grande parte por ter sido a primeira linguagem de alto nível muito usada. Comparativamente com conceitos e linguagens desenvolvidas depois, as primeiras versões do Fortran sofrem de uma variedade de maneiras, como o esperado, até porque não seria justo comparar o desempenho e o conforto de um Ford Modelo T 1910 com o desempenho e o conforto de um Ford Mustang 2009. Independentemente disso, apesar das inadequações do Fortran, o momento de alto investimento em software escrito na linguagem, dentre outros fatores, a mantiveram em uso por mais de meio século.

Alan Perlis, um dos projetistas do ALGOL 60, disse sobre o Fortran em 1978: "O Fortran é a língua franca do mundo da computação. É a língua das ruas no melhor sentido da palavra. E ele tem sobrevivido e sobreviverá porque se tornou uma parte extraordinariamente útil de um comércio vital (Wexelblat, 1981, p. 161)."

A seguir, temos um exemplo de um programa em Fortran 95:

```
! Programa de exemplo do Fortran 95
!  Entrada:   Um inteiro, List_Len, onde List_Len é menor do
!             que 100, seguido por valores inteiros List_Len
!  Saída:     O número de valores de entrada que são maiores
!             do que a média de todos os valores de entrada
Implicit none
Integer Dimension(99) :: dInt_List
Integer :: List_Len, Counter, Sum, Average, Result
Result= 0
Sum = 0
Read *, List_Len
If ((List_Len > 0) .AND. (List_Len < 100)) Then
! Lê os dados de entrada em um vetor e calcula sua soma
   Do Counter = 1, List_Len
      Read *, Int_List(Counter)
      Sum = Sum + Int_List(Counter)
   End Do
! Calcula a média
   Average = Sum / List_Len
! Conta os valores que são maiores do que a média
   Do Counter = 1, List_Len
      If (Int_List(Counter) > Average) Then
         Result = Result + 1
      End If
  End Do
! Imprimir o resultado
   Print *, 'Number of values > Average is:', Result
Else
   Print *, 'Error - list length value is not legal'
End If
End Program Example
```

2.4 PROGRAMAÇÃO FUNCIONAL: LISP

A primeira linguagem de programação funcional foi inventada para fornecer recursos para o processamento de listas, uma necessidade que cresceu a partir das primeiras aplicações na área de Inteligência Artificial (IA).

2.4.1 O início da inteligência artificial e do processamento de listas

O interesse em IA apareceu em meados dos anos 1950 em alguns lugares. Parte cresceu a partir da linguística, parte da psicologia e parte da matemática. Os linguistas estavam interessados no processamento de linguagem natural. Os psicólogos, em modelar o armazenamento e a recuperação de informações humanas. Os matemáticos, em mecanizar certos processos inteligentes, como a prova de teoremas. Todas essas pesquisas chegaram à mesma

conclusão. Algum método deve ser desenvolvido para permitir aos computadores processar dados simbólicos em listas encadeadas. Na época, a maioria das computações era feita em dados numéricos armazenados em vetores.

O conceito de processamento de listas foi desenvolvido por Allen Newell, J. C. Shaw e Herbert Simon. Ele foi publicado pela primeira vez em um artigo clássico que descreve um dos primeiros programas de IA, o Logical Theorist[2] (Teórico Lógico), e uma linguagem na qual ele poderia ser implementado (Newell e Simon, 1956), IPL-I (Information Processing Language I), que nunca foi posta em prática. A próxima versão, IPL-II, foi implementada em um computador Johnniac da Rand Corporation. O desenvolvimento da IPL continuou até 1960, quando a descrição da IPL-V foi publicada (Newell e Tonge, 1960). O baixo nível da linguagem evitou sua popularização. Elas eram linguagens de montagem para um computador hipotético, implementadas com um interpretador, no qual instruções de processamento de listas foram incluídas. Outro fator que impediu as linguagens IPL de se tornarem populares foi sua implementação na obscura máquina Johnniac.

As contribuições das linguagens IPL foram o seu projeto baseado em listas e a sua demonstração de que o processamento de listas era factível e útil.

A IBM começou a se interessar por IA no meio dos anos 1950 e escolheu a prova de teoremas como uma área de demonstração. Na época, o projeto do Fortran ainda estava no meio do caminho. O alto custo do compilador para Fortran I convenceu a IBM de que seu processamento de listas deveria ser anexado ao Fortran, em vez de na forma de uma nova linguagem. Então, a Fortran List Processing Language (FLPL) foi projetada e implementada como uma extensão do Fortran. FLPL foi usada para construir um provador de teoremas para geometria plana, que era considerada a área mais fácil para a prova mecânica de teoremas.

2.4.2 O processo do projeto de LISP

John McCarthy, do MIT, passou uma temporada de verão no Departamento de Pesquisa em Informação da IBM em 1958. Seu objetivo era investigar computações simbólicas e desenvolver um conjunto de requisitos para fazê-las. Como uma área de problema de exemplo piloto, ele escolheu a diferenciação de expressões algébricas. A partir desse estudo, foi criada uma lista dos requisitos da linguagem. Dentre eles, estavam os métodos de controle de fluxo de funções matemáticas: recursão e expressões condicionais. A única linguagem de alto nível disponível na época, o Fortran I, não tinha nenhuma das duas.

Outro requisito que surgiu a partir da pesquisa sobre diferenciação simbólica foi a necessidade de alocar listas encadeadas dinamicamente e algum tipo de liberação implícita de listas abandonadas. McCarthy não permitiria que seu elegante algoritmo para diferenciação fosse inchado com sentenças explícitas de liberação.

---

[2] `O Logical Theorist` descobriu provas para teoremas em cálculo proposicional.

Como a FLPL não suportava recursão, expressões condicionais, alocação dinâmica de armazenamento, e alocação implícita, estava claro para McCarthy que era necessária uma nova linguagem.

Quando McCarthy retornou ao MIT, no final de 1958, ele e Marvin Minsky formaram o Projeto IA do MIT, com financiamento do Laboratório de Pesquisa para Eletrônica. O primeiro esforço importante era produzir um sistema para o processamento de listas. Ele seria usado inicialmente para implementar um programa proposto por McCarthy chamado de Advice Taker[3]. Essa aplicação se tornou o ímpeto para o desenvolvimento da linguagem de processamento de listas chamada LISP. A primeira versão de LISP é algumas vezes chamada de "LISP puro", porque é uma linguagem puramente funcional. Na seção seguinte, descreveremos o desenvolvimento do LISP puro.

2.4.3 Visão geral da linguagem

2.4.3.1 Estruturas de dados

O LISP puro tem apenas dois tipos de estruturas de dados: átomos e listas. Átomos são símbolos, que têm a forma de identificadores ou literais numéricos. O conceito de armazenar informações simbólicas em listas encadeadas é natural e era usado em IPL-II. Tais estruturas permitem a inserção e a exclusão em qualquer ponto – operações pensadas, na época, como parte necessária do processamento de listas. Foi determinado, entretanto, que os programas LISP raramente precisavam dessas operações.

As listas são especificadas com a delimitação de seus elementos com parênteses. Listas simples, nas quais os elementos são restritos a átomos, têm o formato

```
(A B C D)
```

Estruturas de listas aninhadas também são especificadas com parênteses. Por exemplo, a lista

```
(A (B C) D (E (F G)))
```

é composta de quatro elementos. O primeiro é o átomo `A`; o segundo é a sublista `(B C)`; o terceiro é o átomo `D`; o quarto é a sublista `(E (F G))`, que tem como seu segundo elemento a sublista `(F G)`.

Internamente, as listas são armazenadas como estruturas de listas simplesmente encadeadas, nas quais cada nó tem dois ponteiros que apontam para alguma representação do átomo, como seu valor simbólico ou numérico, ou o ponteiro para uma sublista. Um nó para um elemento que é uma sublista tem seu primeiro ponteiro apontando para o primeiro nó da sublista. Em ambos os casos, o segundo ponteiro de um nó aponta para o próximo

[3] O *Advice Taker* representava a informação com sentenças escritas em uma linguagem fomal e usava um processo de inferência lógica para decidir o que fazer.

elemento da lista. Uma lista é referenciada por um ponteiro para seu primeiro elemento.

As representações internas das duas listas mostradas anteriormente são ilustradas na Figura 2.2. Note que os elementos de uma lista são mostrados horizontalmente. O último elemento de uma lista não tem sucesso, então sua ligação é `NIL`, representado na Figura 2.2 como uma linha diagonal no elemento. As sublistas são mostradas com a mesma estrutura.

2.4.3.2 Processos em programação funcional

LISP foi projetada como uma linguagem de programação funcional. Todas as computações em um programa puramente funcional são realizadas por meio da aplicação de funções a argumentos. Nem as sentenças de atribuição nem as variáveis abundantes em programas escritos em linguagens imperativas são necessárias em programas escritos em linguagens funcionais. Além disso, processos repetitivos podem ser especificados com chamadas a funções recursivas, tornando as iterações (laços) desnecessárias. Esses conceitos básicos de programação funcional a tornam significativamente diferente de programar em uma linguagem imperativa.

2.4.3.3 A sintaxe de LISP

LISP é muito diferente das linguagens imperativas, tanto porque é funcional quanto porque a aparência dos programas LISP é muito diferente daquela de linguagens como Java ou C++. Por exemplo, a sintaxe de Java é uma mistura complicada de inglês e álgebra, enquanto a sintaxe de LISP é um modelo de

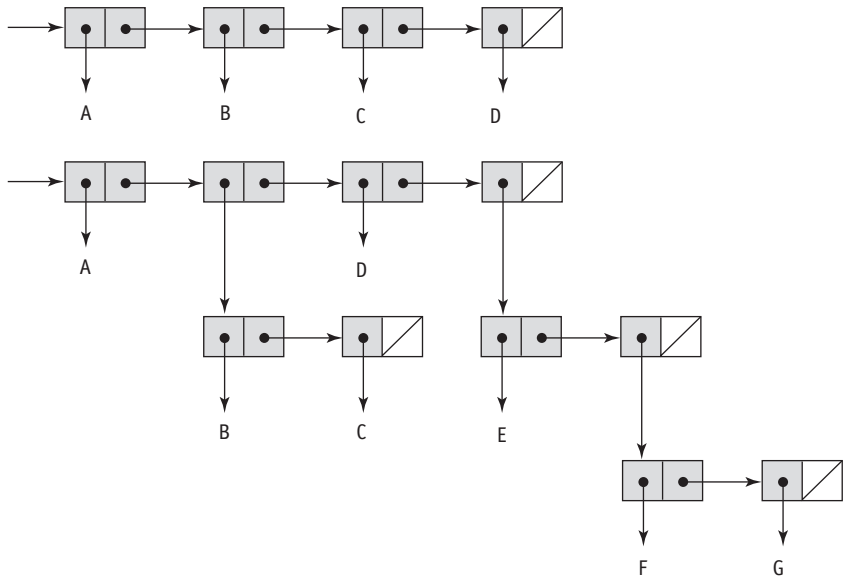

Figura 2.2 Representação interna de duas listas em LISP.

simplicidade. O código e os dados dos programas têm exatamente a mesma forma: listas dentro de parênteses. Considere mais uma vez a lista

```
(A B C D)
```

Quando interpretada como dados, ela é uma lista de quatro elementos. Se vista como código, é a aplicação da função chamada `A` para os três parâmetros `B`, `C` e `D`.

2.4.4 Avaliação

LISP dominou completamente as aplicações de IA por um quarto de século. Muitas das causas da reputação de LISP de ser altamente ineficiente foram eliminadas. Muitas das implementações contemporâneas são compiladas, e o código resultante é muito mais rápido do que rodar o código fonte em um interpretador. Além do seu sucesso em IA, LISP foi pioneira na programação funcional, que se provou uma área extremamente ativa na pesquisa em linguagens de programação. Conforme mencionado no Capítulo 1, muitos pesquisadores de linguagens de acreditam que a programação funcional é uma abordagem muito melhor para o desenvolvimento de software do que a programação procedural usando linguagens imperativas.

A seguir, temos um exemplo de um programa em LISP:

```
; Função de exemplo em LISP
; O código a seguir define uma função de predicado em LISP
; que recebe duas listas como argumentos e retorna True
; se as duas listas forem iguais, e NIL (false) caso contrá-
  rio
    (DEFUN equal_lists (lis1 lis2)
    (COND
      ((ATOM lis1) (EQ lis1 lis2))
      ((ATOM lis2) NIL)
      ((equal_lists (CAR lis1) (CAR lis2))
                (equal_lists (CDR lis1) (CDR lis2)))
      (T NIL)
   )
)
```

2.4.5 Dois descendentes de LISP

Dois dialetos de LISP são muito usados agora, Scheme e COMMON LISP. Eles são discutidos brevemente nas subseções a seguir.

2.4.5.1 Scheme

A linguagem Scheme emergiu do MIT em meados dos anos 1970 (Dybvig, 2003). Ela é caracterizada por seu tamanho diminuto, seu uso exclusivo de escopo estático (discutido no Capítulo 5) e seu tratamento de funções como entidades de primeira classe. As funções Scheme podem ser atribuídas a variáveis, passadas como parâmetros, e retornadas como valores da aplicação

de funções. Elas também podem ser elementos de listas. As primeiras versões de LISP não forneciam todas essas capacidades, nem usavam escopo estático.

Como uma pequena linguagem com uma sintaxe e semântica simples, Scheme é bastante adequada para aplicações educacionais, como cursos sobre programação funcional e introduções gerais à programação. Scheme é descrita em alguns detalhes no Capítulo 15.

2.4.5.2 COMMON LISP

Durante os anos 1970 e o início dos anos 1980, diversos dialetos diferentes de LISP foram desenvolvidos e usados. Isso levou ao familiar problema da portabilidade. COMMON LISP (Graham, 1996) foi criada em um esforço para corrigir essa situação, projetada para combinar os recursos de diferentes dialetos de LISP desenvolvidos nos anos 1980, incluindo Scheme, em uma única linguagem. Sendo um amálgama, COMMON LISP é uma linguagem grande e complexa. Sua base, entretanto, é o LISP puro, então sua sintaxe, funções primitivas e a natureza fundamental vêm dessa linguagem.

Reconhecendo a flexibilidade fornecida pelo escopo dinâmico e a simplicidade do escopo estático, COMMON LISP permite ambos. O escopo padrão para variáveis é estático, mas ao declarar uma como `special`, ela se torna uma variável com escopo dinâmico.

COMMON LISP tem um grande número de tipos e estruturas de dados, incluindo registros, vetores, números complexos e cadeias de caracteres. Ela também tem uma forma de pacotes para modular coleções de funções e dados que fornece controle de acesso.

COMMON LISP é descrita em mais detalhes no Capítulo 15.

2.4.6 Linguagens relacionadas

ML (MetaLanguage; Ullman, 1998) foi originalmente projetada nos anos 1980 por Robin Milner, na Universidade de Edimburgo, como uma metalinguagem para um sistema de verificação de programas chamado Logic for Computable Functions (LCF; Milner et al., 1990). ML é primariamente uma linguagem funcional, mas também oferece suporte para programação imperativa.

Diferentemente de LISP e de Scheme, o tipo de cada variável e de cada expressão em ML pode ser determinado em tempo de compilação[4]. Os tipos são associados com objetos em vez de nomes. Os tipos das expressões são inferidos do contexto da expressão, conforme discutido no Capítulo 5. Diferentemente de LISP e de Scheme, ML não usa a sintaxe funcional baseada em parênteses que se originou com expressões lambda. Em vez disso, a sintaxe de ML lembra aquela das linguagens imperativas, como Java e C++.

[4] Embora isso seja verdade, as funções ML podem ser genéricas, e isso significa que os tipos dos parâmetros e tipos de retorno podem ser diferentes para diferentes chamadas.

Miranda foi desenvolvida por David Turner (1986) na Universidade de Kent em Canterbury, na Inglaterra, no início dos anos 1980. Ela é parcialmente baseada nas linguagens ML, SASL e KRC. Haskell (Hudak e Fasel, 1992), por sua vez, é baseada em Miranda e, assim como ela, é uma linguagem puramente funcional, não tendo variáveis nem sentenças de atribuição. Outra característica que distingue Haskell é seu uso de avaliação tardia (*lazy evaluation*), ou seja, nenhuma expressão é avaliada até seu valor ser necessário, levando a algumas capacidades surpreendentes na linguagem.

Tanto ML quanto Haskell são brevemente discutidas no Capítulo 15.

2.5 O PRIMEIRO PASSO EM DIREÇÃO À SOFISTICAÇÃO: ALGOL 60

O ALGOL 60 tem tido uma grande influência nas linguagens de programação subsequentes e é de uma importância central em qualquer estudo histórico de linguagens.

2.5.1 Perspectiva histórica

O ALGOL 60 foi o resultado de esforços para projetar uma linguagem universal para aplicações científicas. No final de 1954, o sistema algébrico de Laning e Zierler estava em operação por mais de um ano, e o primeiro relatório sobre o Fortran havia sido publicado. O Fortran se tornou uma realidade em 1957, e diversas outras linguagens de alto nível haviam sido desenvolvidas. Dentre essas, a mais notável era a IT, projetada por Alan Perlis no Carnegie Tech, e duas linguagens para os computadores UNIVAC, MATH-MATIC e UNICODE. A proliferação de linguagens fez com que a comunicação entre usuários se tornasse difícil. Além disso, as novas linguagens estavam crescendo em torno de arquiteturas únicas, algumas para os computadores UNIVAC, outras para as máquinas da série IBM 700. Em resposta a essa proliferação de linguagens, diversos dos principais usuários de computadores nos Estados Unidos, incluindo o SHARE (o grupo de usuários científicos da IBM) e o USE (UNIVAC Scientific Exchange, o grupo de grande escala de usuários científicos da UNIVAC), submeteu uma petição à ACM (Association for Computing Machinery) em 10 de maio de 1957, para formar um comitê para estudar e recomendar ações para uma linguagem de programação científica independente de linguagem de programação. Apesar de o Fortran poder ser tal candidato, ele não se tornaria uma linguagem universal, já que era de propriedade única da IBM.

Previamente, em 1955, a GAMM (Sociedade de Matemática e Mecânica Aplicada, na sigla em alemão) formou um comitê para projetar uma linguagem algorítmica universal, independente de máquina. O desejo por essa nova linguagem era em parte devido ao medo dos europeus de serem dominados pela IBM. No final de 1957, entretanto, a aparição de diversas linguagens de alto nível nos Estados Unidos convenceu o subcomitê da GAMM de que seus

esforços precisavam ser ampliados para incluir os americanos, e uma carta convite foi enviada à ACM. Em abril de 1958, após Fritz Bauer, da GAMM, apresentar uma proposta formal à ACM, os dois grupos concordaram oficialmente em desenvolver uma linguagem conjunta.

2.5.2 Processo do projeto inicial

A GAMM e a ACM enviaram cada uma quatro membros para a primeira reunião de projeto. A reunião, realizada em Zurique de 27 de maio a 1º de junho de 1958, começou com os seguintes objetivos para a nova linguagem:

- A sintaxe da linguagem deve ser o mais próxima possível da notação padrão matemática e os programas devem ser legíveis, com poucas explicações adicionais.
- Deve ser possível usar a linguagem para a descrição de algoritmos em publicações.
- Programas na nova linguagem devem ser mecanicamente traduzíveis em código de máquina.

O primeiro objetivo indica que a nova linguagem seria usada para programação científica, a principal área da aplicação de computadores na época. O segundo era algo inteiramente novo nos negócios em computação. O último é uma necessidade óbvia para qualquer linguagem de programação.

A reunião de Zurique foi bem-sucedida em produzir uma linguagem que atendesse os objetivos levantados, mas o processo do projeto necessitava de inúmeros comprometimentos, tanto entre indivíduos quanto entre os dois lados do Atlântico. Em alguns casos, não era tanto em torno de grandes questões, mas em termos de esferas de influência. A questão de usar uma vírgula (o método europeu) ou um ponto (o método americano) para um ponto decimal é um exemplo.

2.5.3 Visão geral do ALGOL 58

A linguagem projetada na reunião em Zurique foi nomeada de Linguagem Algorítmica Internacional (IAL – International Algorithmic Language). Foi sugerido durante o projeto que ela se chamasse ALGOL, do inglês ALGOrithmic Language, mas o nome foi rejeitado porque ele não refletia o escopo internacional do comitê. Durante o ano seguinte, entretanto, o nome foi mudado para ALGOL, e a linguagem ficou conhecida como ALGOL 58.

De muitas formas, o ALGOL 58 era um descendente do Fortran, o que é bastante natural. Ele generalizou muitos dos recursos do Fortran e adicionou novas construções e conceitos. Algumas das generalizações eram relativas ao objetivo de não amarrar a linguagem a nenhuma máquina em particular e outras eram tentativas de tornar a linguagem mais flexível e poderosa. Uma rara combinação de simplicidade e elegância emergiu desse esforço.

O ALGOL 58 formalizou o conceito de tipo de dados, apesar de apenas variáveis que não fossem de ponto flutuante precisarem ser explicitamente declaradas. Ele adicionou a ideia de sentenças compostas, que a maioria das linguagens subsequente incorporou. Alguns dos recursos do Fortran que foram generalizados são: os identificadores podem ter qualquer tamanho, em oposição à restrição do Fortran I de nomes de identificadores com até seis caracteres; qualquer número de dimensões de um vetor era permitido, diferentemente da limitação do Fortran I de até três dimensões; o limite inferior dos vetores podia ser especificado pelo programador, enquanto no Fortran ele era implicitamente igual a 1; sentenças de seleção aninhadas eram permitidas, o que não era o caso no Fortran I.

O ALGOL 58 adquiriu o operador de atribuição de uma maneira um tanto usual. Zuse usava o formato

expressão => variável

para a sentença de atribuição em Plankalkül. Apesar de Plankalkül não ter sido ainda publicada, alguns dos membros europeus do comitê do ALGOL 58 estavam familiarizados com a linguagem. O comitê se interessou com a forma da atribuição em Plankalkül, mas devido aos argumentos sobre limitações do conjunto de caracteres, o símbolo maior foi trocado pelo sinal de dois pontos. Então, em grande parte por causa da insistência dos americanos, toda a sentença foi modificada para ser equivalente ao formato do Fortran

variável := expressão

Os europeus preferiam a forma oposta, mas isso seria o inverso do Fortran.

2.5.4 Recepção do relatório do ALGOL 58

A publicação do relatório do ALGOL 58 (Perlis e Samelson, 1958), em dezembro de 1958, foi recebida com uma boa dose de entusiasmo. Nos Estados Unidos, a nova linguagem foi vista mais como uma coleção de ideias para o projeto de linguagens de programação do que uma linguagem padrão universal. Na verdade, o relatório do ALGOL 58 não pretendia ser um produto finalizado, mas um documento preliminar para discussão internacional. Independentemente disso, três grandes esforços de projeto e implementação foram feitos usando o relatório como base. Na Universidade de Michigan, a linguagem MAD nasceu (Arden et al., 1961). O Grupo de Eletrônica Naval dos Estados Unidos produziu a linguagem NELIAC (Huskey et al., 1963). Na System Development Corporation, a linguagem JOVIAL foi projetada e implementada (Shaw, 1963). JOVIAL, um acrônimo para Jules' Own Version of the International Algebraic Language (Versão de Jules para a Linguagem Internacional Algébrica), representa a única linguagem baseada no ALGOL 58 a atingir um amplo uso. (Jules era Jules I. Schwartz, um dos projetistas de JOVIAL). JOVIAL se

tornou bastante usada porque foi a linguagem científica oficial para a Força Aérea Americana por um quarto de século.

O resto da comunidade da computação nos Estados Unidos não estava tão gentil com a nova linguagem. Em um primeiro momento, tanto a IBM quanto seu maior grupo de usuários científicos, o SHARE, pareciam ter abraçado o ALGOL 58. A IBM começou uma implementação logo após o relatório ter sido publicado, e o SHARE formou um subcomitê, SHAREL IAL, para estudar a linguagem. Logo a seguir, o subcomitê recomendou que a ACM padronizasse o ALGOL 58 e que a IBM o implementasse para toda a série de computadores IBM 700. O entusiasmo durou pouco. Na primavera de 1959, tanto a IBM quanto o SHARE, com sua experiência com o Fortran, já haviam tido sofrimento e despesas suficientes para iniciar uma nova linguagem, tanto em termos de desenvolver e usar os compiladores de primeira geração quanto de treinar os usuários na nova linguagem e persuadi-los a usá-la. Na metade de 1959, tanto a IBM quanto o SHARE haviam desenvolvido um interesse próprio pelo Fortran, levando a decisão de mantê-lo como *a* linguagem científica para as máquinas IBM da série 700, abandonando o ALGOL 58.

2.5.5 O processo do projeto do ALGOL 60

Durante 1959, o ALGOL 58 foi debatido intensamente tanto na Europa quanto nos Estados Unidos. Um grande número de modificações e adições foi publicado no ALGOL Bulletin europeu e na Communications of the ACM. Um dos eventos mais importantes de 1959 foi a apresentação do trabalho do comitê de Zurique na Conferência Internacional de Processamento de Informação, na qual Backus introduziu sua notação para descrever a sintaxe de linguagens de programação, posteriormente conhecida como BNF (do inglês – *Backus-Naur Form*). BNF é descrita em detalhes no Capítulo 3.

Em janeiro de 1960, a segunda reunião do ALGOL foi realizada, dessa vez em Paris, com o objetivo de debater as 80 sugestões submetidas para consideração. Peter Naur da Dinamarca havia se envolvido enormemente no desenvolvimento do ALGOL, apesar de não ser um membro do grupo de Zurique. Foi Naur que criou e publicou o ALGOL Bulletin. Ele gastou bastante de tempo estudando o artigo de Backus que introduzia a BNF e decidiu que ela deveria ser usada para descrever formalmente os resultados da reunião de 1960. Após ter feito algumas mudanças relativamente pequenas à BNF, ele escreveu uma descrição da nova linguagem proposta em BNF e entregou para os membros do grupo de 1960 no início da reunião.

2.5.6 Visão geral do ALGOL 60

Apesar de a reunião de 1960 ter durado apenas seis dias, as modificações feitas no ALGOL 58 foram drásticas. Dentre os mais importantes avanços, estavam:

- O conceito de estrutura de bloco foi introduzido, o que permitia ao programador localizar partes do programa introduzindo novos ambientes ou escopos de dados.
- Duas formas diferentes de passagem de parâmetros a subprogramas foram permitidas: por valor e por nome.
- Foi permitido aos procedimentos serem recursivos. A descrição do ALGOL 58 não era clara em relação a essa questão. Note que, apesar de essa recursão ser nova para as linguagens imperativas, LISP já fornecia funções recursivas em 1959.
- Vetores dinâmicos na pilha eram permitidos. Um vetor dinâmico na pilha é um no qual a faixa ou faixas de índices são especificados por variáveis, de forma que seu tamanho é determinado no momento em que o armazenamento é alocado, o que acontece quando a declaração é alcançada durante a execução. Vetores dinâmicos na pilha são discutidos em detalhe no Capítulo 6.

Diversos recursos que poderiam ter gerado um grande impacto no sucesso ou na falha da linguagem foram propostos, mas rejeitados. O mais importante deles eram sentenças de entrada e saída com formatação, omitidas porque se pensava que seriam muito dependentes de máquina.

O relatório do ALGOL 60 foi publicado em maio de 1960 (Naur, 1960). Algumas ambiguidades ainda permaneceram na descrição da linguagem, e uma terceira reunião foi marcada para abril de 1962, em Roma, para resolver esses problemas. Nessa reunião, o grupo tratou apenas dos problemas; nenhuma adição à linguagem foi permitida. Os resultados foram publicados sob o título *Revised Report on the Algorithmic Language ALGOL 60* (Backus et al., 1963).

2.5.7 Avaliação

De algumas formas, o ALGOL 60 foi um grande sucesso; de outras, um imenso fracasso. Foi bem-sucedido em se tornar, quase imediatamente, a única maneira formal aceitável de comunicar algoritmos na literatura em computação – e permaneceu assim por mais de 20 anos. Todas as linguagens de programação imperativas desde 1960 devem algo ao ALGOL 60. De fato, muitas são descendentes diretos ou indiretos, como PL/I, SIMULA 67, ALGOL 68, C, Pascal, Ada, C++ e Java.

O esforço de projeto do ALGOL 58/ALGOL 60 incluiu uma longa lista de primeiras vezes. Foi a primeira vez que um grupo internacional tentou projetar uma linguagem de programação, a primeira linguagem projetada para ser independente de máquina e também a primeira cuja sintaxe foi formalmente descrita. O sucesso do uso do formalismo BNF iniciou diversos campos importantes na ciência da computação: linguagens formais, teoria de análise sintática e projeto de compilador baseado em BNF. Finalmente, a estrutura do ALGOL 60 afetou as arquiteturas de máquina. No efeito mais contundente disso, uma extensão da linguagem foi usada como a lingua-

gem de sistema de uma série de computadores de grande escala, as máquinas Borroughs B5000, B6000 e B7000, projetadas com uma pilha de hardware para implementar eficientemente a estrutura de bloco e os subprogramas recursivos da linguagem.

Do outro lado da moeda, o ALGOL 60 nunca atingiu um uso disseminado nos Estados Unidos. Mesmo na Europa, onde era mais popular, nunca se tornou a linguagem dominante. Existem diversas razões para sua falta de aceitação. Por um lado, alguns dos recursos se mostraram muito flexíveis – fizeram com que o entendimento da linguagem fosse mais difícil e a implementação ineficiente. O melhor exemplo disso é o método de passagem de parâmetros por nome para os subprogramas, explicado no Capítulo 9. As dificuldades para implementar o ALGOL 60 foram evidenciadas por Rutishauser em 1967, que disse que poucas implementações (se é que alguma) incluíam a linguagem ALGOL 60 completa (Rutishauser, 1967, p. 8).

A falta de sentenças de entrada e saída na linguagem era outra razão para sua falta de aceitação. A entrada e saída dependente de implementação fez os programas terem uma portabilidade ruim para outros computadores.

Uma das mais importantes contribuições à ciência da computação associada ao ALGOL, a BNF, também foi um fator. Apesar de a BNF ser agora considerada uma maneira simples e elegante de descrição de sintaxe, para o mundo de 1960 ela parecia estranha e complicada.

Finalmente, apesar de haver muitos outros problemas, o forte estabelecimento do Fortran entre os usuários e a falta de suporte da IBM foram provavelmente os fatores mais importantes na falha do ALGOL 60 em ter seu uso disseminado.

O esforço do ALGOL 60 nunca foi realmente completo, no sentido de que ambiguidades e obscuridades sempre fizeram parte da descrição da linguagem (Knuth, 1967).

A seguir, é mostrado um exemplo de um programa em ALGOL:

```
comment Programa de exemplo do ALGOL 60
 Entrada:  Um inteiro, listlen, onde listlen é menor do que
           100, seguido por valores inteiros listlen
 Saída:    O número de valores de entrada que são maiores do
           que a média de todos os valores de entrada ;
begin
  integer array intlist [1:99];
  integer listlen, counter, sum, average, result;
  sum := 0;
  result := 0;
  readint (listlen);
  if (listlen > 0) ∧ (listlen < 100) then
    begin
comment Lê os dados de entrada em um vetor e calcula sua média;
    for counter := 1 step 1 until listlen do
      begin
      readint (intlist[counter]);
```

```
      sum := sum + intlist[counter]
      end;
comment Calcula a média;
    average := sum / listlen;
comment Conta os valores que são maiores que a média;
    for counter := 1 step 1 until listlen do
      if intlist[counter] > average
        then result := result + 1;
comment Imprimir o resultado;
    printstring("The number of values > average is:");
    printint (result)
    end
  else
    printstring ("Error—input list length is not legal");
end
```

2.6 INFORMATIZANDO OS REGISTROS COMERCIAIS: COBOL

A história do COBOL é, de certa forma, o oposto da do ALGOL 60. Apesar de ter sido mais usado do que qualquer outra linguagem de programação, ele teve pouco efeito no projeto de linguagens subsequentes, exceto por PL/I. O COBOL pode ainda ser a linguagem mais usada[5], apesar de ser difícil ter certeza disso. Talvez a razão mais importante pela qual o COBOL tem uma influência pequena é que poucos tentaram projetar uma nova linguagem de negócios desde sua aparição. Isso ocorreu em parte por causa do quão bem as capacidades do COBOL satisfazem as necessidades de sua área de aplicação. Outra razão é que uma grande parcela da computação em negócios nos últimos 30 anos ocorreu em pequenas empresas. E nelas, pouco desenvolvimento de software ocorre. Em vez disso, muito do software usado é comprado como pacotes de prateleira para várias aplicações gerais.

2.6.1 Perspectiva histórica

O início do COBOL é, de certa forma, similar ao do ALGOL 60. A linguagem também foi projetada por um comitê de pessoas que se reuniam em períodos relativamente curtos. O estado da computação de negócios na época, no caso 1959, era similar ao estado da computação científica quando o Fortran estava sendo projetado. Uma linguagem compilada para aplicações de negócios, chamada FLOW-MATIC, havia sido implementada em 1957, mas pertencia à UNIVAC e foi projetada para os computadores dessa empresa. Outra linguagem, a AIMACO, estava sendo usada pela Força Aérea americana, mas era apenas uma pequena variação do FLOW-MATIC.

[5] No final dos anos 1990, em um estudo associado ao problema do ano 2000, foi estimado que existiam aproximadamente 800 milhões de linhas de código COBOL em uso em uma área de 35 quilômetros quadrados de Manhattan.

A IBM havia projetado uma linguagem de programação para aplicações comerciais, chamada COMTRAN (COMmercial TRANslator), mas que nunca foi implementada. Diversos outros projetos de linguagens haviam sido planejados.

2.6.2 FLOW-MATIC

Vale a pena discutir brevemente as origens do FLOW-MATIC, porque ele foi o principal progenitor do COBOL. Em dezembro de 1953, Grace Hopper na Remington-Rand UNIVAC escreveu uma proposta profética. Ela sugeria que "programas matemáticos devem ser escritos em notação matemática, programas de processamento de dados devem ser escritos em sentenças em inglês" (Wexelbal, 1981, p. 16). Infelizmente, era impossível em 1953 convencer não programadores de que um computador poderia ser feito para entender palavras em inglês. Somente em 1955 uma proposta similar teve alguma esperança de ser patrocinada pela UNIVAC, e mesmo assim precisou de um protótipo para convencer a gerência. Parte do processo de venda envolvia compilar e rodar um pequeno programa, primeiro usando palavras-chave em inglês, depois palavras-chave em francês, e então palavras-chave em alemão. Essa demonstração foi considerada notável pela gerência da UNIVAC e foi um dos fatores principais para a aceitação da proposta de Hopper.

2.6.3 O processo do projeto do COBOL

A primeira reunião formal sobre o assunto de uma linguagem comum para aplicações de negócios, patrocinada pelo Departamento de Defesa, ocorreu no Pentágono em 28 e 29 de maio de 1959 (exatamente um ano após a reunião do ALGOL em Zurique). O consenso do grupo era que a linguagem, na época chamada CBL (de Common Business Language), deveria ter certas características gerais. A maioria concordou que ela deveria usar inglês o máximo possível, apesar de alguns terem argumentado a favor de uma notação mais matemática. A linguagem deveria ser fácil de utilizar, mesmo ao custo de ser menos poderosa, de forma a aumentar a base daqueles que poderiam programar computadores. Somando-se ao fato de tornar a linguagem mais fácil, acreditava-se que o uso de inglês poderia permitir aos gerentes lerem os programas. Finalmente, o projeto não deveria ser restringido pelos problemas de sua implementação.

Uma das preocupações recorrentes na reunião era que os passos para criar essa linguagem universal deveriam ser dados rapidamente, já que um monte de trabalho já estava sendo feito para criar novas linguagens de negócios. Além das existentes, a RCA e a Sylvania estavam trabalhando em suas próprias linguagens para aplicações de negócios. Estava claro que, quanto mais tempo levasse para produzir uma universal, mais difícil seria para que ela se tornasse amplamente usada. Dessa forma, foi decidido que deveria existir um rápido estudo sobre as linguagens existentes. Para essa tarefa, o Short Range Committee foi formado.

Existiram decisões iniciais para separar as sentenças da linguagem em duas categorias – descrições de dados e operações executáveis – e que as sentenças dessas duas categorias residiriam em partes diferentes dos programas. Um dos debates do Short Range Committee foi sobre a inclusão de índices. Muitos membros do comitê argumentaram que índices seriam muito complexos para as pessoas trabalharem em processamento de dados, as quais se pensava que não ficariam confortáveis com uma notação matemática. Argumentos similares eram feitos sobre incluir ou não expressões aritméticas. O relatório final do Short Range Committee, completado em dezembro de 1959, descrevia a linguagem que mais tarde foi chamada de COBOL 60.

As especificações de linguagem do COBOL 60, publicadas pela Agência de Impressão do Governo americano (Government Printing Office) em abril de 1960 (Department of Defense, 1960), foram descritas como "iniciais". Versões revisadas foram publicadas em 1961 e 1962 (Department of Defense, 1961, 1962). A linguagem foi padronizada pelo Instituto Nacional de Padrões dos Estados Unidos (ANSI – American National Standards Institute) em 1968. As três revisões seguintes foram padronizadas pelo ANSI em 1974, 1985 e 2002. A linguagem continua a evoluir até hoje.

2.6.4 Avaliação

A linguagem COBOL originou diversos conceitos inovadores, alguns dos quais apareceram em outras linguagens. Por exemplo, o verbo `DEFINE` do COBOL 60 foi a primeira construção para macros de uma linguagem de alto nível. E o mais importante: estruturas de dados hierárquicas (registros), que apareceram em Plankalkül, foram primeiro implementadas em COBOL. Os registros têm sido incluídos na maioria das linguagens imperativas projetadas desde então. COBOL também foi a primeira linguagem a permitir nomes realmente conotativos, pois permitia nomes longos (até 30 caracteres) e caracteres conectores de palavras (hifens).

De um modo geral, a divisão de dados (*data division*) é a parte forte do projeto do COBOL, enquanto a divisão de procedimentos (*procedure division*) é relativamente fraca. Cada variável é definida em detalhes na divisão de dados, incluindo o número de dígitos decimais e a localização do ponto decimal esperado. Registros de arquivos também são descritos com esse nível de detalhes, assim como o são as linhas a serem enviadas a uma impressora, tornando o COBOL ideal para imprimir relatórios contábeis. Talvez a fraqueza mais importante da divisão de procedimentos original do COBOL tenha sido sua falta de funções. Versões do COBOL anteriores ao padrão 1974 também não permitiam subprogramas com parâmetros.

Nosso comentário final sobre o COBOL: foi a primeira linguagem de programação cujo uso foi obrigatório pelo Departamento de Defesa Americano (DoD). Essa obrigatoriedade veio após o seu desenvolvimento inicial, já que o COBOL não foi especificamente projetado para o DoD. Apesar de seus méritos, o COBOL provavelmente não teria sobrevivido sem essa obrigatoriedade. O desempenho ruim dos primeiros compiladores sim-

plesmente o tornava muito caro. Eventualmente, é claro, os compiladores se tornaram mais eficientes e os computadores ficaram mais rápidos e baratos e tinham memórias muito maiores. Juntos, esses fatores permitiram o sucesso do COBOL, dentro e fora do DoD. Sua aparição levou à mecanização eletrônica da contabilidade, uma revolução importante por qualquer medida.

A seguir, temos exemplo de programa em COBOL. Ele lê um arquivo chamado `BAL-FWD-FILE`, que contém informação de inventário sobre certa coleção de itens. Dentre outras coisas, cada registro inclui o número atual de itens (`BAL-ON-HAND`) e o ponto de novo pedido (`BAL-REORDER-POINT`). O ponto de novo pedido é o número limite de itens para a solicitação de novos itens. O programa produz uma lista de itens que devem ser reorganizados como um arquivo chamado `REORDER-LISTING`.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. PRODUCE-REORDER-LISTING.

ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. DEC-VAX.
OBJECT-COMPUTER. DEC-VAX.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT BAL-FWD-FILE   ASSIGN TO READER.
    SELECT REORDER-LISTING  ASSIGN TO LOCAL-PRINTER.

DATA DIVISION.
FILE SECTION.
FD  BAL-FWD-FILE
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 80 CHARACTERS.

01  BAL-FWD-CARD.
    02 BAL-ITEM-NO        PICTURE IS 9(5).
    02 BAL-ITEM-DESC      PICTURE IS X(20).
    02 FILLER             PICTURE IS X(5).
    02 BAL-UNIT-PRICE     PICTURE IS 999V99.
    02 BAL-REORDER-POINT  PICTURE IS 9(5).
    02 BAL-ON-HAND        PICTURE IS 9(5).
    02 BAL-ON-ORDER       PICTURE IS 9(5).
    02 FILLER             PICTURE IS X(30).

FD  REORDER-LISTING
    LABEL RECORDS ARE STANDARD
    RECORD CONTAINS 132 CHARACTERS.

01  REORDER-LINE.
    02 RL-ITEM-NO         PICTURE IS Z(5).
    02 FILLER             PICTURE IS X(5).
    02 RL-ITEM-DESC       PICTURE IS X(20).
    02 FILLER             PICTURE IS X(5).
```

```
    02 RL-UNIT-PRICE      PICTURE IS ZZZ.99.
    02 FILLER             PICTURE IS X(5).
    02 RL-AVAILABLE-STOCK PICTURE IS Z(5).
    02 FILLER             PICTURE IS X(5).
    02 RL-REORDER-POINT   PICTURE IS Z(5).
    02 FILLER             PICTURE IS X(71).

WORKING-STORAGE SECTION.
01  SWITCHES.
    02 CARD-EOF-SWITCH    PICTURE IS X.
01  WORK-FIELDS.
    02 AVAILABLE-STOCK    PICTURE IS 9(5).

PROCEDURE DIVISION.
000-PRODUCE-REORDER-LISTING.
    OPEN INPUT BAL-FWD-FILE.
    OPEN OUTPUT REORDER-LISTING.
    MOVE "N" TO CARD-EOF-SWITCH.
    PERFORM 100-PRODUCE-REORDER-LINE
        UNTIL CARD-EOF-SWITCH IS EQUAL TO "Y".
    CLOSE BAL-FWD-FILE.
    CLOSE REORDER-LISTING.
    STOP RUN.

100-PRODUCE-REORDER-LINE.
    PERFORM 110-READ-INVENTORY-RECORD.
    IF CARD-EOF-SWITCH IS NOT EQUAL TO "Y"
        PERFORM 120-CALCULATE-AVAILABLE-STOCK
        IF AVAILABLE-STOCK IS LESS THAN BAL-REORDER-POINT
            PERFORM 130-PRINT-REORDER-LINE.

110-READ-INVENTORY-RECORD.
    READ BAL-FWD-FILE RECORD
        AT END
            MOVE "Y" TO CARD-EOF-SWITCH.

120-CALCULATE-AVAILABLE-STOCK.
ADD BAL-ON-HAND BAL-ON-ORDER
    GIVING AVAILABLE-STOCK.

130-PRINT-REORDER-LINE.
    MOVE SPACE             TO REORDER-LINE.
    MOVE BAL-ITEM-NO       TO RL-ITEM-NO.
    MOVE BAL-ITEM-DESC     TO RL-ITEM-DESC.
    MOVE BAL-UNIT-PRICE    TO RL-UNIT-PRICE.
    MOVE AVAILABLE-STOCK   TO RL-AVAILABLE-STOCK.
    MOVE BAL-REORDER-POINT TO RL-REORDER-POINT.
    WRITE REORDER-LINE.
```

2.7 O INÍCIO DO COMPARTILHAMENTO DE TEMPO: BASIC

BASIC (Mather e Waite, 1971) é outra linguagem de programação que teve um amplo uso, mas pouco respeito. Como o COBOL, foi ignorada pelos cientistas da computação. Além disso, como o COBOL, em suas primeiras versões a linguagem era deselegante e incluía apenas um conjunto pobre de sentenças de controle.

O BASIC era muito popular em microcomputadores no final dos anos 1970 e no início dos anos 1980, basicamente por causa de duas das suas principais características. Ele é fácil de aprender, especialmente para aqueles que não são orientados à ciência, e seus dialetos menores podem ser implementados em computadores com memórias muito pequenas[6]. Quando a capacidade dos microcomputadores cresceu e outras linguagens foram implementadas, o uso do BASIC decaiu. Uma forte revitalização no uso do BASIC surgiu com a aparição do Visual Basic (Microsoft, 1991) no início dos anos 1990.

2.7.1 Processo do projeto

O BASIC (Beginner's All-purpose Symbolic Instruction Code) foi originalmente projetado na Faculdade de Dartmouth (agora Universidade de Dartmouth) em New Hampshire por dois matemáticos, John Kemeny e Thomas Kurtz, que no início dos anos 1960 desenvolveram compiladores para uma variedade de dialetos do Fortran e do ALGOL 60. Seus estudantes de ciências básicas tinham pouco trabalho para aprender ou usar essas linguagens em seus estudos. No entanto, Dartmouth era primariamente uma instituição da área de humanas, na qual os estudantes de ciências básicas e engenharia eram apenas cerca de 25%. Na primavera de 1963, foi decidido que uma linguagem especialmente voltada para os estudantes da área de humanas seria projetada. Essa nova linguagem usaria terminais como o método de acesso a computadores. Os objetivos do sistema eram:

1. Deve ser fácil para estudantes que não são de ciências básicas a aprenderem e usarem.
2. Deve ser prazerosa e amigável.
3. Deve agilizar os deveres de casa.
4. Deve permitir acesso livre e privado.
5. Deve considerar o tempo do usuário mais importante do que o tempo do computador.

O último objetivo era um conceito revolucionário. Ele era baseado, ao menos parcialmente, na crença de que os computadores se tornariam mais baratos com o passar do tempo, o que aconteceu.

[6] Alguns dos primeiros microcomputadores incluíam interpretadores BASIC que residiam em 4096 bytes de ROM.

A combinação do segundo, terceiro e quarto objetivos levaram ao aspecto de compartilhamento de tempo do BASIC. Somente com acesso individual por terminais usados por numerosos usuários simultaneamente, esses objetivos poderiam ser alcançados no início dos anos 1960.

No verão de 1963, Kemeny começou a trabalhar no compilador para a primeira versão do BASIC, usando acesso remoto a um computador GE 225. O projeto e a codificação do sistema operacional para o BASIC começaram no final de 1963. Às 4 horas da manhã de 1º de maio de 1964, o primeiro programa usando o BASIC com compartilhamento de tempo foi digitado e executado. Em junho, o número de terminais no sistema cresceu para 11 – e, no fim do ano, para 20.

2.7.2 Visão geral da linguagem

A versão original do BASIC era bastante pequena e não era interativa: não existiam maneiras de um programa executável obter dados de entrada de um usuário. Os programas eram digitados, compilados e executados de uma maneira um tanto quanto orientada a lotes. O BASIC original tinha apenas 14 tipos diferentes de sentenças e um único tipo de dados – ponto flutuante. Como se acreditava que poucos dos usuários-alvo iriam apreciar a diferença entre tipos inteiros e de ponto flutuante, o tipo foi chamado de "*numbers*" (números). De um modo geral, era uma linguagem muito limitada, apesar de bastante fácil de aprender.

2.7.3 Avaliação

O aspecto mais importante do BASIC original foi ser a primeira linguagem amplamente usada por meio de terminais conectados a um terminal remoto[7]. Os terminais haviam recém-começado a ficar disponíveis na época. Antes disso, a maioria dos programas eram inseridos nos computadores ou por meio de cartões perfurados ou por fitas de papel.

Muito do projeto do BASIC veio do Fortran, com alguma influência da sintaxe do ALGOL 60. Posteriormente, ele cresceu em uma variedade de maneiras, com pouco ou nenhum esforço para padronizá-lo. O Instituto Nacional de Padrões dos Estados Unidos publicou um padrão mínimo para o BASIC, chamado de Minimal BASIC (ANSI, 1978b), mas ele representava apenas o mínimo essencial dos recursos da linguagem. Na verdade, o BASIC original era bastante similar ao Minimal BASIC.

Apesar de parecer surpreendente, a Digital Equipment Corporation (DEC) usou uma versão um tanto elaborada do BASIC chamada de BASIC-PLUS para escrever partes significativas de seu maior sistema operacional para os minicomputadores PDP-11, RSTS, nos anos 1970.

O BASIC foi criticado pela estrutura pobre dos programas escritos na linguagem, dentre outras coisas. Pelo critério de avaliação discutido no

[7] LISP inicialmente era empregada em terminais, mas não era usada amplamente no início dos anos 1960.

Capítulo 1, mais especificamente em relação à legibilidade e à confiabilidade, a linguagem de fato é muito pobre. As primeiras versões da linguagem não foram projetadas para serem usadas em programas sérios de qualquer tamanho significativo e não deveriam ter sido usadas para tal. As versões subsequentes eram bem mais adequadas para tais tarefas.

O ressurgimento do BASIC nos anos 1990 foi provocado pelo surgimento do Visual BASIC (VB), que se tornou amplamente usado em grande parte porque fornece uma maneira simples de construir interfaces gráficas para interação com o usuário (GUIs). O Visual Basic .NET, ou apenas VB.NET, é uma das linguagens .NET da Microsoft. Apesar de ter se distanciado do VB, ela rapidamente substituiu a linguagem mais antiga. Talvez a diferença mais importante entre o VB e o VB.NET é que o segundo suporta completamente a programação orientada a objetos. Pensava-se que os usuários do VB migrariam para uma linguagem diferente, como o C# (veja a Seção 2.19), em vez de aprender VB.NET, principalmente porque todas as linguagens .NET têm acesso às ferramentas de construção de interfaces gráficas do VB. Entretanto, isso não ocorreu – o VB.NET é mais usado do que o C#.

A seguir, temos um exemplo de um programa em BASIC:

```
REM  Programa de exemplo do BASIC
REM  Entrada:  Um inteiro, listlen, onde listlen é menor do
REM            que 100, seguido por valores inteiros listlen
REM  Saída:    O número de valores de entrada que são maiores
REM            do que a média de todos os valores de entrada
   DIM intlist(99)
   result = 0
   sum = 0
   INPUT listlen
   IF listlen > 0 AND listlen < 100 THEN
REM  Lê os dados de entrada em um vetor e calcula sua soma
   FOR counter = 1 TO listlen
      INPUT intlist(counter)
      sum = sum + intlist(counter)
   NEXT counter
REM  Calcula a média
   average = sum / listlen
REM  Conta o número de valores que são maiores do que a média
   FOR counter = 1 TO listlen
      IF intlist(counter) > average
         THEN result = result + 1
   NEXT counter
REM  Imprimir o resultado
   PRINT "The number of values that are > average is:";
         result
   ELSE
      PRINT "Error—input list length is not legal"
   END IF
END
```

Projeto de usuário e projeto de linguagens

ALAN COOPER

Autor do *best-seller About Face: The Essentials of User Interface Design*, Alan Cooper tem ampla experiência em projetar o que pode ser considerada a linguagem que mais se preocupa com o projeto de interface com o usuário, o Visual Basic. Para ele, tudo se resume ao objetivo de humanizar a tecnologia.

ALGUMAS INFORMAÇÕES BÁSICAS

Como você começou? Sou um desistente do ensino médio formado em um curso tecnólogo* em programação em uma faculdade comunitária da Califórnia. Meu primeiro emprego foi como programador na America President Lines em São Francisco (uma das companhias mais antigas de transporte oceânico dos Estados Unidos). Exceto por alguns meses aqui e ali, me mantive autônomo.

Qual é o seu trabalho atual? Fundador e presidente da Cooper, a empresa que humaniza a tecnologia (www.cooper.com).

Qual é ou foi seu trabalho favorito? Consultor de projeto de interação.

Você é muito conhecido nas áreas de projeto de linguagem e de interface com o usuário. O que você pensa a respeito de projetar linguagens *versus* projetar software *versus* projetar qualquer outra coisa? É basicamente o mesmo no mundo do software: conheça seu usuário.

SOBRE AQUELE LANÇAMENTO INICIAL DO WINDOWS

Nos anos 1980, você começou a usar o Windows e ficou entusiasmado com suas qualidades: o suporte para a interface gráfica com o usuário e as bibliotecas ligadas dinamicamente (DLLs) que permitem a criação de ferramentas que se autoconfiguram. O que você tem a dizer sobre as partes do Windows que ajudou a dar forma? Fiquei bastante impressionado com a inclusão da Microsoft do suporte para multitarefas no Windows de forma prática, que incluía realocação dinâmica e comunicação interprocessos. O MSDOS.exe era o programa de interpretação de comandos (*shell*) para os primeiros lançamentos do Windows. Era um programa terrível, e eu acreditava que poderia melhorá-lo drasticamente. Em meu tempo livre, comecei a escrever um interpretador de comandos melhor que batizei de Tripod. O interpretador de comandos original da Microsoft, o MSDOS.exe, era um dos principais elementos bloqueadores para o sucesso inicial do Windows. O Tripod tentava resolver o problema sendo mais fácil de usar e configurar.

Quando você teve essa ideia? Ela não ocorreu até o fim de 1987, quando eu estava entrevistando um cliente corporativo, e a estratégia-chave de projeto para o Tripod surgiu em minha cabeça. Como o gerente de sistemas de informação explicou sua necessidade de criar e publicar uma ampla faixa de soluções de interpretação de comandos para sua base de dados heterogênea, eu me dei conta de que o problema era a falta de tal interpretador de comandos ideal. Cada usuário precisaria de seu interpretador de comandos pessoal, configurado para suas necessidades e no seu nível de conhecimento. Em um instante, percebi a solução para o problema do projeto do interpretador de comandos: deveria ser um conjunto de construção de interpretadores de comandos, uma ferramenta na qual cada usuário seria capaz de construir exatamente o interpretador necessário para um misto único de aplicações e de treinamento.

O que é tão atraente na ideia de um interpretador de comandos que pode ser personalizado? Em vez de dizer aos usuários qual interpretador de comandos é o ideal, eles podem projetar seu próprio interpretador de comandos ideal e personalizado. Assim, um programador poderia criar um interpretador de comandos poderoso e com uma ampla faixa de ação, mas também um tanto perigoso. Ao passo que um gerente de tecnologia da informação criaria um interpretador de comandos para dar a um atendente expondo apenas as poucas ferramentas específicas que esse atendente usa.

---

* N. de T.: Do inglês *associate degree*.

> ***"MSDOS.exe era o programa de interpretação de comandos (shell) para os primeiros lançamentos do Windows. Era um programa terrível, e eu acreditava que poderia melhorá-lo drasticamente. Em meu tempo livre, comecei a escrever um interpretador de comandos melhor."***

Como você foi de escritor de um interpretador de comandos a colaborados da Microsoft? Tripod e Ruby são a mesma coisa. Depois de assinar um acordo com Bill Gates, troquei o nome do protótipo de Tripod para Ruby. Usei o protótipo de Ruby como os protótipos devem ser usados: um modelo descartável para construir código de alta qualidade. A Microsoft pegou a versão de distribuição do Ruby e adicionou Quick-BASIC a ela, criando o VB. Todas aquelas inovações originais estavam no Tripod/Ruby.

RUBY COMO INCUBADORA PARA O VISUAL BASIC

Fale sobre seu interesse nas primeiras versões do Windows e o tal recurso chamado DLL. As DLLs não eram uma coisa, eram um recurso do sistema operacional. Elas permitiam que um programador construísse objetos de código para serem ligados em tempo de execução em vez de somente em tempo de compilação. Isso me permitiu inventar as partes dinamicamente extensíveis do VB, em que os controles podem ser adicionados por terceiros.

O produto Ruby continha muitos avanços significativos em projeto de software, mas dois deles se destacam como excepcionalmente bem-sucedidos. Como mencionei, a capacidade de ligação dinâmica do Windows sempre me intrigou, mas ter as ferramentas e saber o que fazer com elas são coisas diferentes. Com Ruby, finalmente encontrei dois usos para a ligação dinâmica, e o programa original continha ambas. Primeiro, a linguagem era tanto instalável quanto poderia ser estendida. Segundo, a paleta de componentes poderia ser adicionada de maneira dinâmica.

A sua linguagem em Ruby foi a primeira a ter uma biblioteca de ligação dinâmica e ser ligada a um *front-end* visual? Pelo que sei, sim.

Usando um exemplo simples, o que isso permitia a um programador fazer com seu programa? Comprar um controle, como um controle de grade, de uma empresa qualquer, instalar em seu computador, e ter o controle de grade como se fosse parte da linguagem, incluindo o *front-end* visual de programação.

Por que o chamam de "pai do Visual Basic"? Ruby vinha com uma pequena linguagem, voltada apenas para a execução de cerca de uma dúzia de comandos simples que um interpretador de comandos precisa. Entretanto, essa linguagem era implementada como uma cadeia de DLLs, as quais poderiam ser instaladas em tempo de execução. O analisador sintático interno identificaria um verbo e então o passaria para a cadeia de DLLs até uma delas confirmavar que poderia processar o verbo. Se todas as DLLs passassem, havia um erro de sintaxe. A partir de nossas primeiras discussões, tanto a Microsoft quanto eu tínhamos gostado da ideia de aumentar a linguagem, possivelmente até mesmo substituindo-a por uma linguagem "real". C era o candidato mais mencionado, mas, a Microsoft tirou vantagem dessa interface dinâmica para substituir nossa pequena linguagem de interpretação de comandos pelo Quick-BASIC. Esse novo casamento de linguagem com um visual *front-end* era estático e permanente, e apesar da interface dinâmica original ter possibilitado o casamento, ela foi perdida no processo.

COMENTÁRIOS FINAIS SOBRE NOVAS IDEIAS

No mundo da programação e das ferramentas de programação, incluindo linguagens e ambientes, que projetos mais lhe interessam? Tenho interesse em projetar ferramentas de programação para ajudar os usuários e não os programadores.

Que regra, citação famosa ou ideia de projeto devemos sempre manter em mente? As pontes não são construídas por engenheiros, mas por metalúrgicos. De modo similar, os programas de software não são construídos por engenheiros, mas por programadores.

2.8 TUDO PARA TODOS: PL/I

PL/I representa a primeira tentativa em grande escala de linguagem que possa ser usada por um amplo espectro de áreas de aplicação. Todas as linguagens anteriores e a maioria das subsequentes focavam em uma área de aplicação específica, como aplicações científicas, de inteligência artificial ou de negócios.

2.8.1 Perspectiva histórica

Como o Fortran, PL/I foi desenvolvida como um produto da IBM. No início dos anos 1960, os usuários de computadores na indústria haviam se instalado em dois campos separados e bastante diferentes: aplicações científicas e aplicações de negócios. Do ponto de vista da IBM, os programadores científicos poderiam usar ou os computadores IBM 7090 de grande porte ou os 1620 de pequeno porte. Tais programadores utilizavam dados em formato de ponto flutuante e vetores extensivamente. Fortran era a principal linguagem, apesar de alguma linguagem de montagem também ser usada. Eles tinham seu próprio grupo de usuários, chamado SHARE, e pouco contato com qualquer um que trabalhasse em aplicações de negócios.

Para aplicações de negócios, as pessoas usavam os computadores da IBM de grande porte 7080 ou os de pequeno porte 1401. Elas precisavam de tipos de dados decimais e de cadeias de caracteres, assim como recursos elaborados e eficientes para entrada e saída. Elas usavam COBOL, apesar de no início de 1963, quando a história da PL/I começou, a conversão de linguagem de montagem para COBOL não estar completa. Essa categoria de usuários também tinha seu próprio grupo de usuários, chamado GUIDE, e raramente mantinha contato com usuários científicos.

No início de 1963, os planejadores da IBM perceberam uma mudança nessa situação. Os dois grupos amplamente separados estavam se aproximando um do outro, o que se pensava que causaria problemas. Os cientistas começaram a obter grandes arquivos de dados para serem processados. Esses dados requeriam recursos de entrada e saída mais sofisticados e eficientes. Os profissionais das aplicações de negócios começaram a usar análise de regressão para construir sistemas de informação de gerenciamento, os quais requeriam dados de ponto flutuante e vetores. Parecia que as instalações de computação logo precisariam de duas equipes técnicas e de computadores diferentes, de forma a oferecer suporte para duas linguagens de programação bastante diferentes[8].

Essas percepções levaram de forma bastante natural ao conceito de projetar um computador universal que deveria ser capaz de realizar tanto operações de ponto flutuante quanto aritmética decimal – e, dessa forma, suportar tanto aplicações científicas quanto comerciais. Nasceu então o conceito da linha de computadores IBM System/360. Com isso, veio a ideia de uma linguagem de programação que poderia ser usada tanto para

[8] Na época, grandes instalações de computação requeriam equipes de manutenção em tempo integral tanto para hardware quanto para software de sistema.

aplicações comerciais quanto para aplicações científicas. Para atender tais aplicações, recursos para suportar a programação de sistemas e para o processamento de listas foram adicionados. Assim, a nova linguagem viria para substituir o Fortran, o COBOL, o LISP e as aplicações de sistemas das linguagens de montagem.

2.8.2 O processo de projeto

O esforço de projeto começou quando a IBM e o SHARE formaram o Comitê de Desenvolvimento de Linguagem Avançada do Projeto Fortran do SHARE em outubro de 1963. Esse novo comitê rapidamente se encontrou e formou um subcomitê chamado de Comitê 3-3, nomeado assim porque era formado por três membros da IBM e três do SHARE. O Comitê 3-3 se encontrava três ou quatro dias, uma semana sim, uma não, para projetar a linguagem.

Como ocorreu com o Comitê Short Range para o COBOL, o projeto inicial foi planejado para ser completado em um tempo excepcionalmente curto. Aparentemente, independentemente do escopo de um esforço de projeto de linguagem, no início dos anos 1960, acreditava-se que ele poderia ser feito em três meses. A primeira versão de PL/I, chamada de Fortran VI, supostamente deveria estar completa em dezembro, menos de três meses após a formação do comitê. Ele obteve extensões de prazo em duas ocasiões, movendo a data de finalização para janeiro e, posteriormente, para o final de fevereiro de 1964.

O conceito inicial de projeto era que a nova linguagem seria uma extensão do Fortran IV, mantendo a compatibilidade. Mas esse objetivo foi abandonado rapidamente, assim como o nome Fortran VI. Até 1965, a linguagem era conhecida como NPL (New Programming Language – Nova Linguagem de Programação). O primeiro relatório publicado sobre a NPL foi apresentado na reunião do grupo SHARE em março de 1964. Uma descrição mais completa foi apresentada em abril, e a versão que seria implementada foi publicada em dezembro de 1964 (IBM, 1964) pelo grupo de compiladores no Laboratório Hursley da IBM na Inglaterra, escolhido para a implementação. Em 1965, o nome foi trocado para PL/I para evitar a confusão do nome NPL com o National Physical Laboratory na Inglaterra. Se o compilador tivesse sido desenvolvido fora do Reino Unido, o nome poderia ter permanecido NPL.

2.8.3 Visão geral da linguagem

Talvez a melhor descrição em uma única sentença da linguagem PL/I é que ela incluía o que eram consideradas as melhores partes do ALGOL 60 (recursão e estrutura de bloco), Fortran IV (compilação separada com comunicação por meio de dados globais) e COBOL 60 (estruturas de dados, entrada e saída e recursos para a geração de relatórios), além de uma extensa coleção de novas construções, todas unidas de maneira improvisada. Como

PL/I é agora uma linguagem quase morta, não tentaremos, mesmo que de maneira abreviada, discutir todos os recursos da linguagem ou mesmo suas construções mais controversas. Em vez disso, iremos mencionar algumas das contribuições aos conhecimentos acerca de linguagens de programação.

PL/I foi a primeira linguagem a ter os seguintes recursos:

- Era permitido aos programas criar subprogramas executados concorrentemente.
- Era possível detectar e manipular 23 tipos diferentes de exceções ou erros em tempo de execução.
- Era permitida a utilização de subprogramas recursivamente, mas tal mecanismo podia ser desabilitado, o que permitia uma ligação mais eficiente para subprogramas não recursivos.
- Ponteiros foram incluídos como um tipo de dados.
- Porções de uma matriz podiam ser referenciadas. Por exemplo, a terceira linha de uma matriz poderia ser referenciada como se fosse um vetor.

2.8.4 Avaliação

Quaisquer avaliações de PL/I devem começar reconhecendo a ambição do esforço de projeto. Retrospectivamente, parece ingenuidade pensar que tantas construções poderiam ser combinadas com sucesso. Entretanto, tal julgamento deve levar em consideração que existia pouca experiência em projeto de linguagens na época. De um modo geral, o projeto da PL/I era baseado na premissa de que qualquer construção útil e que poderia ser implementada deveria ser incluída, com poucas preocupações sobre como um programador poderia entender e usar efetivamente tal coleção de construções e recursos. Edsger Dijkstra, em sua Palestra do Prêmio Turing (Dijkstra, 1972), fez uma das críticas mais contundentes a respeito da complexidade de PL/I: "Eu não consigo ver como poderemos manter o crescimento de nossos programas de maneira firme, dentro de nossas capacidades intelectuais, quando simplesmente a complexidade e a irregularidade da linguagem de programação – nossa ferramenta básica, vejam só – já fogem de nosso controle intelectual".

Além do problema da complexidade por conta de seu tamanho, PL/I sofria também pelo fato de ter diversas construções que atualmente são consideradas pobremente projetadas. Dentre essas estavam os ponteiros, o tratamento de exceções e concorrência, apesar de que devemos dizer que em cada um desses casos as construções não haviam aparecido em nenhuma linguagem anterior.

Em termos de uso, a PL/I deve ser considerada ao menos parcialmente bem-sucedida. Nos anos 1970, ela desfrutou de um uso significativo tanto em aplicações comerciais quanto científicas. Ela também foi bastante usada como um veículo de ensino em faculdades, principalmente em formatos que eram subconjuntos da linguagem, como PL/C (Cornell, 1977) e PL/CS (Conway e Constable, 1976).

A seguir, temos um exemplo de um programa em PL/I:

```
/* PROGRAMA DE EXEMPLO DE PL/I
 ENTRADA:  UM INTEIRO, LISTLEN, ONDE LISTLEN É MENOR DO QUE
           100, SEGUIDO POR VALORES INTEIROS LISTLEN
 SAÍDA:    UM INTEIRO, LISTLEN, ONDE LISTLEN É MENOR DO QUE
           100, SEGUIDO POR VALORES INTEIROS LISTLEN  */
PLIEX: PROCEDURE OPTIONS (MAIN);
  DECLARE INTLIST (1:99) FIXED.
  DECLARE (LISTLEN, COUNTER, SUM, AVERAGE, RESULT) FIXED;
  SUM = 0;
  RESULT = 0;
  GET LIST (LISTLEN);
  IF (LISTLEN > 0) & (LISTLEN < 100) THEN
    DO;
/* LÊ OS DADOS DE ENTRADA EM UM VETOR E CALCULA SUA SOMA */
    DO COUNTER = 1 TO LISTLEN;
      GET LIST (INTLIST (COUNTER));
      SUM = SUM + INTLIST (COUNTER);
    END;
/* CALCULA A MÉDIA */
    AVERAGE = SUM / LISTLEN;
/* CONTA O NÚMERO DE VALORES QUE SÃO MAIORES QUE A MÉDIA */
    DO COUNTER = 1 TO LISTLEN;
      IF INTLIST (COUNTER) > AVERAGE THEN
        RESULT = RESULT + 1;
    END;
/* IMPRIMIR O RESULTADO */
    PUT SKIP LIST ('THE NUMBER OF VALUES > AVERAGE IS:');
    PUT LIST (RESULT);
    END;
  ELSE
    PUT SKIP LIST ('ERROR—INPUT LIST LENGTH IS ILLEGAL');
  END PLIEX;
```

2.9 DUAS DAS PRIMEIRAS LINGUAGENS DINÂMICAS: APL E SNOBOL

A estrutura desta seção é diferente das outras deste capítulo, porque as linguagens discutidas aqui são bastante diferentes. Nem APL nem SNOBOL foram baseadas em linguagens prévias e nenhuma delas teve muita influência em linguagens utilizadas posteriormente[9]. Alguns dos recursos interessantes de APL são discutidos mais adiante.

[9] Entretanto, elas tiveram alguma influência em algumas linguagens que não são muito usadas (J é baseada em APL, ICON é baseada em SNOBOL, e AWK é parcialmente baseada em SNOBOL).

Tanto em relação à aparência quanto ao propósito, APL e SNOBOL são muito diferentes. Elas compartilham, entretanto, duas características fundamentais: tipagem dinâmica e alocação dinâmica de armazenamento. Variáveis em ambas as linguagens são essencialmente não tipadas. Uma variável adquire um tipo quando lhe atribuem um valor, ou seja, quando ela assume o tipo desse valor. O armazenamento, por sua vez, é alocado a uma variável apenas quando lhe é atribuído um valor, já que antes disso não existe uma maneira de saber a quantidade de armazenamento necessário.

2.9.1 Origens e características da APL

APL (Brown et al., 1988) foi projetada em torno de 1960 por Kenneth E. Iverson na IBM. Ela não foi originalmente projetada para ser uma linguagem de programação implementada, mas para ser um veículo para descrever arquiteturas de computadores. A APL foi descrita inicialmente no livro do qual obtém seu nome, *A Programming Language* (Iverson, 1962). No meio dos anos 1960, a primeira implementação de APL foi desenvolvida na IBM.

APL tem vários operadores poderosos, os quais criaram um problema para os implementadores. As primeiras formas de uso da APL foram por meio de terminais de impressão IBM. Tais terminais tinham elementos de impressão especiais que forneciam o estranho conjunto de caracteres requerido pela linguagem. Uma das razões pelas quais APL tem tantos operadores é que ela fornece um grande número de operações unitárias em vetores. Por exemplo, a transposição de qualquer matriz é feita com um único operador. A grande coleção de operadores fornece uma alta expressividade, mas também faz os programas APL serem de difícil leitura. Isso levou as pessoas a pensarem em APL como uma linguagem mais bem usada para programação "descartável". Apesar de os programas serem escritos rapidamente, eles poderiam ser descartados após seu uso porque seriam de difícil manutenção.

APL está ativa por cerca de 45 anos e ainda é usada, apesar de não amplamente, e não mudou muito em todo esse tempo de vida.

2.9.2 Origens e características do SNOBOL

SNOBOL (pronuncia-se "snowball"; Griswold et al., 1971) foi projetada no início dos anos 1960 por três pessoas no Laboratório Bell: D. J. Farber, R. E. Griswold e I. P. Polonsky (Farber et al., 1964), especificamente para o processamento de texto. O coração do SNOBOL é uma coleção de operações poderosas para o casamento de padrões de cadeias. Uma de suas primeiras aplicações foi a escrita de editores de texto. Como a natureza dinâmica de SNOBOL a torna mais lenta do que linguagens alternativas, ela não é mais usada para tais programas. Entretanto, ainda é uma linguagem viva e com suporte usada para uma variedade de tarefas de processamento de textos em diferentes áreas de aplicação.

2.10 O INÍCIO DA ABSTRAÇÃO DE DADOS: SIMULA 67

Apesar de o SIMULA 67 nunca ter atingido um amplo uso e ter tido pouco impacto nos programadores e na computação de sua época, algumas das construções que introduziu o tornam historicamente importante.

2.10.1 Processo de projeto

Dois noruegueses, Kristen Nygaard e Ole-Johan Dahl, desenvolveram a linguagem SIMULA I entre 1962 e 1964, no Centro de Computação Norueguês (NCC). Eles estavam inicialmente interessados em usar computadores para simulação, mas também trabalhavam em pesquisa operacional. SIMULA I foi projetado exclusivamente para a simulação de sistemas e implementado pela primeira vez no final de 1964 em um computador UNIVAC 1107.

Quando a implementação do SIMULA I estava completa, Nygaard e Dahl começaram os esforços para estender a linguagem com novos recursos e modificações de algumas das construções existentes para torná-la útil para aplicações de propósito geral. O resultado desse trabalho era o SIMULA 67, cujo projeto foi apresentado publicamente pela primeira vez em março de 1967 (Dahl e Nygaard, 1967). Discutiremos apenas o SIMULA 67, apesar de alguns de seus recursos interessantes também estarem disponíveis no SIMULA I.

2.10.2 Visão geral da linguagem

O SIMULA 67 é uma extensão do ALGOL 60, com sua estrutura de bloco e suas sentenças de controle. A principal deficiência do ALGOL 60 (e de outras linguagens da época) para aplicações de simulação era o projeto de seus subprogramas. As simulações precisam de subprogramas que possam ser reiniciados na posição em que foram previamente parados. Subprogramas com esse tipo de controle são chamados de **corrotinas,** porque o chamador e os subprogramas chamados têm um relacionamento de certa forma igualitário, em vez do mestre/escravo rígido existente na maioria das linguagens imperativas.

Para fornecer suporte às corrotinas no SIMULA 67, a construção "classe" foi desenvolvida, determinando um avanço importante já que foi como começou o conceito de abstração de dados. A definição de uma estrutura de dados e as rotinas que manipulam suas instâncias são empacotadas em uma mesma unidade gerando a ideia básica de uma classe. Além disso, uma definição de classe é apenas um modelo para uma estrutura de dados e, dessa forma, é distinta de uma instância de classe, de maneira que um programa pode criar e usar qualquer número de instâncias de uma classe específica. Instâncias de classes podem conter dados locais e incluir código, executado em tempo de criação, podendo inicializar alguma estrutura de dados da instância de classe.

Uma discussão mais aprofundada de classes e instâncias de classes é apresentada no Capítulo 11. É interessante notar que o importante conceito de abstração de dados não foi desenvolvido e atribuído à construção classe até 1972, quando Hoare (1972) reconheceu a conexão.

2.11 PROJETO ORTOGONAL: ALGOL 68

O ALGOL 68 foi fonte de diversas ideias novas no projeto de linguagens, algumas adotadas por outras linguagens. O incluímos aqui por essa razão, apesar de ele nunca ter atingido um uso amplo nem na Europa, nem nos Estados Unidos.

2.11.1 Processo de projeto

O desenvolvimento da família ALGOL não terminou quando o relatório revisado apareceu em 1962, apesar de demorar seis anos até que a próxima iteração de projeto tenha sido publicada. A linguagem resultante, o ALGOL 68 (van Wijngaarden et al.,1969) era drasticamente diferente de seus predecessores.

Uma das inovações mais interessantes do ALGOL 68 foi em relação a um de seus critérios de projeto primários: ortogonalidade. O uso da ortogonalidade resultou em diversos recursos inovadores no ALGOL 68, um deles descrito na seção seguinte.

2.11.2 Visão geral da linguagem

Um resultado importante da ortogonalidade no ALGOL 68 foi a inclusão dos tipos de dados definidos pelo usuário. Linguagens anteriores, como o Fortran, incluíam apenas algumas estruturas de dados básicas. PL/I incluiu um número maior de estruturas de dados) o que a tornou mais difícil de aprender e de implementar) mas obviamente ela não poderia fornecer uma estrutura de dados apropriada para cada necessidade.

A abordagem do ALGOL 68 para as estruturas de dados era fornecer alguns tipos primitivos e permitir que o usuário os combinasse em um grande número de estruturas. Esse recurso de fornecer tipos de dados definidos pelo usuário foi usado por todas as principais linguagens imperativas projetadas desde então. Os tipos de dados definidos pelo usuário são valiosos, porque permitem que ele projete abstrações de dados que se encaixem com os problemas em particular de uma maneira muito forte. Todos os aspectos de tipos de dados são discutidos no Capítulo 6.

O ALGOL 68 também introduziu um tipo de vetores dinâmicos que serão chamados de "implícitos dinâmicos do monte" no Capítulo 5. Um vetor dinâmico é um no qual a declaração não especifica os limites dos índices. Atribuições a um vetor dinâmico fazem com que o armazenamento necessário seja alocado em tempo de execução. Em ALGOL 68, os vetores dinâmicos são chamados de vetores `flex`.

2.11.3 Avaliação

O ALGOL 68 inclui um número significativo de recursos. Seu uso da ortogonalidade, o qual alguns podem argumentar que era demasiado, foi revolucionário.

Entretanto, o ALGOL 68 repetiu uma das sinas do ALGOL 60, um importante fator para sua popularidade limitada: a linguagem era descrita com uma metalinguagem elegante e concisa, porém desconhecida. Antes que alguém pudesse ler o documento que descrevia a linguagem (van Wijngaarden et al., 1969), teria de aprender a nova metalinguagem, chamada de gramáticas de van Wijngaarden. Para piorar, os projetistas inventaram uma coleção de palavras para explicar a gramática e a linguagem. Por exemplo, as palavras-chave eram chamadas de *indicativos*, a extração de subcadeias era chamada de *redução* e o processo de execução de procedimentos era chamado de *coerção de desprocedimento*, a qual poderia ser *obediente*, *firme* ou alguma outra coisa.

É natural avaliar o contraste do projeto de PL/I com o do ALGOL 68. O ALGOL 68 atingiu uma boa facilidade de escrita por meio do princípio da ortogonalidade: alguns conceitos primitivos e o uso irrestrito de alguns mecanismos de combinação. PL/I atingiu uma boa facilidade de escrita com a inclusão de um grande número de construções fixas. O ALGOL 68 estendeu a simplicidade elegante do ALGOL 60, enquanto PL/I simplesmente atirou os recursos de diversas linguagens em um mesmo recipiente para atingir seus objetivos. É claro, precisamos manter em mente que o objetivo da linguagem PL/I era fornecer uma ferramenta unificada para uma ampla classe de problemas; por outro lado, o ALGOL 68 era focado em uma classe: aplicações científicas.

PL/I atingiu uma aceitação muito maior do que o ALGOL 68, principalmente pelos esforços promocionais da IBM e pelos problemas de entendimento e de implementação do ALGOL 68. A implementação era difícil para ambas as linguagens, mas PL/I tinha os recursos da IBM para aplicar na construção de um compilador. O ALGOL 68 não desfrutava de tal benfeitor.

2.12 ALGUNS DOS PRIMEIROS DESCENDENTES DOS ALGOLS

Todas as linguagens imperativas, incluindo aquelas que oferecem suporte à programação orientada a objetos, devem algo de seu projeto ao ALGOL 60 e/ou ao ALGOL 68. Esta seção discute alguns dos primeiros descendentes dessas linguagens.

2.12.1 Simplicidade por meio do projeto: Pascal

2.12.1.1 Perspectiva histórica

Niklaus Wirth (Wirth é pronunciado "Virt") era membro do Grupo de Trabalho 2.1 da IFIP (International Federation of Information Processing – Federação Internacional de Processamento de Informações), criado para con-

tinuar o desenvolvimento do ALGOL em meados dos anos 1960. Em agosto de 1965, Wirth e C. A. R. ("Tony") Hoare contribuíram para esse esforço ao apresentar ao grupo uma proposta modesta de adições e modificações ao ALGOL 60 (Wirth e Hoare, 1966). A maioria do grupo rejeitou a proposta como um avanço muito pequeno em relação ao ALGOL 60. Em vez disso, uma revisão muito mais complexa foi realizada – o que, no fim das contas, tornou-se o ALGOL 68. Wirth, com membros desse grupo, não acreditava que o relatório do ALGOL 68 deveria ter sido publicado, devido à complexidade tanto da linguagem quanto da metalinguagem usada para descrevê-la. Tal posicionamento se provou válido, já que os documentos do ALGOL 68 (e dessa forma a linguagem) foram tidos como desafiadores para a comunidade de computação.

A versão de Wirth e Hoare do ALGOL 60 foi nomeada ALGOL-W. Ela foi implementada na Universidade de Stanford e usada basicamente como um veículo educacional, mas apenas em algumas universidades. As principais contribuições do ALGOL-W eram o método de passagem de parâmetros por valor-resultado e a sentença `case` para seleção múltipla. O método valor-resultado é uma alternativa ao método por nome do ALGOL 60. Ambos são discutidos no Capítulo 9. A sentença `case` é discutida no Capítulo 8.

O próximo grande esforço de projeto de Wirth, mais uma vez baseado no ALGOL 60, foi seu mais bem-sucedido: Pascal[10]. A definição do Pascal publicada originalmente apareceu em 1971 (Wirth, 1971). Essa versão foi modificada no processo de implementação e é descrita em Wirth (1973). Os recursos geralmente creditados ao Pascal vieram de linguagens anteriores. Por exemplo, os tipos de dados definidos pelo usuário foram introduzidos no ALGOL 68, a sentença `case` no ALGOL-W, e os registros do Pascal são similares àqueles do COBOL e da linguagem PL/I.

2.12.1.2 Avaliação

O maior impacto do Pascal foi no ensino de programação. Em 1970, a maioria dos estudantes de ciência da computação, engenharia e ciências básicas começava seus estudos em programação com o Fortran, apesar de algumas universidades usarem PL/I, linguagens baseadas em PL/I e ALGOL-W. No meio dos anos 1970, o Pascal se tornou a linguagem mais usada para esse propósito. Isso era bastante natural, já que Pascal foi especificamente projetada para ensinar programação. Não foi antes do final dos anos 1990 que o Pascal deixou de ser a linguagem mais usada para o ensino de programação em faculdades e universidades.

Como o Pascal foi projetado como uma linguagem de ensino, ele não tinha diversos recursos essenciais para muitos tipos de aplicação. O melhor

[10] Pascal foi nomeada em homenagem a Blaise Pascal, um filósofo e matemático francês do século XVII que inventou a primeira máquina mecânica de adição em 1642 (dentre outras coisas).

exemplo disso é a impossibilidade de escrever um subprograma que recebe como parâmetro um vetor de tamanho variável. Outro exemplo é a falta de quaisquer capacidades de compilação de arquivos separados. Essas deficiências levaram a muitos dialetos não padronizados, como o Turbo Pascal.

A popularidade do Pascal, tanto para o ensino de programação quanto para outras aplicações, era baseada em sua excepcional combinação de simplicidade e expressividade. Apesar de existirem algumas inseguranças na linguagem, o Pascal é, mesmo assim, relativamente seguro, especialmente quando comparado com Fortran ou C. Em meado dos anos 1990, a popularidade do Pascal estava em declínio, tanto na indústria quanto nas universidades, principalmente devido à escalada de Modula-2, Ada e C++, todas as quais incluíam recursos que não estavam disponíveis no Pascal.

A seguir, temos um exemplo de um programa em Pascal:

```
{Programa de exemplo em Pascal
 Entrada:  Um inteiro, listlen, onde listlen é menor do que
           100, seguido por valores inteiros listlen
 Saída:    O número de valores de entrada que são maiores do
           que a média de todos os valores de entrada }
program pasex (input, output);
  type intlisttype = array [1..99] of integer;
  var
    intlist : intlisttype;
    listlen, counter, sum, average, result : integer;
  begin
  result := 0;
  sum := 0;
  readln (listlen);
  if ((listlen > 0) and (listlen < 100)) then
    begin
{ Lê os dados de entrada em um vetor e calcula sua soma }
    for counter := 1 to listlen do
      begin
      readln (intlist[counter]);
      sum := sum + intlist[counter]
      end;
{ Calcula a média }
    average := sum / listlen;
{ Conta o número de valores que são maiores do que a média }
    for counter := 1 to listlen do
      if (intlist[counter] > average) then
        result := result + 1;
{ Imprimir o resultado }
    writeln ('The number of values > average is:',
             result)
    end { of the then clause of if ((listlen > 0 ... }
```

```
  else
    writeln ('Error—input list length is not legal')
end.
```

2.12.2 Uma linguagem de sistemas portável: C

Assim como Pascal, C contribuiu pouco para a coleção de recursos de linguagem conhecidos previamente, mas tem sido muito usada por um longo período de tempo. Apesar de originalmente projetada para a programação de sistemas, a linguagem C é bastante adequada para uma variedade de aplicações.

2.12.2.1 Perspectiva histórica

Os ancestrais do C incluem CPL, BCPL, B e ALGOL 68. CPL foi desenvolvida na Universidade de Cambridge no início dos anos 1960. BCPL é uma linguagem de sistemas simples desenvolvida por Martin Richards em 1967 (Richards, 1969).

O primeiro trabalho no sistema operacional UNIX foi feito no fim dos anos 1960 por Ken Thompson nos Laboratórios da Bell (Bell Labs) e a primeira versão foi escrita em linguagem de montagem. A primeira linguagem de alto nível implementada no UNIX foi B, que era baseada no BCPL. B foi projetada e implementada por Thompson em 1970.

Nem BCPL nem B são linguagens tipadas, o que é estranho entre as linguagens de alto nível, apesar de ambas serem de muito mais baixo nível do que Java, por exemplo. Ser não tipada significa que todos os dados são considerados palavras de máquina, que levam a muitas complicações e inseguranças. Por exemplo, existe o problema de especificar pontos flutuantes em vez de aritméticas de inteiros em uma expressão. Em uma implementação de BCPL, as variáveis que atuavam como operandos de uma operação de ponto flutuante eram precedidas de pontos. Variáveis que eram usadas como operandos não precedidas de pontos eram consideradas inteiras. Uma alternativa a isso seria o uso de símbolos diferentes para os operadores de ponto flutuante.

Esse problema, com diversos outros, levou ao desenvolvimento de uma nova linguagem tipada baseada em B. Originalmente chamada de NB, mas posteriormente nomeada de C, ela foi projetada e implementada por Dennis Ritchie no Bell Labs em 1972 (Kernighan e Ritchie, 1978). Em alguns casos por meio de BCPL, e em outros diretamente, a linguagem C foi influenciada pelo ALGOL 68. Isso é visto em suas sentenças **`for`** e **`switch`**, em seus operadores de atribuição e em seu tratamento de ponteiros.

O único "padrão" para C em sua primeira década e meia era o livro de Kernighan e Ritchie (1978)[11]. Ao longo desse período, a linguagem evoluiu lentamente, com diferentes implementadores adicionando recursos. Em 1989, a ANSI produziu uma descrição oficial de C (ANSI, 1898), que incluía muitos dos recursos que os implementadores haviam incorporado na linguagem. Esse

[11] Essa linguagem é geralmente chamada de "K&R C".

padrão foi atualizado em 1999 (ISO, 1999), com mudanças significativas à linguagem. A versão 1989, chamada por muito tempo de ANSI C, passou a se chamar C89. Iremos nos referir à versão de 1999 como C99.

2.12.2.2 Avaliação

C tem sentenças de controle adequadas e recursos para a utilização de estruturas de dados que permitem seu uso em muitas áreas de aplicação. Ela também tem um rico conjunto de operadores que fornecem um alto grau de expressividade.

Uma das principais razões pelas quais a linguagem C é tão admirada como odiada é sua falta de uma verificação de tipos completa. Por exemplo, em versões anteriores ao C99, podiam ser escritas funções para os quais os parâmetros não eram verificados em relação ao tipo. Aqueles que gostam de C apreciam a flexibilidade; aqueles que não gostam o acham muito inseguro. Uma das grandes razões para seu grande aumento em popularidade nos anos 1980 foi o fato de que um compilador para a linguagem era parte do amplamente usado sistema operacional UNIX. Essa inclusão no UNIX forneceu um compilador bastante bom e essencialmente livre que estava disponível para os programadores em muitos tipos de computadores.

A seguir, temos um exemplo de um programa em C:

```
/* Programa de exemplo em C
 Entrada:  Um inteiro, listlen, onde listlen é menor do que
           100, seguido por valores inteiros listlen
 Saída:    O número de valores de entrada que são maiores do
           que a média de todos os valores de entrada */
int  main (){
  int intlist[99], listlen, counter, sum, average, result;
  sum = 0;
  result = 0;
  scanf("%d", &listlen);
  if  ((listlen > 0) && (listlen < 100)) {
/* Lê os dados de entrada em um vetor e calcula sua soma */
     for (counter = 0; counter < listlen; counter++) {
      scanf("%d", &intlist[counter]);
      sum += intlist[counter];
     }
/* Calcula a média */
    average = sum / listlen;
/* Conta o número de valores que são maiores do que a média */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Imprimir o resultado */
    printf("Number of values > average is:%d\n", result);
   }
  else
    printf("Error—input list length is not legal\n");
 }
```

2.13 PROGRAMAÇÃO BASEADA EM LÓGICA: PROLOG

Explicando de uma forma simples, a programação lógica é o uso de uma notação lógica formal para comunicar processos computacionais para um computador. O cálculo de predicados é a notação usada nas linguagens de programação lógica atuais.

A programação em linguagens de programação lógica é não procedural. Os programas não exprimem exatamente *como* um resultado deve ser computado, mas descrevem a forma necessária e/ou as características dele. O que é preciso para fornecer essa capacidade em linguagens de programação lógica é uma maneira concisa de disponibilizar ao computador tanto a informação relevante quanto um processo de inferência para computar os resultados desejados. O cálculo de predicados fornece a forma básica de comunicação com o computador. E o método de prova, a resolução de nomes, desenvolvida inicialmente por Robinson (1965), fornece a técnica de inferência.

2.13.1 Processo de projeto

Durante o início dos anos 1970, Alain Colmerauer e Phillippe Roussel do Grupo de Inteligência Artificial da Universidade de Aix-Marseille, com Robert Kowalski do Departamento de Inteligência Artificial da Universidade de Edimburgo, desenvolveram o projeto fundamental de Prolog. Os componentes primários do Prolog são um método para a especificação de proposições de cálculo de predicados e uma implementação de uma forma restrita de resolução. Tanto o cálculo de predicados quanto a resolução são descritos no Capítulo 16. O primeiro interpretador Prolog foi desenvolvido em Marselha em 1972. A versão da linguagem que foi implementada é descrita em Roussel (1975). O nome Prolog vem de *pro*gramação *ló*gica.

2.13.2 Visão geral da linguagem

Os programas em Prolog consistem em coleções de sentenças. Prolog tem apenas alguns tipos de sentenças, mas elas podem ser complexas.

Um uso comum de Prolog é como um tipo de base de dados inteligente. Essa aplicação fornece um *framework* simples para discutir a linguagem.

A base de dados de um programa Prolog consiste em dois tipos de sentenças: fatos e regras. Exemplos de sentenças factuais são:

```
mother(joanne, jake).
father(vern, joanne).
```

Essas sentenças afirmam que `joanne` é a mãe (`mother`) de `jake`, e que `vern` é o pai (`father`) de `joanne`.

Um exemplo de uma regra é

```
grandparent(X, Z) :- parent(X, Y), parent(Y, Z).
```

A regra afirma que se pode deduzir que X é o avô ou a avó de Z se for verdade que X é o pai ou a mãe de Y e que Y é o pai ou a mãe de Z, para alguns valores específicos para as variáveis X, Y e Z.

A base de dados Prolog pode ser consultada interativamente com sentenças-objetivo, um exemplo do qual é

```
father(bob, darcie).
```

Tal sentença pergunta se bob é o pai (father) de darcie. Quando tal consulta, ou objetivo, é apresentada para o sistema Prolog, ele usa seu processo de resolução para tentar determinar a verdade da sentença. Se conseguir concluir que o objetivo é verdadeiro, ele mostra "true". Se não puder prová-lo, ele mostra "false".

2.13.3 Avaliação

Nos anos 1980, um grupo relativamente pequeno de cientistas da computação acreditava que a programação lógica fornecia a melhor esperança para escapar da complexidade das linguagens imperativas e do problema de produzir a imensa quantidade de software confiável necessária. Até agora, existem duas grandes razões pelas quais a programação lógica não se tornou mais usada. Primeiro, como acontece com outras abordagens não imperativas, os programas escritos em linguagens lógicas têm provado ser ineficientes se comparados aos programas imperativos equivalentes. Segundo, foi determinado que essa abordagem é efetiva para apenas algumas áreas de aplicação: certos tipos de sistemas de gerenciamento de bancos de dados e áreas de IA.

Existe um dialeto de Prolog que oferece suporte à programação orientada a objetos – Prolog++ (Moss, 1994). Programação lógica e Prolog são descritos em maiores detalhes no Capítulo 16.

2.14 O MAIOR ESFORÇO DE PROJETO DA HISTÓRIA: ADA

A linguagem Ada é resultado do mais extenso e caro esforço de projeto de uma linguagem de programação da história. Os parágrafos a seguir descrevem brevemente a evolução de Ada.

2.14.1 Perspectiva histórica

A linguagem Ada foi desenvolvida para o Departamento de Defesa dos Estados Unidos (DoD) – logo, o estado do ambiente de computação do DoD foi fundamental para determinar sua forma. Em 1974, quase metade das aplicações de computadores no DoD eram sistemas embarcados, nos quais a parte de hardware do computador é embarcada no dispositivo que ele controla ou para o qual ele fornece serviços. Os custos de software estavam crescendo

rapidamente, principalmente devido à crescente complexidade dos sistemas. Mais de 450 linguagens de programação estavam em uso para projetos do DoD, e nenhuma delas era padronizada pelo DoD. Cada contratado para um projeto de defesa podia definir uma linguagem nova e diferente para cada contrato[12]. Devido a essa proliferação de linguagens, os aplicativos de software raramente eram reutilizados. Além disso, nenhuma ferramenta de software havia sido criada (porque elas são normalmente dependentes de linguagens). Muitas linguagens excelentes estavam sendo usadas, mas nenhuma era adequada para aplicações de sistemas embarcados. Por essas razões, o Exército, a Marinha e a Força Aérea dos Estados Unidos propuseram, cada uma delas independentemente, em 1974, o desenvolvimento de uma única linguagem de alto nível para sistemas embarcados.

2.14.2 Processo de projeto

Ao notar esse amplo interesse, Malcolm Currie, diretor de pesquisa e engenharia de defesa, em janeiro de 1975, formou o High-Order Language Working Group (HOLWG – Grupo de Trabalho para Linguagem de Alto Nível), inicialmente liderado pelo tenente-coronel William Whitaker, da Força Aérea. O HOLWG tinha representantes de todos os serviços militares americanos e colaboradores da Grã-Bretanha, França e Alemanha Ocidental. Seu objetivo inicial era:

- Identificar os requisitos para uma nova linguagem de alto nível para o DoD.
- Avaliar linguagens existentes para determinar se existia uma candidata viável.
- Recomendar a adoção ou implementação de um conjunto mínimo de linguagens de programação.

Em abril de 1975, o HOLWG produziu o documento de requisitos inicial (Strawman – Homem de Palha) para a nova linguagem (Departament of Defense, 1975a). Ele foi distribuído para órgãos militares, agências federais, representantes selecionados da indústria e das universidades e colaboradores interessados na Europa.

O documento Strawman foi seguido pelo Woodenman (Homem de Madeira) em agosto de 1975 (Departament of Defense, 1975b), Tinman (Homem de Estanho) em janeiro de 1976 (Departament of Defense, 1976), Ironman (Homem de Ferro) em janeiro de 1977 (Departament of Defense, 1977) e finalmente Steelman (Homem de Aço) em junho de 1978 (Departament of Defense, 1978).

Após um tedioso processo, as propostas submetidas para a linguagem foram reduzidas a quatro finalistas, todas baseadas no Pascal. Em maio de 1979, a proposta de projeto da Cii Honeywell/Bull foi escolhida a partir

[12] Isso ocorria devido ao uso disseminado de linguagens de montagem para sistemas embarcados, cuja maioria usava processadores especializados.

dos quatro finalistas como o projeto que seria usado. A equipe de projeto da Cii Honeywell/Bull, único competidor estrangeiro, era liderada por Jean Ichbiah.

Na primavera de 1979, Jack Cooper, do Comando Material da Marinha, recomendou o nome para a nova linguagem, Ada, o qual foi então adotado. O nome homenageia Augusta Ada Byron (1815-1851), condessa de Lovelace, matemática e filha do poeta Lord Byron. Ela é comumente reconhecida como a primeira programadora do mundo. Augusta trabalhou com Charles Babbage em seus computadores mecânicos, as Máquinas Diferenciais e Analíticas, escrevendo programas para diversos processos numéricos.

O projeto e as razões fundamentais para a linguagem Ada foram publicados pela ACM na SIGPLAN Notices (ACM, 1979) e distribuídos para mais de 10 mil pessoas. Um teste público e uma conferência de avaliação foram conduzidos em outubro de 1979 em Boston, com representantes de mais de cem organizações dos Estados Unidos e da Europa. Em novembro, mais de 500 relatórios sobre a linguagem, de 15 países, haviam sido recebidos. A maioria sugeria pequenas modificações em vez de mudanças drásticas e rejeições completas. Baseada nos relatórios sobre a linguagem, a próxima versão da especificação de requisitos, o documento Stoneman (Homem de Pedra) (Departament of Defense, 1980a), foi lançado em fevereiro de 1980.

Uma versão revisada do projeto da linguagem foi finalizada em julho de 1980 e aceita com o nome MIL-STD 1815, o *Manual de Referência da Linguagem Ada* padrão. O número 1815 foi escolhido por ser o ano de nascimento de Augusta Ada Byron. Outra versão revisada do *Manual de Referência da Linguagem Ada* foi lançada em julho de 1982. Em 1983, o Instituto Nacional de Padrões dos Estados Unidos padronizou Ada. Essa versão oficial "final" é descrita em Goos e Hartmanis (1983). O projeto da linguagem Ada foi então congelado por cinco anos.

2.14.3 Visão geral da linguagem

Esta seção descreve brevemente quatro das principais contribuições da linguagem Ada.

Pacotes na linguagem Ada fornecem os meios para encapsular objetos de dados, especificações para tipos de dados e procedimentos. Isso, por sua vez, fornece o suporte para o uso de abstração de dados no projeto de programas, conforme descrito no Capítulo 11.

A linguagem Ada inclui diversos recursos para o tratamento de exceções, os quais permitem que os programadores ganhem o controle após ter sido detectada a ocorrência de uma exceção, ou erros em tempo de execução, dentro de uma variedade de exceções possíveis. O tratamento de exceções é discutido no Capítulo 14.

As unidades de programas podem ser genéricas em Ada. Por exemplo, é possível escrever um procedimento de ordenação que usa um tipo não especificado para os dados a serem ordenados. Tal procedimento genérico deve ser

instanciado para um tipo específico antes de poder ser usado, o que é feito com uma sentença que faz o compilador gerar uma versão do procedimento com o tipo informado. A disponibilidade de tais unidades genéricas aumenta a faixa de unidades de programas que podem ser reutilizadas, em vez de duplicadas, pelos programadores. Tipos genéricos são discutidos nos Capítulos 9 e 11.

A linguagem Ada também fornece a execução concorrente de unidades de programa especiais, chamadas tarefas, usando o mecanismo *rendezvous*. *Rendezvous* é o nome de um método de sincronização e comunicação intertarefas. Concorrência é o assunto discutido no Capítulo 13.

2.14.4 Avaliação

Talvez os aspectos mais importantes do projeto da linguagem Ada a serem considerados são:

- Como o projeto era competitivo, não existiam limites na participação.
- A linguagem Ada agrupa a maioria dos conceitos de engenharia de software e projeto de linguagem do final dos anos 1970. Apesar de alguém poder questionar as abordagens concretas usadas para incorporar esses recursos, bem como a inclusão de um número muito grande deles em uma linguagem, a maioria das pessoas concorda que os recursos são valiosos.
- Apesar de a maioria das pessoas não ter esperado isso, o desenvolvimento de um compilador para a linguagem Ada era uma tarefa difícil. Apenas em 1985, quase quatro anos após o projeto da linguagem estar completo, é que compiladores Ada realmente usáveis começaram a aparecer.

O criticismo mais sério em relação à Ada em seus primeiro anos foi que a linguagem era muito grande e complexa. Em particular, Hoare (1981) afirmou que ela não deveria ser usada por quaisquer aplicações nas quais a confiabilidade fosse crítica, o que é precisamente o tipo de aplicações para o qual ela foi projetada. Por outro lado, outros a consideram o epítome do projeto de linguagens de sua época. De fato, até mesmo Hoare no fim das contas suavizou sua visão da linguagem.

A seguir, temos um exemplo de um programa escrito em Ada:

```
-- Programa de exemplo em Ada
-- Entrada:  Um inteiro, listlen, onde listlen é menor do que
--           100, seguido por valores inteiros listlen
-- Saída:    O número de valores de entrada que são maiores
--           do que a média de todos os valores de entrada
with Ada.Text_IO, Ada.Integer.Text_IO;
use Ada.Text_IO, Ada.Integer.Text_IO;
procedure Ada_Ex is
  type Int_List_Type is array (1..99) of Integer;
  Int_List : Int_List_Type;
  List_Len, Sum, Average, Result : Integer;
begin
```

```
  Result:= 0;
  Sum := 0;
  Get (List_Len);
  if (List_Len > 0) and (List_Len < 100) then
-- Read input data into an array and compute the sum
    for Counter := 1 .. List_Len loop
      Get (Int_List(Counter));
      Sum := Sum + Int_List(Counter);
    end loop;
-- Calcula a média
    Average := Sum / List_Len;
-- Count the number of values that are > average
    for Counter := 1 .. List_Len loop
      if Int_List(Counter) > Average then
        Result:= Result+ 1;
      end if;
    end loop;
-- Imprimir o resultado
    Put ("The number of values > average is:");
    Put (Result);
    New_Line;
else
    Put_Line ("Error—input list length is not legal");
  end if;
end Ada_Ex;
```

2.14.5 Ada 95

Dois dos mais importantes novos recursos de Ada 95 são brevemente descritos nos parágrafos seguintes. No restante deste livro, iremos usar o nome Ada 83 para a versão original e Ada 95 (seu nome atual) para a versão posterior quando for importante distinguir entre as duas. Em discussões sobre recursos comuns em ambas, usaremos o nome Ada. O padrão da linguagem Ada 95 é definido em ARM (1995).

O mecanismo de derivação de tipos de Ada 83 é estendido em Ada 95 para permitir adições de novos componentes àqueles herdados de uma classe base. Isso fornece herança, um ingrediente chave em linguagens de programação orientadas a objetos. A vinculação dinâmica de chamadas a definições de subprogramas é realizada por meio de despacho de subprogramas, o qual é baseado no valor de marcação de tipos derivados por tipos com a amplitude de classes (*classwide types*). Esse recurso possibilita o uso de polimorfismo, outro recurso principal da programação orientada a objetos. Tais recursos de Ada 95 são discutidos no Capítulo 12.

O mecanismo de *rendezvous* de Ada 83 fornecia apenas um meio pesado e ineficiente de compartilhar dados entre processos concorrentes. Era necessário introduzir uma tarefa para controlar o acesso aos dados compartilhados. Os objetos protegidos de Ada 95 oferecem uma alternativa atraente. Os dados compartilhados são encapsulados em uma estrutura sintática que controla

todos os acessos aos dados por *rendezvous* ou por chamadas a subprogramas. Os novos recursos de Ada 95 para concorrência e dados compartilhados são discutidos no Capítulo 13.

Acredita-se que a popularidade de Ada 95 diminuiu porque o Departamento de Defesa parou de obrigar seu uso em sistemas de software militares. Existem, é claro, outros fatores que impediram seu crescimento em popularidade. O mais importante foi a ampla aceitação de C++ para programação orientada a objetos, que ocorreu antes do lançamento de Ada 95.

2.15 PROGRAMAÇÃO ORIENTADA A OBJETOS: SMALLTALK

Smalltalk foi a primeira linguagem de programação que ofereceu suporte completo à programação orientada a objetos. Logo, é uma parte importante de qualquer discussão sobre a evolução das linguagens de programação.

2.15.1 Processo de projeto

Os conceitos que levaram ao desenvolvimento de Smalltalk se originaram na tese de doutorado de Alan Kay no final dos anos 1960, na Universidade de Utah (Kay, 1969). Kay teve a excepcional visão de prever a disponibilidade futura de computadores de mesa poderosos. Lembre-se de que os primeiros sistemas de microcomputadores não foram comercializados até o meio dos anos 1970, e eles estavam apenas remotamente relacionados às máquinas vislumbradas por Kay, as quais deveriam executar 1 milhão ou mais instruções por segundo e conter diversos megabytes de memória. Tais máquinas, na forma de estações de trabalho, tornaram-se disponíveis apenas no início dos anos 1980.

Kay acreditava que os computadores de mesa seriam usados por não programadores e, dessa forma, precisariam de capacidades muito poderosas de interação humano-computador. Os computadores do final dos anos 1960 eram orientados a lote e usados exclusivamente por programadores profissionais e cientistas. Para o uso de não programadores, Kay determinou que um computador deveria ser altamente interativo e usar interfaces gráficas sofisticadas com os usuários. Alguns dos conceitos gráficos vieram da experiência do LOGO de Seymour Paper, nos quais gráficos eram usados para ajudar crianças no uso de computadores (Papert, 1980).

Kay originalmente vislumbrou um sistema que chamou de Dynabook, pensado como um processador de informações gerais. Ele era baseado em parte na linguagem Flex, a qual Kay havia ajudado a projetar. Flex era baseada primariamente no SIMULA 67. O Dynabook usava o paradigma em uma mesa de trabalho típica, na qual existem diversos papéis, alguns parcialmente cobertos. A folha de cima é o foco da atenção, enquanto as outras estão temporariamente fora de foco. A visualização do Dynabook modelaria essa cena, usando janelas de tela para representar diversas folhas de papel na área de trabalho. O usuário interagiria com tal visualização tanto por meio de um

teclado quanto tocando a tela com seus dedos. Após o projeto preliminar do Dynabook lhe dar um doutorado, o objetivo de Kay se tornou ver tal máquina construída.

Kay conseguiu entrar no Xerox Palo Alto Research Center (Xerox PARC – Centro de Pesquisa da Xerox em Palo Alto) e apresentar suas ideias sobre o Dynabook. Isso o levou a ser empregado lá e, subsequentemente, ajudar no nascimento do Learning Research Group (Grupo de Pesquisa em Aprendizagem) da Xerox. A primeira atribuição do grupo era projetar uma linguagem para suportar o paradigma de programação de Kay e implementá-lo no melhor computador pessoal possível. Esses esforços resultaram em um Dynabook "Interino", composto do hardware Xerox Alto e do software Smalltalk-72. Juntos, eles formaram uma ferramenta de pesquisa para desenvolvimento futuro. Numerosos projetos de pesquisa foram conduzidos com esse sistema, incluindo experimentos para ensinar programação a crianças. Com os experimentos, vieram desenvolvimentos adicionais, levando a uma sequência de linguagens que culminaram com o Smalltalk-80. A linguagem cresceu, assim como o poder do hardware no qual ela residia. Em 1980, tanto a linguagem quanto o hardware da Xerox praticamente casavam com a visão original de Alan Kay.

2.15.2 Visão geral da linguagem

O mundo de Smalltalk é populado por nada além de objetos, de constantes inteiros até grandes sistemas de software complexos. Toda a computação em Smalltalk é feita pela mesma técnica uniforme: enviando uma mensagem a um objeto para invocar um de seus métodos. Uma resposta a uma mensagem é um objeto, o qual retorna a informação requisitada ou simplesmente notifica o chamador que o processamento solicitado foi completado. A diferença fundamental entre uma mensagem e uma chamada a subprograma é que uma mensagem é enviada para um objeto de dados, especificamente para um dos métodos definidos para o objeto. O método chamado é então executado, geralmente modificando os dados do objeto para o qual a mensagem foi enviada; uma chamada a subprograma é uma mensagem ao código de um subprograma. Normalmente, os dados a serem processados pelo subprograma são enviados a ele como um parâmetro[13].

Em Smalltalk, abstrações de objetos são classes, bastante similares às do SIMULA 67. Instâncias da classe podem ser criadas e são então os objetos do programa.

A sintaxe do Smalltalk não é parecida com qualquer outra linguagem de programação, em grande parte por causa do uso de mensagens, em vez de expressões lógicas e aritméticas e sentenças de controle convencionais. Uma das construções de controle do Smalltalk é ilustrada no exemplo da próxima subseção.

[13] Obviamente, uma chamada de método também pode passar dados a serem processados pelo método chamado.

2.15.3 Avaliação

Smalltalk teve um grande papel na promoção de dois aspectos separados da computação: interfaces gráficas com o usuário e programação orientada a objetos. O sistema de janelas que é agora o método dominante de interfaces com o usuário em sistemas de software cresceu a partir do Smalltalk. Atualmente, as metodologias de projeto de software e as linguagens de programação mais significativas são orientadas a objetos. Apesar de a origem de algumas das ideias de linguagens orientadas a objetos serem do SIMULA 67, elas alcançaram a maturidade apenas em Smalltalk. É claro que o impacto de Smalltalk no mundo da computação é extenso e terá vida longa.

A seguir, temos um exemplo de uma definição de classe em Smalltalk:

```
"Programa de Exemplo em Smalltalk"
"A seguir, está uma definição de classe, instanciações que
podem desenhar polígonos equiláteros de qualquer número de
lados"
class name                    Polygon
superclass                    Object
instance variable names       ourPen
numSides
sideLength
"Métodos de Classe"
"Cria uma instância"
new
   ^ super new getPen

"Obtém uma caneta para desenhar o polígono"
getPen
ourPen <- Pen new defaultNib: 2
"Métodos de instância"
"Desenha um polígono"
draw
   numSides timesRepeat: [ourPen go: sideLength;
                          turn: 360 // numSides]

 "Configura o tamanho dos lados"
 length: len
     sideLength <- len

 "Configura o número de lados"
 sides: num
     numSides <- num
```

2.16 COMBINANDO RECURSOS IMPERATIVOS E ORIENTADOS A OBJETOS: C++

As origens da linguagem C foram discutidas na Seção 2.1, as do Simula 67 foram discutidas na Seção 2.10 e as do Smalltalk foram discutidas na Seção 2.15. C++ tem diversos recursos de linguagem, emprestados do Simula 67,

sobre a linguagem C para oferecer suporte aos recursos em que o Smalltalk foi pioneiro. C++ evoluiu a partir do C com uma série de modificações para melhorar seus recursos imperativos e adicionar construções para suporte à programação orientada a objetos.

2.16.1 Processo de projeto

O primeiro passo de C em direção a C++ foi dado por Bjarne Stroustrup, no *Bell Labs*, em 1980. As modificações no C incluíam a de verificação de tipos e conversão de parâmetros de funções e classes, as quais estavam relacionadas àquelas de SIMULA 67 e Smalltalk. Também estavam incluídas classes derivadas, controle de acesso público/privado de componentes herdados, métodos construtores e destrutores e classes amigas (*friend classes*). Durante 1981, foram adicionadas funções internalizadas (*inline functions*), parâmetros padrão e a sobrecarga do operador de atribuição. A linguagem resultante foi chamada de C com Classes e é descrita em Stroustrup (1983).

É útil considerar alguns dos objetivos do C com Classes. O primário era fornecer uma linguagem na qual os programas pudessem ser organizados da mesma forma que no SIMULA 67 – ou seja, com classes e herança. Um segundo objetivo importante era que deveriam existir penalidades pequenas ou nenhuma em termos de desempenho ao C. Por exemplo, a verificação de faixa de índices de vetores não foi considerada por causa de uma desvantagem significativa de desempenho, em relação ao C, que poderia resultar disso. Um terceiro objetivo do C com Classes era que ele poderia ser usado para quaisquer aplicações para as quais C poderia, então praticamente nenhum dos recursos de C seriam removidos, nem aqueles considerados inseguros.

Em 1984, a linguagem foi estendida com inclusão de métodos virtuais, os quais fornecem vinculação dinâmica de chamadas de métodos a definições de métodos específicos, nomes de métodos e sobrecarga de operadores e tipos de referência. Essa versão da linguagem foi chamada de C++, e ela é descrita em Stroustrup (1984).

Em 1985, a primeira implementação disponível apareceu: um sistema chamado Cfront, o qual traduzia programas C++ em C. Essa versão do Cfront e a de C++ que a ferramenta implementava foram chamadas de *Release 1.0*. Essa versão é descrita em Stroustrup (1986).

Entre 1985 e 1989, C++ continuou a evoluir, baseada nas reações dos usuários sobre a primeira implementação distribuída. A próxima versão foi chamada *Release 2.0*. Sua implementação Cfront foi lançada em junho de 1989. Os recursos mais importantes adicionados ao C++ Release 2.0 foram o suporte à herança múltipla (classes com mais de uma classe pai) e classes abstratas, com algumas outras melhorias. Classes abstratas são descritas no Capítulo 12.

O *Release 3.0* de C++ evoluiu entre 1989 e 1990. Ele adicionou *templates*, os quais fornecem tipos parametrizados, e tratamento de exceções. A versão atual de C++, padronizada em 1998, é descrita pela ISO (1998).

Em 2002, a Microsoft lançou sua plataforma de computação .NET, a qual incluía uma nova versão de C++, chamada de Managed C++ (ou C++ Gerenciado), ou MC++. MC++ estende C++ para fornecer acesso às funcionalidades do framework .NET. As adições incluem propriedades, *delegates*, interfaces e um tipo de referência para objetos coletados por um coletor de lixo. Propriedades são discutidas no Capítulo 11. *Delegates* são brevemente discutidos na introdução ao C# na Seção 2.19. Como o .NET não suporta herança múltipla, o MC++ também não o faz.

2.16.2 Visão geral da linguagem

C++ fornece duas construções que definem tipos: classes e estruturas, com poucas diferenças entre as duas. Na prática, estruturas que incluem definições de métodos são geralmente usadas. Herança múltipla é suportada. Em C++, os métodos são geralmente chamados de funções membro.

Como C++ tem tanto funções quanto métodos, ele suporta a programação procedural e a orientada a objetos.

Operadores em C++ podem ser sobrecarregados – ou seja, o usuário pode criar novos operadores para os já existentes em tipos definidos pelo usuário. Métodos em C++ também podem ser sobrecarregados, e isso significa que o usuário pode definir mais de um método com o mesmo nome, desde que o número ou tipos dos parâmetros seja diferente.

A vinculação dinâmica em C++ é fornecida por métodos virtuais. Esses métodos definem operações dependentes do tipo, usando métodos sobrecarregados, dentro de uma coleção de classes que são relacionadas por herança. Um ponteiro para um objeto de uma classe A pode também apontar para objetos de classes que têm a classe A como um ancestral. Quando esse ponteiro aponta para um método virtual sobrecarregado, o do tipo atual é escolhido dinamicamente.

Tanto métodos quanto classes podem ser usados como *templates*, e isso significa que eles podem ser parametrizados. Por exemplo, um método pode ser escrito como um método com *template* de forma a permitir que ele tenha versões para uma variedade de tipos de parâmetros. As classes desfrutam da mesma flexibilidade.

C++ inclui tratamento de exceções, significativamente diferente daquele de Ada. Uma diferença é que exceções detectáveis por hardware não podem ser tratadas. As construções de tratamento de exceção de Ada e C++ são discutidas no Capítulo 14.

2.16.3 Avaliação

C++ rapidamente se tornou (e se mantém) uma linguagem muito popular. Um fator para sua popularidade é a disponibilidade de compiladores bons e baratos. Outro é que C++ é quase completamente compatível com C (os programas em C podem, com poucas mudanças, ser compilados como programas

C++) e, na maioria das implementações, é possível ligar código C++ com código em C. Por último, no momento em que C++ apareceu pela primeira vez, quando a programação orientada a objetos começou a receber mais atenção, C++ era a única linguagem disponível adequada para grandes projetos comerciais de software.

No lado negativo, como C++ é uma linguagem muito grande e complexa, ela sofre deficiências similares àquelas da linguagem PL/I. C++ herdou muitas das inseguranças de C, tornando-a menos segura do que linguagens como Ada e Java. Os recursos de orientação a objetos de C++ são descritos em detalhes no Capítulo 12.

2.16.4 Uma linguagem relacionada: Eiffel

Eiffel é outra linguagem híbrida, que contém tanto recursos imperativos quanto orientados a objetos (Meyer, 1992). Eiffel foi projetada por uma pessoa, Bertrand Meyer, francês, que vive na Califórnia, nos EUA. A linguagem inclui recursos para oferecer suporte a tipos abstratos de dados, herança e vinculação dinâmica, de forma que oferece suporte completo à programação orientada a objetos. Talvez o recurso que mais distingue Eiffel é o uso integrado de asserções para garantir o "contrato" entre subprogramas e seus chamadores. É uma ideia que nasceu com Plankalkül, mas que foi ignorada pela maioria das linguagens projetadas desde então. É natural comparar Eiffel com C++. Eiffel é menor, mais simples e mais segura, mas tem quase a mesma expressividade e facilidade de escrita. As razões para a rápida popularidade de C++, enquanto Eiffel tem um uso muito mais limitado, não são difíceis de determinar. C++ era a maneira mais fácil para organizações de desenvolvimento de software se moverem para a programação orientada a objetos, porque, em muitos casos, seus desenvolvedores já conheciam C. Eiffel não desfrutou de tal caminho fácil em termos de adoção. Além disso, para os primeiros anos nos quais C++ se espalhou, o sistema Cfront estava disponível e era barato. C++ tinha o aval do prestigioso *Bell Labs*, enquanto Eiffel era mantida por Bertran Meyer e sua companhia de software relativamente pequena, a Interactive Software Engineering.

2.16.5 Outra linguagem relacionada: Delphi

Delphi (Lischner, 2000) é uma linguagem híbrida, similar a C++ porque foi criada por meio da adição de suporte a orientação a objetos, dentre outras coisas, a uma linguagem imperativa existente, Pascal. Muitas das diferenças entre C++ e Delphi são resultado das linguagens predecessoras e das culturas de programação que as envolviam e das quais elas são derivadas. Como C é uma linguagem poderosa, mas potencialmente insegura, C++ também casa com essa descrição, ao menos nas áreas de verificação de faixas de índices de vetores, aritmética de ponteiros e seus numerosos tipos de coerção. Da mesma forma, como Pascal é mais elegante e mais seguro do que C, Delphi é

mais elegante e seguro do que C++. Delphi também é uma linguagem menos complexa. Por exemplo, ela não permite sobrecarga de operadores definida pelo usuário, subprogramas genéricos e classes parametrizadas, os quais são parte de C++. Delphi, como Visual C++, fornece uma interface gráfica com o usuário (GUI) para o desenvolvedor e maneiras simples para criar interfaces GUI para aplicações escritas em Delphi. A linguagem Delphi foi projetada por Anders Hejlsberg, que havia desenvolvido o sistema Turbo Pascal. Ambos foram comercializados e distribuídos pela Borland. Hajlsbert foi também o projetista líder de C#.

2.17 UMA LINGUAGEM ORIENTADA A OBJETOS BASEADA NO PARADIGMA IMPERATIVO: JAVA

Os projetistas de Java começaram com C++, removeram algumas construções, modificaram outras e adicionaram poucas mais. A linguagem resultante fornece muito do poder e flexibilidade de C++, mas em uma linguagem menor, mais simples e mais segura.

2.17.1 Processo de projeto

Java, como muitas linguagens de programação, foi projetada para uma aplicação que parecia não ter uma linguagem existente satisfatória. Em 1990, a Sun Microsystems determinou que existia a necessidade de uma linguagem de programação para dispositivos eletrônicos embarcados para o consumidor, como torradeiras, fornos de micro-ondas e sistemas interativos de TV. Confiabilidade era um dos objetivos primários para tal linguagem. Pode não parecer que a confiabilidade seria um fator importante no software para um forno de micro-ondas. Se um forno tem um problema de software, provavelmente não será um grave risco para ninguém e provavelmente não levará a grandes casos na justiça. Entretanto, se o sistema de software de um modelo em particular contiver erros que forem descobertos após 1 milhão de unidades terem sido fabricadas e vendidas, um *recall* teria custos significativos. Logo, a confiabilidade *é* uma característica importante do software em produtos eletrônicos para o consumidor.

Após considerar C e C++, foi decidido que nenhuma das duas linguagens seria satisfatória para desenvolver software para dispositivos eletrônicos para o consumidor. Apesar de C ser relativamente pequeno, ele não fornece suporte para programação orientada a objetos, o que pensavam ser uma necessidade. C++ suportava programação orientada a objetos, mas a equipe da Sun o julgava muito grande e complexo, em parte porque também suportava programação procedimental. Também se acreditava que nem C nem C++ forneciam o nível necessário de confiabilidade. Então, uma nova linguagem, posteriormente chamada Java, foi projetada. Seu pro-

jeto foi guiado pelo objetivo fundamental de fornecer simplicidade e confiabilidade maiores do que acreditavam serem fornecidas por C++.

Apesar de o ímpeto inicial de Java serem os eletrônicos para consumidores, nenhum dos produtos nos quais ela foi usada em seus primeiros anos foram comercializados. Quando a World Wide Web se tornou bastante usada, iniciando em 1993, por causa dos novos navegadores gráficos, descobriu-se que Java era uma ferramenta útil de programação para a Web. Em particular, os *applets* Java, programas relativamente pequenos cuja saída pode ser incluída e mostrada em documentos Web, rapidamente se tornaram populares do meio para o fim da década de 1990. Em seus primeiros anos de uso público, a Web era a aplicação de Java mais comum.

A equipe de projeto de Java era liderada por James Gosling, que havia projetado o editor emacs do UNIX e o sistema de janelas NeWS.

2.17.2 Visão geral da linguagem

Conforme mencionado, Java é baseada em C++, mas foi projetada para ser menor, mais simples e mais confiável. Como C++, Java tem tanto classes quanto tipos primitivos. Vetores em Java são instâncias de uma classe pré-definida, enquanto em C++ eles não são – apesar de muitos usuários C++ construírem classes que encapsulam vetores, de forma a adicionar recursos como verificação de índice de faixa, que é implícito em Java.

Java não tem ponteiros, mas seus tipos de referência fornecem algumas das capacidades de ponteiros. Essas referências são usadas para apontar a instâncias de classes. Todos os objetos são alocados no monte. Embora ponteiros e referências possam parecer bastante semelhantes, existem algumas diferenças semânticas importantes. Ponteiros apontam para locais de memória, referências apontam para objetos. Isso torna a aritmética de referências sem sentido, eliminando tal prática passível de erros. A distinção entre um valor de ponteiro e o valor para o qual ele aponta é responsabilidade do programa em muitas linguagens, nas quais os ponteiros precisam ser explicitamente desreferenciados. As referências são sempre implicitamente desreferenciadas, quando necessário. Dessa forma, elas se comportam mais como variáveis escalares ordinárias.

Java tem um tipo booleano chamado **`boolean`**, usado principalmente para as expressões de controle de suas sentenças de controle (como **`if`** e **`while`**). Diferentemente de C e C++, expressões aritméticas não podem ser usadas para expressões de controle.

Uma diferença significativa entre Java e muitas de suas antecessoras que suportam orientação a objetos, incluindo C++, é não ser possível escrever subprogramas autocontidos em Java. Todos os subprogramas em Java são métodos definidos em classes. Além disso, os métodos podem ser chamados apenas por meio de uma classe ou objeto. Uma consequência disso é que enquanto C++ oferece suporte tanto para programação orientada a

objetos quanto procedural, Java oferece suporte apenas para programação orientada a objetos.

Outra diferença importante entre C++ e Java é que o primeiro oferece suporte à herança múltipla diretamente em suas definições de classes. Algumas pessoas acreditam que a herança múltipla leva a mais complexidade e confusão do que traz benefícios. Java oferece suporte apenas para herança simples de classes, apesar de alguns dos benefícios de herança múltipla poderem ser obtidos pelo uso de interfaces.

Dentre as construções C++ que não foram copiadas por Java estão as estruturas e as uniões.

Java inclui uma forma relativamente simples de controle de concorrência por meio de seu modificador **`synchronized`**, que pode aparecer em métodos e blocos. Em ambos, ele faz com que um bloqueio seja anexado. O bloqueio garante acesso ou execução mutuamente exclusivo. Em Java, é relativamente fácil criar processos concorrentes, chamados de linhas de execução (*threads*).

Java usa liberação implícita de armazenamento para seus objetos, geralmente chamada de **coleta de lixo**. Isso libera o programador da necessidade de remover explicitamente os objetos quando eles não forem mais necessários. Programas escritos em linguagens que não têm coleta de lixo sofrem de um problema chamado de vazamento de memória, ou seja, o armazenamento é alocado, mas nunca liberado. Isso pode levar ao consumo de todo o armazenamento disponível. A liberação de objetos é discutida em detalhes no Capítulo 6.

Diferentemente de C e C++, Java inclui coerções de tipo em atribuições (conversões de tipo implícitas) apenas se elas aumentarem o tipo (de um "menor" para um "maior"). Logo, coerções de **`int`** para **`float`** são feitas por meio do operador de atribuição, mas coerções de **`float`** para **`int`** não.

2.17.3 Avaliação

Os projetistas de Java fizeram bem em remover o excesso e/ou recursos inseguros de C++. Por exemplo, a eliminação de metade das coerções de atribuição feitas em C++ é um passo a frente em direção a maior confiabilidade. A verificação de faixas de índices de acessos a vetores também torna a linguagem mais segura. A adição de concorrência melhora o escopo de aplicações que podem ser escritas na linguagem, assim como as bibliotecas de classes para interfaces gráficas com o usuário, acesso a bases de dados e redes.

A portabilidade de Java, ao menos em sua forma intermediária, é geralmente atribuída ao projeto da linguagem, mas na verdade essa atribuição não é correta. Qualquer linguagem poderia ser traduzida para uma forma intermediária e "executada" em qualquer plataforma que tivesse uma máquina virtual para essa forma intermediária. O preço desse tipo de portabilidade é o custo de interpretação, que tradicionalmente tem sido uma ordem de magnitude maior do que a execução de código de máquina. A versão inicial do interpre-

tador Java, chamado de Máquina Virtual Java (JVM), era ao menos 10 vezes mais lento do que os programas compilados em C equivalentes. Entretanto, muitos programas Java são agora traduzidos para código de máquina antes de serem executados, usando compiladores Just-in-Time (JIT). Isso torna a eficiência de programas Java competitiva com a dos programas escritos em linguagens convencionalmente compiladas, como C++.

O uso de Java aumentou mais rapidamente do que o de qualquer outra linguagem de programação. Inicialmente, isso ocorreu por causa de seu valor na programação de documentos Web dinâmicos. Outro fator é o sistema de compilação/interpretação para Java ser gratuito e fácil de obter na Web. É claro que uma das razões para a rápida ascensão de Java é que os programadores gostam de seu projeto. Sempre existiram alguns desenvolvedores que pensavam que a linguagem C++ era muito grande e complexa para ser prática e segura. Java ofereceu a eles uma alternativa com muito do poder de C++, mas em uma linguagem mais simples, segura. Java é agora usada em uma variedade de áreas de aplicação.

A versão mais recente, que apareceu em 2004 e foi chamada de Java 1.5, mas depois renomeada para Java 5.0, inclui algumas adições significativas. Tais adições incluem uma classe de enumerações, tipos genéricos e uma nova construção de iteração.

A seguir, temos um exemplo de um programa em Java:

```
/ Programa de exemplo em Java
//  Entrada: Um inteiro, listlen, onde listlen é menor do que
//           100, seguido por valores inteiros listlen
// Saída:    O número de valores de entrada que são maiores
//           do que a média de todos os valores de entrada
import java.io.*;
class IntSort {
public static void main(String args[]) throws IOException {
  DataInputStream in = new DataInputStream(System.in);
  int listlen,
      counter,
      sum = 0,
      average,
      result = 0;
  int[] intlist = new int[99];
  listlen = Integer.parseInt(in.readLine());
  if ((listlen > 0) && (listlen < 100)) {
/* Lê os dados de entrada em um vetor e calcula sua soma  */
    for (counter = 0; counter < listlen; counter++) {
      intlist[counter] =
             Integer.valueOf(in.readLine()).intValue();
      sum += intlist[counter];
    }
/* Calcula a média */
    average = sum / listlen;
```

```
/* Conta o número de valores que são maiores do que a média */
    for (counter = 0; counter < listlen; counter++)
      if (intlist[counter] > average) result++;
/* Imprimir o resultado */
     System.out.println(
          "\nNumber of values > average is:" + result);
    }  //** end of then clause of if ((listlen > 0) ...
    else System.out.println(
              "Error—input list length is not legal\n");
  }  //** end of method main
}  //** end of class IntSort
```

2.18 LINGUAGENS DE *SCRIPTING*

As linguagens de *scripting* evoluíram nos últimos 25 anos. As primeiras eram usadas por meio de uma lista de comandos, chamada de ***script***, em um arquivo a ser interpretado. A primeira dessas linguagens, chamada sh (de *shell*), começou como uma pequena coleção de comandos interpretados como chamadas a subprogramas de sistema que realizavam funções utilitárias, como gerenciamento de arquivos e filtragens de arquivos simples. Nessa base, foram adicionadas variáveis, sentenças de controle de fluxo, funções e várias capacidades, e o resultado é uma linguagem de programação completa. Uma das mais poderosas e usadas dessas linguagens é a ksh (Bolsky e Korn, 1995), desenvolvida por David Korn no Bell Labs.

Outra linguagem de *scripting* é a awk, desenvolvida por Al Aho, Brian Kernighan e Peter Weinberger no Bell Labs (Aho et al., 1988). A awk começou como uma linguagem de geração de relatórios, mas se tornou uma linguagem de propósito mais geral.

2.18.1 Origens e características de Perl

A linguagem Perl, desenvolvida por Larry Wall, era originalmente uma combinação de sh e awk. Perl cresceu significativamente desde seu início e é agora uma linguagem de programação poderosa. Apesar de ser geralmente chamada de linguagem de *scripting*, ela é mais similar a uma linguagem imperativa típica, já que é sempre compilada, ao menos para uma linguagem intermediária, antes de ser executada. Além disso, ela tem todas as construções que a tornam aplicável a uma variedade de áreas de problemas computacionais.

Perl tem diversos recursos interessantes, dos quais apenas alguns são mencionados neste capítulo e discutidos no restante do livro.

Variáveis em Perl são estaticamente tipadas e implicitamente declaradas. Existem três espaços de nomes distintos para variáveis, denotados pelo primeiro caractere de nomes de variáveis. Todos os nomes de variáveis escalares

começam com cifrão ($), todos os nomes de vetores começam com um arroba (@) e todos os nomes de dispersões (*hashes*) – dispersões são brevemente descritas abaixo – começam com sinais de percentual (%). Essa convenção torna os nomes em programas mais legíveis do que aqueles em qualquer outra linguagem de programação.

Perl inclui um grande número de variáveis implícitas. Algumas são usadas para armazenar parâmetros Perl, como o formato particular do caractere de nova linha ou caracteres que são usados na implementação. Variáveis implícitas são usadas como parâmetros padrão para funções pré-definidas e operandos padrão para alguns operadores. As variáveis implícitas têm nomes distintos – apesar de crípticos, como `$!` e `@_`. Os nomes de variáveis implícitas, como os de variáveis definidas pelo usuário, usam os três espaços de nomes, logo $! é um escalar.

Os vetores em Perl têm duas características que os separam de outros vetores das linguagens imperativas comuns. Primeiro, eles têm tamanho dinâmico, ou seja, podem crescer e encolher conforme necessário durante a execução. Segundo, vetores podem ser esparsos, ou seja, pode haver espaços em branco entre os elementos. Esses espaços em branco não ocupam espaço em memória, e a sentença de iteração usada para vetores, **`foreach`**, itera sobre os elementos que faltam.

Perl inclui vetores associativos, chamados de *dispersões*. Essas estruturas de dados são indexadas por cadeias e são tabelas de dispersão (*hash tables*) implicitamente controladas. O sistema Perl fornece a função *hash* e aumenta o tamanho da estrutura conforme necessário.

Perl é uma linguagem poderosa, mas de certa forma perigosa. Seus tipos escalares armazenam tanto cadeias quanto números, normalmente armazenados em formato de ponto flutuante em precisão dupla. Dependendo do contexto, os números podem sofrer coerção para cadeias e vice-versa. Se uma string é usada em um contexto numérico e ela não puder ser convertida para um número, é usado zero e não existe mensagem de aviso ou erro para o usuário. Esse efeito pode levar a erros que não são detectados pelo compilador ou pelo sistema de tempo de execução. A indexação de vetores não pode ser verificada, porque não existem conjuntos de faixas de índices para os vetores. Referências a elementos não existentes retornam **`undef`**, interpretado como zero em contexto numérico. Então, não existe também detecção de erros no acesso a elementos de vetores.

O uso inicial de Perl era como um utilitário do UNIX para processar arquivos de texto. Perl era e ainda é muito usada como uma ferramenta de administração de sistema em UNIX. Quando a World Wide Web apareceu, Perl atingiu grade utilização como uma linguagem CGI (Common Gateway Interface) para uso na Web, apesar de agora ser raramente usada para esse propósito. Perl é usada como uma linguagem de propósito geral para uma variedade de aplicações, como biologia computacional e inteligência artificial.

A seguir, temos um exemplo de um programa em Perl:

```
# Programa de exemplo em Perl
# Entrada:  Um inteiro, listlen, onde listlen é menor do que
#           100, seguido por valores inteiros listlen
# Saída:    O número de valores de entrada que são maiores
#           do que a média de todos os valores de entrada.
($sum, $result) = (0, 0);
$listlen = <STDIN>;
if (($listlen > 0) && ($listlen < 100)) {
# Lê os dados de entrada em um vetor e calcula sua soma
  for ($counter = 0; $counter < $listlen; $counter++) {
    $intlist[$counter] = <STDIN>;
  } #- end of for (counter ...
# Calcula a média
  $average = $sum / $listlen;
# Conta o número de valores que são maiores do que a média
 foreach $num (@intlist) {
    if ($num > $average) { $result++; }
  } #- end of foreach $num ...
# Imprimir o resultado
  print "Number of vlues > average is: $result \n";
} #- end of if (($listlen ...
else {
  print "Error--input list length is not legal \n";
}
```

2.18.2 Origens e características de JavaScript

O uso da Web explodiu no meio dos anos 1990, após a aparição dos primeiros navegadores gráficos. A necessidade de computação associada com documentos HTML, os quais por si só eram completamente estáticos, rapidamente se tornou crítica. A computação no lado servidor era possível com o uso de CGI (Common Gateway Interface), que permitia aos documentos HTML requisitarem a execução de programas no servidor. Os resultados de tais computações retornavam ao navegador na forma de documentos HTML. A computação no navegador se tornou disponível com o advento dos *applets* Java. Ambas abordagens foram agora substituídas em grande parte por novas tecnologias, primariamente linguagens de *scripting*.

JavaScript (Flanagan, 2002), originalmente chamado LiveScript, foi desenvolvida na Netscape. No final de 1995, LiveScript se tornou um projeto conjunto da Netscape com a Sun Microsystems e seu nome foi modificado para JavaScript. JavaScript passou por uma evolução extensa, da versão 1.0 para a versão 1.5 com a adição de muitos recursos e capacidades. Um padrão de linguagem para JavaScript foi desenvolvido no final dos anos 1990 pela Associação Europeia de Fabricantes de Computadores (ECMA – European Computer Manufacturers Association), chamado ECMA-262. Esse padrão também foi aprovado pela Organização Internacional de Padrões (ISO – International Standards Organization) como a ISO-16262. A versão da Microsoft de JavaScript é chamada de JScript .NET.

Apesar de um interpretador JavaScript poder ser embarcado em muitas aplicações, seu uso mais comum é em navegadores Web. Código JavaScript é embarcado em documentos HTML e interpretado pelo navegador quando os documentos são mostrados. Os principais usos de JavaScript na programação Web são a validação de dados de entrada de formulários e a criação de documentos HTML dinâmicos. JavaScript também é usada com o framework de desenvolvimento Web Rails.

Apesar de seu nome, JavaScript é relacionada com Java apenas pelo uso de uma sintaxe similar. Java é fortemente tipada, mas JavaScript é dinamicamente tipada (veja o Capítulo 5). As cadeias de caracteres e os vetores de JavaScript têm tamanho dinâmico. Assim, os índices de vetores não são verificados em relação a sua vaidade, apesar de isso ser obrigatório em Java. Java oferece suporte completo para programação orientada a objetos, mas JavaScript não oferece suporte para herança e para vinculação dinâmica de chamadas a métodos.

Um dos usos mais importantes de JavaScript é para a criação e modificação dinâmica de documentos HTML. JavaScript define uma hierarquia de objetos que casa com um modelo hierárquico de um documento HTML, definido pelo Document Object Model (DOM). Elementos de um documento HTML são acessados por meio desses objetos, fornecendo a base para o controle dinâmico dos elementos dos documentos.

A seguir, temos um *script* em JavaScript para o problema previamente solucionado em diversas linguagens neste capítulo. Note que assumimos que esse *script* será chamado de um documento HTML e interpretado por um navegador Web.

```
// example.js
//   Entrada: Um inteiro, listlen, onde listlen é menor do que
//            100, seguido por valores numéricos listlen
//  Saída:    O número de valores de entrada que são maiores
//            do que a média de todos os valores de entrada

var intList = new Array(99);
var listLen, counter, sum = 0, result = 0;

listLen = prompt (
        "Please type the length of the input list", "");
if ((listLen > 0) && (listLen < 100)) {

// Get the input and compute its sum
   for (counter = 0; counter < listLen; counter++) {
      intList[counter] = prompt (
                    "Please type the next number", "");
      sum += parseInt(intList[counter]);
   }

// Calcula a média
   average = sum / listLen;

// Conta o número de valores que são maiores do que a média
```

```
    for (counter = 0; counter < listLen; counter++)
      if (intList[counter] > average) result++;
// Display the results
    document.write("Number of values > average is: ",
                   result, "<br />");
} else
    document.write(
        "Error - input list length is not legal <br />");
```

2.18.3 Origens e características de PHP

PHP (Converse e Park, 2000) foi desenvolvido por Rasmus Lerdorf, membro do Grupo Apache, em 1994. Sua motivação inicial era fornecer uma ferramenta para ajudar a rastrear os visitantes em seu site pessoal. Em 1995, ele desenvolveu um pacote chamado Personal Home Page Tools (Ferramentas para Páginas Pessoais), que foi a primeira versão distribuída publicamente de PHP. Originalmente, PHP era uma abreviação para Personal Home Page (Página Pessoal). Mais tarde, sua comunidade de usuários começou a usar o nome recursivo **P**HP: **H**ypertext **P**reprocessor (PHP: Processador de Hipertexto), o que levou o nome original à obscuridade. PHP é agora desenvolvido, distribuído e suportado como um produto de código aberto. Processadores PHP estão disponíveis na maioria dos servidores Web.

PHP é uma linguagem de *scripting*, do lado servidor, embutida em HTML e, projetada para aplicações Web. O código PHP é interpretado no servidor Web quando um documento HTML no qual ele está embutido for requisitado por um navegador. O código PHP normalmente produz código HTML como saída, o que o substitui no documento HTML. Logo, um navegador Web nunca vê código PHP.

PHP é similar a JavaScript em sua aparência sintática, na natureza dinâmica de suas cadeias e vetores e no uso de tipagem dinâmica. Os vetores em PHP são uma combinação dos vetores de JavaScript e das dispersões em Perl.

A versão original de PHP não oferecia suporte à programação orientada a objetos, mas isso foi adicionado na segunda versão. Entretanto, PHP não suporta classes abstratas ou interfaces, destrutores ou controles de acesso para membros de classes.

PHP permite um acesso simples aos dados de formulários HTML, logo o processamento de formulários é fácil com PHP. PHP fornece suporte para muitos sistemas de gerenciamento de bancos de dados. Isso o torna uma excelente linguagem para construir programas que precisam de acesso Web a bases de dados.

2.18.4 Origens e características de Python

Python (Lutz e Ascher, 2004) é uma linguagem de *scripting* orientada a objetos interpretada relativamente recente. Seu projeto original foi feito por Guido van Rossum no Stichting Mathematisch Centrum, na Holanda, no iní-

cio dos anos 1990. Seu desenvolvimento é feito agora pela Python Software Foundation (Fundação de Software Python). Python é usada para os mesmos tipos de aplicação que Perl: administração de sistemas, programação em CGI, e outras tarefas computacionais relativamente pequenas. É um sistema de código aberto e está disponível para a maioria das plataformas de computação comuns. A distribuição padrão da indústria para Windows está disponível em www.activestate.com/Products/ActivePython. Implementações para outras plataformas estão disponíveis em www.python.org, o qual fornece também informações sobre Python.

A sintaxe de Python não é baseada diretamente em nenhuma linguagem comumente usada. Ela é uma linguagem com verificação de tipos, mas tipada dinamicamente. Em vez de vetores, inclui três tipos de estruturas de dados: listas; listas imutáveis, chamadas de **tuplas**; e dispersões, chamadas de **dicionários**. Existe uma coleção de métodos de lista, como inserir no final (`append`), inserir em uma posição arbitrária (`insert`), remover (`remove`) e ordenar (`sort`), assim como uma coleção de métodos para dicionários, como para obter chaves (`keys`), valores (`values`), para copiar (`copy`) e para verificar a existência de uma chave (`has_key`). Python também oferece suporte para compreensões de lista, originadas na linguagem Haskell. Compreensões de listas são discutidas na Seção 15.8.

Python é orientada a objetos, inclui as capacidades de casamento de padrões de Perl e tem tratamento de exceções. Coleta de lixo é usada para remover elementos da memória quando não são mais necessários.

O suporte para programação CGI, e para o processamento de formulários em particular, é fornecido pelo módulo `cgi`. Módulos que oferecem suporte a *cookies*, redes e acesso a bases de dados também estão disponíveis.

Um dos recursos mais interessantes de Python é que ela pode ser facilmente estendida por qualquer usuário. Os módulos que suportam as extensões podem ser escritos em qualquer linguagem compilada. Extensões podem adicionar funções, variáveis e tipos de objetos. Essas extensões são implementadas como adições ao interpretador Python.

2.18.5 Origens e características de Ruby

Ruby (Thomas et al., 2005) foi projetada por Yukihiro Matsumoto (também conhecido como Matz) no início dos anos 1990 e lançada em 1996. Desde então, tem evoluído continuamente. A motivação para Ruby era a falta de satisfação de seu projetista com Perl e Python. Apesar de tanto Perl quanto Python oferecerem suporte à programação orientada a objetos, nenhuma delas é uma linguagem puramente orientada a objetos, ao menos no sentido de que ambas têm tipos primitivos (não objetos) e possibilitam o uso de funções.

O recurso característico primário de Ruby é que se trata uma linguagem orientada a objetos pura, como Smalltalk. Cada valor de dados é um objeto e todas as operações são feitas por meio de chamadas a métodos. Os operadores em Ruby são os únicos mecanismos sintáticos para especificar chamadas a

métodos para as operações correspondentes. Como são métodos, podem ser redefinidos. Todas as classes, pré-definidas ou definidas pelos usuários, são passíveis de extensão por herança.

Tanto as classes quanto os objetos em Ruby são dinâmicos no sentido de que os métodos podem ser adicionados dinamicamente a ambos. Isso significa que tanto classes quanto objetos podem ter conjuntos de métodos em diferentes momentos durante a execução. Então, instanciações diferentes da mesma classe podem se comportar de maneira diferente. Coleções de métodos, dados e constantes podem ser incluídas na definição de uma classe.

A sintaxe de Ruby está relacionada à de Eiffel e à de Ada. Não existe necessidade de declarar variáveis, porque a tipagem dinâmica é usada. O escopo de uma variável é especificado em seu nome: uma variável cujo nome começa com uma letra tem escopo local; outra que começa com @ é uma variável de instância e aquela que começa com $ tem escopo global. Diversos recursos de Perl estão presentes em Ruby, incluindo variáveis implícitas com nomes patéticos, como $_.

Como no caso de Python, qualquer usuário pode estender ou modificar Ruby, que foi a primeira linguagem de programação projetada no Japão a atingir um uso relativamente amplo nos Estados Unidos.

2.18.6 Origens e características de Lua

Lua foi projetada no início dos anos 1990 por Roberto Ierusalimschy, Waldemar Celes e Luis Henrique de Figueiredo na Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), no Brasil. É uma linguagem de *scripting* que oferece suporte para programação procedural e funcional com extensibilidade como um de seus objetivos primários. Dentre as linguagens que influenciaram seu projeto estão Scheme, Icon e Python.

Lua é similar a JavaScript no sentido em que não oferece suporte a programação orientada a objetos, mas foi influenciada por ela. Ambas têm objetos que desempenham o papel tanto de classes quanto de objetos e herança baseada em protótipo em vez de herança de classe. Entretanto, no caso de Lua, a linguagem pode ser estendida para oferecer suporte à programação orientada a objetos.

Como em Scheme, as funções em Lua são valores de primeira classe. Além disso, a linguagem oferece suporte para *closures*. Essas capacidades permitem que ela seja usada para programação funcional. Também como Scheme, Lua tem apenas uma estrutura de dados, apesar de ser a tabela. As tabelas de Lua estendem os vetores associativos de PHP, os quais incluem os vetores presentes nas linguagens imperativas tradicionais. Referências a elementos de tabela podem tomar a forma de referências a vetores tradicionais, vetores associativos ou registros. Como as funções são valores de primeira classe, elas podem ser armazenadas em tabelas, e essas tabelas podem servir como espaços de nomes.

Lua usa coleção de lixo para seus objetos, os quais são todos alocados no monte. Ela usa tipagem dinâmica, como a maioria das outras linguagens de *scripting*.

Lua é uma linguagem pequena e relativamente simples, com apenas 21 palavras reservadas. A filosofia de projeto da linguagem é de fornecer apenas o necessário e maneiras simples de estender a linguagem para permitir que ela se encaixe em uma variedade de áreas de aplicação. Muito de sua extensibilidade é derivada de sua estrutura de dados, a tabela, a qual pode ser customizada usando o conceito de metatabelas de Lua.

Lua pode ser usada como uma extensão de uma linguagem de *scripting* para outras linguagens. Como as primeiras implementações de Java, Lua é traduzida para um código intermediário e interpretada. Ela pode ser facilmente embarcada em outros sistemas, em parte por causa do tamanho pequeno de seu interpretador, apenas cerca de 150Kbytes.

Durante 2006 e 2007, a popularidade de Lua cresceu rapidamente, em decorrência de seu uso na indústria de jogos. A sequência de linguagens de *scripting* que apareceram nos últimos 20 anos já produziu diversas linguagens bastante usadas. Lua, a última a chegar, está se tornando rapidamente uma delas.

2.19 UMA LINGUAGEM BASEADA EM C PARA O NOVO MILÊNIO: C#

C#, ao lado da nova plataforma de desenvolvimento .NET[14], foi anunciada pela Microsoft em 2000. Em janeiro de 2002, versões de produção de ambas estavam disponíveis.

2.19.1 Processo de projeto

O C# é baseado em C++ e Java, mas também inclui algumas ideias do Delphi e do Visual BASIC. Seu projetista líder, Anders Hejlsberg, também projetou o Turbo Pascal e o Delphi, o que explica as partes Delphi da herança do C#.

O propósito de C# é fornecer uma linguagem para o desenvolvimento de software baseado em componentes, especificamente para tal desenvolvimento no *framework* .NET. Nesse ambiente, componentes de uma variedade de linguagens podem ser facilmente combinados para formarem sistemas. Todas as linguagens do .NET, incluindo C#, Visual Basic .NET, C++ gerenciado, J# .NET e JScript .NET, usam o Common Type System (CTS – Sistema de Tipos Comum). O CTS fornece uma biblioteca de classes comum. Todos os tipos, nas cinco linguagens do .NET, herdam de uma classe raiz, `System.Object`. Compiladores que estão em conformidade com a especificação CTS criam objetos que podem ser combinados em sistemas de software. As cinco linguagens .NET são compiladas para o mesmo formato intermediário, Intermediate Language (IL – Linguagem Intermediária)[15]. Diferentemente de Java, entretanto, a IL nunca é interpretada. Um compi-

[14] O sistema de desenvolvimento .NET é brevemente discutido no Capítulo 1.

[15] Inicialmente, a IL era chamada de MSIL (de Microsoft Intermediate Language – Linguagem Intermediária da Microsoft), mas muitas pessoas acharam o nome extenso.

lador Just-in-Time é usado para converter IL em código de máquina antes de esse ser executado.

2.19.2 Visão geral da linguagem

Muitos acreditam que um dos avanços mais importantes de Java em relação a C++ está na exclusão de alguns recursos. Por exemplo, C++ oferece suporte a herança múltipla, ponteiros, estruturas, tipos **enum**, sobrecarga de operadores e uma sentença *goto*, mas Java não inclui nenhum desses recursos[16]. Os projetistas de C# discordaram com essas remoções, o que fez com que esses recursos, exceto herança múltipla, fossem trazidos de volta na nova linguagem.

Dando crédito aos projetistas de C#, entretanto, em diversos casos, a versão C# de um recurso C++ foi melhorada. Por exemplo, os tipos `enum` de C# são mais seguros, porque nunca são implicitamente convertidos para inteiros. Isso permite que sejam mais seguros em relação a tipos. O tipo `struct` foi modificado significativamente, resultando em uma construção verdadeiramente útil, enquanto em C++ tal construção é praticamente inútil. Estruturas (*structs*) em C# são discutidas no Capítulo 12 – "Suporte à Programação Orientada a Objetos". C# dá um passo à frente ao melhorar a sentença `switch` usada em C, C++ e Java. Nessas linguagens, não existe um desvio implícito no final dos segmentos selecionáveis de código, que causava inúmeros erros de programação. Em C#, quaisquer segmentos `case` não vazios devem terminar com uma sentença de desvio incondicional. A sentença `switch` de C# é discutida no Capítulo 8 – "Estruturas de Controle no Nível de Sentenças".

Apesar de C++ incluir ponteiros para funções, eles compartilham a falta de segurança inerente nos ponteiros para variáveis de C++. C# inclui um novo tipo, os *delegates*, referências a métodos que são tanto orientadas a objetos quanto seguras em relação a tipos. *Delegates* são usados para implementar manipuladores de eventos e *callbacks*[17]. *Callbacks* são implementadas em Java por meio de interfaces; em C++, ponteiros para métodos são usados.

Em C#, os métodos podem ter um número variável de parâmetros, desde que sejam do mesmo tipo. Isso é especificado pelo uso de um parâmetro formal do tipo vetor, precedido pela palavra reservada `params`.

Tanto C++ quanto Java usam sistemas de tipos distintos: um para tipos primitivos e outro para objetos. Além de ser confusa, essa distinção leva à necessidade de converter valores entre os dois sistemas – por exemplo, para colocar um valor primitivo em uma coleção que armazena objetos. C# faz

[16] N. de R. T.: Tipos `enum` estão disponíveis em Java desde a versão 1.5, de 2004.

[17] Quando um objeto chama um método de outro objeto e precisa ser notificado quando esse método tiver completado sua tarefa, o método chamado chama seu chamador novamente, o que é denominado um *callback*.

a conversão entre valores dos dois sistemas de tipos de forma parcialmente implícita por meio de operações de *boxing* e *unboxing*, discutidas em detalhes no Capítulo 12[18].

Dentre outros recursos de C#, estão os vetores retangulares, não suportados pela maioria das linguagens de programação, e uma sentença **foreach**, usada para iterar em vetores e objetos de coleção. Uma sentença **foreach** similar é encontrada em Perl, PHP e Java 5.0. Além disso, C# inclui propriedades, uma alternativa aos atributos de dados públicos. Propriedades são especificadas como atributos de dados com métodos de leitura e escrita, os quais são implicitamente chamados quando referências e atribuições são feitas aos atributos de dados associados.

2.19.3 Avaliação

C# foi criada como um avanço tanto em relação a C++ quanto em relação a Java como uma linguagem de programação de propósito geral. Apesar de ser possível argumentar que alguns de seus recursos são vistos como um passo atrás, C# inclui algumas construções que a movem à frente de suas antecessoras. Alguns de seus recursos certamente serão adotados por linguagens de programação em um futuro próximo.

A seguir, temos um exemplo de um programa em C#:

```
// Programa de exemplo em C#
// Entrada:  Um inteiro, listlen, onde listlen é menor do que
//           100, seguido por valores inteiros listlen.
// Saída:    O número de valores de entrada que são maiores
//           do que a média de todos os valores de entrada.
using System;
public class Ch2example {
  static void Main() {
    int[] intlist;
    int listlen,
        counter,
        sum = 0,
        average,
        result = 0;
    intList = new int[99];
    listlen = Int32.Parse(Console.readLiine());
    if ((listlen > 0) && (listlen < 100)) {
// Lê os dados de entrada em um vetor e calcula sua soma
      for (counter = 0; counter < listlen; counter++) {
        intList[counter] =
                      Int32.Parse(Console.readLine());
        sum += intList[counter];
      } //- end of for (counter ...
```

[18] Esse recurso foi adicionado à linguagem Java em sua versão 5.0.

```
// Calcula a média
      average = sum / listlen;
// Conta o número de valores que são maiores do que a média
      foreach (int num in intList)
        if (num > average) result++;
// Imprimir o resultado
      Console.WriteLine(
         "Number of values > average is:" + result);
    } //- end of if ((listlen ...
    else
      Console.WriteLine(
         "Error--input list length is not legal");
  } //- end of method Main
} //- end of class Ch2example
```

2.20 LINGUAGENS HÍBRIDAS DE MARCAÇÃO/PROGRAMAÇÃO

Uma linguagem híbrida de marcação/programação é uma linguagem de marcação na qual alguns dos elementos podem especificar ações de programação, como controle de fluxo e computação. As seguintes subseções introduzem duas linguagens híbridas, XSLT e JSP.

2.20.1 XSLT

XML (eXtensible Markup Language – Linguagem de Marcação Extensível) é uma linguagem de metamarcação usada para definir linguagens de marcação. Linguagens de marcação derivadas de XML são usadas para definir documentos de dados, chamados de documentos XML. Apesar de os documentos XML serem legíveis por humanos, eles são processados por computadores. Esse processamento algumas vezes consiste apenas em transformações para formatos que possam ser efetivamente visualizados ou impressos. Em muitos casos, as transformações são para XHTML, que pode ser mostrada por um navegador Web. Em outros, os dados no documento são processados, como outras formas de arquivos de dados.

A transformação de documentos XML para XHTML é especificada em outra linguagem de marcação, chamada de XSLT (eXtensible Stylesheet Language Transformations – Transformações em Linguagem de Folhas de Estilo Extensível) – www.w3.org/TR/XSLT. XSLT pode especificar operações similares àquelas de programação. Logo, XSLT é uma linguagem híbrida de marcação/programação. XSLT foi definida pelo World Wide Web Consortium (W3C – Consórcio Web) no fim dos anos 1990.

Um processador XSLT é um programa que recebe como entrada um documento de dados XML e um XSLT (também especificado na forma de um documento XML). Nesse processamento, o documento de dados XML é

transformado em outro XML[19], usando as transformações descritas no XSLT. O XSLT especifica as transformações pela definição de *templates*, padrões de dados que podem ser encontrados pelo processador XSLT no arquivo XML de entrada. Associadas a cada *template* no documento XSLT estão suas instruções de transformação, as quais especificam como os dados que casam com os *templates* devem ser transformados antes de serem colocados no documento de saída. Logo, os *templates* (e seu processamento associado) agem como subprogramas, que são "executados" quando o processador XSLT encontra um casamento de padrões nos dados do documento XML.

XSLT também tem construções de programação em um nível mais baixo. Por exemplo, uma construção de iteração é incluída, permindo que partes repetidas do documento XML sejam selecionadas. Existe também um processo de ordenação. Essas construções de baixo nível são especificadas com *tags* XSLT, como `<for-each>`.

2.20.2 JSP

A parte principal de JSTL (Java Server Pages Standard Tag Library) é outra linguagem híbrida de marcação/programação, apesar de seu formato e propósito serem diferentes daqueles de XSLT. Antes de discutir JSTL, é necessário introduzir as ideias de *servlets* e de *Java Server Pages* (JSP). Um **servlet** é uma instância de uma classe Java que reside e é executada em um sistema de servidor Web. A execução de um *servlet* é solicitada por um documento de marcação mostrado em um navegador Web. A saída de um *servlet*, feita na forma de um documento HTML, é retornada para o navegador requisitante. Um programa que roda no processo do servidor Web, chamado de **servlet container**, controla a execução dos *servlets*. *Servlets* são comumente usados para o processamento de formulário e para o acesso a bases de dados.

JSP é uma coleção de tecnologias projetadas para oferecer suporte a documentos Web dinâmicos e fornecer outras necessidades de processamento para documentos Web. Quando um documento JSP, normalmente um misto de HTML e Java, é solicitado por um navegador, o programa processador de JSP, que reside em um sistema servidor Web, converte o documento para um *servlet*. O código Java embarcado no documento é copiado para o *servlet*. O código HTML puro é copiado em sentenças de impressão Java que o mostram como ele é. A marcação JSTL no documento JSP é processada, conforme discutido no próximo parágrafo. O *servlet* produzido pelo processador JSP é executado pelo *servlet container*.

A JSTL define uma coleção de elementos de ações XML que controlam o processamento do documento JSP no servidor Web. Tais elementos têm o mesmo formato que outros de HTML e XML. Um dos elementos de controle de ação mais usados de JSTL é o `if`, que especifica uma ex-

[19] O documento de saída do processador XSLT pode ser também em HTML ou texto plano.

pressão booleana como um atributo[20]. O conteúdo do elemento **if** (o texto entre a *tag* de abertura (**<if>**) e a *tag* de fechamento (**</if>**) é código de marcação que será incluído no documento de saída apenas se a expressão booleana for avaliada como verdadeira. O elemento `if` é relacionado ao comando de pré-processador `#if` de C/C++. O contêiner JSP processa as partes de marcação JSTL dos documentos JSP de maneira similar à forma pela qual processadores C/C++ processam programas C e C++. Os comandos do pré-processador são instruções para que ele especifique como o arquivo de saída deve ser construído a partir do arquivo de entrada. De maneira similar, os elementos de controle de ação de JSTL são instruções para o processador JSP sobre como construir o arquivo de saída XML a partir do arquivo de entrada XML.

Um uso comum do elemento **if** é para a validação de dados de formulários submetidos por um usuário de um navegador. Os dados de formulários são acessíveis pelo processador JSP e podem ser testados com o elemento **if** para garantir que são os dados esperados. Se não forem, o elemento **if** pode inserir uma mensagem de erro para o usuário no documento de saída.

Para controle de múltipla seleção, JSTL tem os elementos **choose**, **when** e **otherwise**. JSTL também inclui **forEach**, o qual itera sobre coleções, em geral valores de formulário de um cliente. O elemento **forEach** pode incluir atributos **begin**, **end** e **step** para controlar suas iterações.

RESUMO

Investigamos o desenvolvimento e o ambiente de algumas das linguagens de programação mais importantes. Este capítulo deve ter dado ao leitor uma boa perspectiva em relação às questões atuais do projeto de linguagens. Esperamos ter preparado o terreno para uma discussão aprofundada dos recursos importantes das linguagens contemporâneas.

NOTAS BIBLIOGRÁFICAS

Talvez a fonte mais importante de informações históricas sobre o desenvolvimento de linguagens de programação seja History of Programming Languages, editada por Richard Wexelblat (1981). Ele contém a perspectiva histórica do desenvolvimento e do ambiente de 13 linguagens de programação importantes, de acordo com seus projetistas. Um trabalho similar resultou em uma segunda conferência sobre "história", publicada como uma edição especial de uma *ACM SIGPLAN Notices* (ACM, 1993a). Nesse trabalho, são discutidas a história e a evolução de mais 13 linguagens de programação.

O artigo "Early Development of Programming Languages" (Knuth e Pardo, 1977), parte da Encyclopedia of Computer Science and Technology, é um trabalho excelente de 85 páginas que detalha o desenvolvimento de linguagens até o Fortran

[20] Um atributo em HTML, embarcado na *tag* de abertura de um elemento, fornece informações adicionais sobre o elemento.