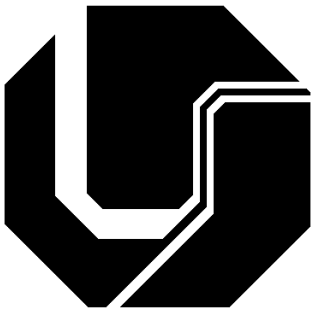
UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA



Linguagens de Domínio Específico e Sensores Baseados em Modelos Biológicos de Computação

PAULO SERGIO CAPARELLI

FEVEREIRO
2010

UNIVERSIDADE FEDERAL DE UBERLÂNDIA
FACULDADE DE ENGENHARIA ELÉTRICA
PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Linguagens de Domínio Específico e Sensores Baseados em Modelos Biológicos de Computação

Tese apresentada por Paulo Sergio Caparelli à Universidade Federal de Uberlândia para obtenção do título de Doutor em Ciências, aprovada em 19/02/2010 pela Banca Examinadora:

Antonio Eduardo Costa Pereira, PhD (UFU) (Orientador)
Evandro Afonso do Nascimento, PhD (UFU)
Alexsandro Santos Soares, Dr (UFG)
Jose Romildo Malaquias, Dr (UFOP)
Gilberto Arantes Carrijo, PhD (UFU)
Keiji Yamanaka, PhD (UFU)

Dados Internacionais de Catalogação na Publicação (CIP)

C236L Caparelli, Paulo Sergio, 1954-

Linguagens de domínio específico e sensores baseados em modelos biológicos de computação / Paulo Sergio Caparelli. - 2010.

200 f. : il.

Orientador: Antonio Eduardo Costa Pereira.

Tese (doutorado) – Universidade Federal de Uberlândia, Programa de Pós-Graduação em Engenharia Elétrica.

Inclui bibliografia.

1. Inteligência artificial - Teses. 2. Redes neurais (Computação) - Teses. 3. Reconhecimento de padrões - Teses. 4. Programação genética (Computação) - Teses. I. Pereira, Antonio Eduardo Costa. II. Universidade Federal de Uberlândia. Programa de Pós-Graduação em Engenharia Elétrica. IV. Título.

CDU: 621.32

Elaborado pelo Sistema de Bibliotecas da UFU / Setor de Catalogação e Classificação

Linguagens de Domínio Específico e Sensores Baseados em Modelos Biológicos de Computação

PAULO SERGIO CAPARELLI

Tese apresentada por Paulo Sergio Caparelli à Universidade Federal de Uberlândia como parte dos requisitos para obtenção do título de Doutor em Ciências.

Antonio Eduardo Costa Pereira, PhD. Orientador	Alexandre Cardoso, Dr. Coordenador do Programa de Pós-Graduação

À minha esposa Alabibia, pelo amor, entusiasmo e incentivos sempre constantes. Aos meus filhos Thiago e Renata e aos meus netos Artur e Hanna, que me motivam a sempre seguir adiante.

Agradecimentos

A Deus pela vida e por estar incondicionalmente sempre presente.

Aos professores Luís Carlos de Freitas e João Batista Vieira Junior, pelas produtivas discussões e pela oportunidade de trabalho conjunto.

Um muito obrigado também, a todos os amigos, que de uma forma ou de outra contribuíram para o desenvolvimento deste trabalho.

Conteúdo

Abstract

A Domain Specific Language is a specification language dedicated to a particular domain, representation technique, or solution searching method. On the other hand, a general-purpose programming language is a language designed with the goal of emulating Lambda Calculus or Turing Machine. Since general-purpose languages must accept any algorithm that can be executed by a Turing Machine, they do not contain a knowledge base of a specific domain, which makes them difficult to master by a professional who is not a specialist in Computer Science.

Many fields of science and technology have well advanced domain specific languages: LaTeX and XML for text processing, SQL for data base management, Matlab for engineering, etc. Designers of neural network applications do not have good domain specific languages. The main reason for this situation is that they are considered computer scientists, and supposed to know general-purpose languages and even low level languages close to Turing machines (languages for microprogramming). Since we know that this has no base on reality, the goal of this work is to develop methods of creating domain specific languages rooted on Lisp macros and on functional languages (Clean and Haskell) for prototypes. We believe that Lisp is easy enough to be mastered even by people who has difficulty with formal methods and mathematics. Therefore, with well designed extensions, and a rapid training in programming methods, an engineer can use Lisp as a powerful tool of productivity.

Lisp is a traditional tool for creating both embedded languages and domain specific languages. Therefore, Lisp offers all tools necessary to design, test and deploy domain specific languages. Nevertheless, neural networks

have features that require special care. For instance, they need to have biological feasibility. In order to reach code quality close to human designed nets, while preserving biological feasibility, this work will use advanced Artificial Intelligence methods and guided automatic theorem provers similar to the one used in HOL4 or Isabelle. Our system has a simpler goal than general theorem provers, since it will focus in a few problems related to neural networks. One of these problems is the designing of devices for data acquisition.

This thesis differs from similar works in one point that may cause controversy: Besides the contributions of the author to the state of the art, it discusses elementary aspects of the technologies involved. We decided to do this in order to bring the ideas that we are exposing to the reach of all readers. What are these ideas?

The first contribution is to bring software engineering technology to biological modeling. We are not aware of any researcher who attempted this goal before. A second aspect of this work can be resumed as an effort to create a language to describe combinations of neuron models into complex networks.

All programs discussed here were designed to avoid obsolescence. The author achieved this goal by writing prototypes in λ-calculus, a formal system that can be executed in modern computers.

Resumo

Linguagem de domínio específico é uma linguagem de especificação dedicada a um domínio particular, ou a uma técnica de representação, ou mesmo a um método de busca, no sentido dado a este termo pelos pesquisadores de inteligência artificial. Por outro lado, uma linguagem de propósito geral é projetada com a meta de emular o cálculo lambda ou a máquina de Turing. Considerando que as linguagens de propósito geral devem aceitar qualquer algoritmo que possa ser executado por uma máquina de Turing, elas raramente usam métodos específicos a um domínio restrito. Isso as torna difíceis de aprender por profissionais que não sejam especialistas em ciência de computação. Para dizer toda a verdade, o aumento da popularidade de linguagens específicas entre cientistas de computação faz nos crer que linguagens de propósito geral são difíceis até para esses profissionais.

Muitos campos da ciência e da tecnologia possuem avançadas linguagens de domínio específico: LaTeX e XML são utilizadas por pessoas que trabalham com processamento de texto, SQL é popular em áreas que dependem de gerenciamento de bases de dados, Matlab é amplamente utilizada entre engenheiros, etc. Já projetistas de aplicações baseadas em redes neurais não possuem boas linguagens de domínio específico. A principal razão para isso é que eles são considerados cientistas de computação e, por isso, deveriam — na opinião geral — conhecer bem linguagens de propósito geral e até mesmo linguagens de microprogramação. A realidade, contudo, não é essa; por isso, a meta deste trabalho é desenvolver uma metodologia de criação de linguagens de domínio específico, metodologia essa que é baseada no sistema de macros da Lisp e em protótipos escritos em linguagens funcionais baseadas no λ-calculus (Clean e Haskell). Lisp é a ferramenta clássica

para criar tanto sistemas embarcados quanto linguagens específicas. Por isso, Lisp oferece todas as ferramentas necessárias para projetar, testar e aplicar linguagens de domínio específico. Mesmo assim, redes neurais têm características que exigem cuidado especial. Por exemplo, elas devem exibir exequibilidade biológica. Para atingir qualidade próxima de código gerado por programadores profissionais, mantendo a exequibilidade biológica, o presente trabalho utiliza avançadas técnicas de inteligência artificial e provadores de teorema baseados em táticas fornecidas pelo usuário, similares às do HOL4, ou do Isabelle. O sistema descrito nesta tese tem metas mais simples do que um provador geral, já que focaliza alguns poucos problemas relacionados com redes neurais. Um desses problemas é a coleta de dados de temperatura para construir a verdade terrestre (verdade terrestre refere-se à informação que é recolhida "no local") de sistemas utilizados no estudo do aquecimento global.

Um aspecto em que esta tese difere de trabalhos similares e que pode gerar controvérsia está no fato do autor ter procurado escrever um texto autocontido. Tentou-se, ao mesmo tempo em que se discutiam as contribuições do autor, expor de maneira didática e completa todos os recursos tecnológicos utilizados. No caso de linguagens de domínio específico, várias foram apresentadas de forma mais ou menos completa, em uma tentativa de convencer o leitor da difusão e da importância dessa tecnologia.

No que se refere às contribuições do autor, elas são três. Em primeiro lugar, o autor tentou repassar ao leitor sua longa experiência no projeto e construção de dispositivos eletrônicos de alta tecnologia, como sistemas de aquisição de dados, GPS, interfaces e monitores miniaturizados. Em segundo lugar, procurou-se trazer as modernas técnicas de engenharia de *software* para a geração de redes neurais.

Finalmente, propôs-se nesta tese uma arquitetura virtual, que procura unificar os diversos algoritmos de redes neurais e programação genética, além de criar uma tecnologia resistente à obsolescência. Essa última meta é atingida através de protótipos escritos em λ-calculus, um sistema formal que serve de base aos computadores modernos.

Capítulo 1

Linguagens de Domínio Específico

Uma linguagem é um conjunto de símbolos léxicos (palavras), ou seja, uma sequência finita de letras ou caracteres, que podem ser gráficos ou não. Um caracter é gráfico se ele pode ser impresso. O conjunto dos caracteres válidos para formar um símbolo é chamado alfabeto. Uma linguagem é definida por meio de uma gramática recursiva. A gramática é recursiva se, dada uma sequência S de símbolos, é possível determinar se essa sequência pertence à linguagem com um número finito de passos. Se a sequência pertencer à linguagem, ela é chamada de *fórmula bem formada*. Esta é a definição informal de linguagem. Vamos examinar uma outra definição um pouco mais formal.

Linguagem é uma entidade matemática com estrutura léxica, sintática, semântica e pragmática. Formalmente, uma linguagem é definida por:

1. Um alfabeto, que é um conjunto de caracteres. Embora alfabetos possam ter cardinalidade infinita, vamos lidar apenas com alfabetos finitos. Caracter é um elemento do alfabeto. Vamos denotar o alfabeto pela letra grega Σ (sigma).

2. Um símbolo léxico (ou palavra) é qualquer sequência finita de caracteres. Vamos relembrar que caracter é um elemento do alfabeto, que é um conjunto. O conjunto de todas as palavras sobre o alfabeto Σ é

denotado por Σ^*, onde o asterisco é denominado fecho de Kleene. Para qualquer alfabeto, existe apenas uma palavra de comprimento zero, que é denotada por ε.

3. Concatenação é a operação que permite combinar duas palavras de comprimento l_1 e l_2 para formar outra de comprimento $l_1 + l_2$. O resultado de concatenar qualquer palavra w com a palavra ε é a própria palavra w.

4. Uma linguagem L sobre um alfabeto Σ é qualquer subconjunto de Σ^*.

As propriedades acima são suficientes para definir linguagem. Na prática, é preciso que a linguagem possua também uma gramática, que é um conjunto de regras que permite determinar se um elemento de Σ^* pertence à linguagem com um número finito de passos.

1.1 Regras de produção

Uma gramática consiste em:

1. Um conjunto finito de símbolos terminais. Informalmente um símbolo é terminal se não é definido por uma regra de gramática. Um símbolo terminal não pode ser analisado (quebrado) em termos de outros símbolos.

2. Um conjunto finito de símbolos não-terminais, ou seja, uma sequência de símbolos terminais e de outros símbolos não terminais.

3. Um conjunto finito de regras de produção, onde cada regra tem a forma:

$$\alpha_1\alpha_2\alpha_3 \ldots \rightarrow \beta_1\beta_2\beta_3\beta_4 \ldots$$

   α_i e β_i podem ser símbolos terminais ou não terminais. Regras de gramática com o formato acima são denominadas **regras de produção** ou **regras de derivação**.

4. Um símbolo inicial, pertencente ao conjunto de símbolos não terminais.

Consideremos uma gramática simplificada de expressões em português:

```
frase --> sintagma_nominal(_G, N), sintagma_verbal(N).

sintagma_nominal(G, N) --> artigo(G, N), substantivo(G, N).

sintagma_verbal(N) --> verbo_intransitivo(N).
sintagma_verbal(N) --> verbo_transitivo(N), sintagma_nominal(_, _).

artigo(m, s) --> [o].
artigo(f, s) --> [a].
artigo(m, p) --> [os].
artigo(f, p) --> [as].

substantivo(m, s) --> [gato].
substantivo(f, s) --> [gata].
substantivo(m, s) --> [rato].
substantivo(f, p) --> [rosas].

verbo_intransitivo(s) --> [fugiu].

verbo_transitivo(s) --> [pegou].
verbo_transitivo(s) --> [viu].
```

Figura 1.1: Regras de produção

Na gramática acima, toda regra tem a forma $\alpha \rightarrow \beta_1, \beta_2, \beta_3 \ldots \beta_n$. A seta pode ser interpretada assim: α é definido como β_1, seguido de β_2, seguido de β_3, etc. Os símbolos apresentados entre colchetes são símbolos terminais. Observe que um símbolo terminal nunca aparece no lado esquerdo de uma regra. Exemplos de símbolos terminais na figura 1.1: **o, gato e fugiu**. Na gramática da figura 1.1, os argumentos de alguns símbolos não terminais estabelecem regras de concordância. Por exemplo, a regra

```
frase --> sintagma_nominal(_G, N), sintagma_verbal(N)
```

informa que tanto o `sintagma_nominal`, quanto o `sintagma_verbal` devem ter o mesmo número `N`, que pode ser `s`/ingular ou `p`/lural. Tecnicamente, dizemos que o número do `sintagma_nominal` unifica-se com o número do `sintagma_verbal`.

Vamos examinar alguns casos concretos de interpretação das regras da figura 1.1, página 3:

`frase -->`
`  sintagma_nominal(_G, N),`
`  sintagma_verbal(N).`
	`frase` é `sintagma_nominal` seguido de `sintagma_verbal`

`sintagma_nominal -->`
` artigo(G, N),`
` substantivo(G, N).`
	`sintagma_nominal` é `artigo` seguido de `substantivo`

`sintagma_verbal(N) -->` `  verbo_intransitivo(N).`	`sintagma_verbal` é `verbo_intransitivo`.

`sintagma_verbal(N) -->`
`  verbo_transitivo(N),`
`  sintagma_nominal(_, _).`
	`sintagma_verbal` é `verbo_transitivo` seguido de `sintagma_nominal`.

O linguista francês [Alain Colmerauer] desenvolveu um sistema – denominado Prolog – para interpretar regras de gramática como a que acabamos de estudar. Abaixo mostramos o resultado da interpretação da regra de `frase`. Ao leitor que não está familiarizado com Prolog, recomendamos a leitura de um ou mais dos seguintes textos: [Eduardo Costa], [Sterling e Shapiro], [Elena Efimova] ou [Ivan Bratko].

A gramática da figura 1.1 (página 3) é derivacional ou de produção. Isto significa que, a partir de um símbolo da gramática, o sistema pode derivar (ou produzir) sentenças que obedecem à classe gramatical denotada pelo símbolo. Abaixo, mostramos exemplos de derivações produzidas em um computador por um intérprete de Prolog.

```
1 ?-
% c:/caparelli/simplgram.pl compiled 0.00 sec, 2,172 bytes
1 ?- frase(S, []).
S = [o, gato, fugiu] ;
S = [o, gato, pegou, o, gato] ;
S = [o, gato, pegou, o, rato] ;
S = [o, gato, pegou, a, gata] ;
S = [o, gato, pegou, as, rosas] .
```

O leitor deve ter notado que, na Prolog, o símbolo de frase possui dois argumentos, enquanto, na gramática listada na figura 1.1 da página 3, `frase` não tem argumentos. A gramática é conhecida tecnicamente pelos linguistas como DCG (Definite Clause Grammar). Um compilador de DCG acrescenta dois argumentos a cada símbolo da DCG. Isto é feito de modo que os argumentos fiquem encadeados, conforme mostrado na seguinte sentença:

```
frase(S0, S2) :- sintagma_nominal(G, N, S0, S1),
                 sintagma_verbal(N, S1, S2).
```

No processo de encadeamento, as variáveis lógicas `Si` são projetadas de modo que obedeçam à seguinte equação:

$$S_0 - S_2 = (S_0 - S_1) + (S_1 - S_2)$$

Na equação acima, fica claro a igualdade dos dois lados. De fato,

$$S_0 - S_2 = (S_0 - S_1) + (S_1 - S_2) \therefore S_0 - S_2 = S_0 - S_1 + S_1 - S_2 \therefore S_0 - S_2 = S_0 - S_2$$

Vamos pegar um caso concreto para entender melhor o que está acontecendo. Vamos imaginar que

$$S_0 = [\text{o, gato, fugiu}]$$

A diferença $S_0 - S_2$ é uma frase, e S_2 é a lista vazia, pois S_0 também é uma frase. A diferença entre S_0 e S_1 deve ser um sintagma nominal; portanto, S_1 deve ser `[fugiu]` e $S_0 - S_1$= `[o, gato]`. Finalmente, a diferença $S_1 - S_2$= `[fugiu]` e S_2= `[ ]`, como já havíamos dito.

1.2 Chomsky-Schützenberger

O linguista americano Noam Chomsky e o matemático francês Marcel-Paul Schützenberger descobriram, independentemente, a hierarquia que existe entre as linguagens. Chomsky não estava a par do trabalho de Schützenberger, tendo feito a descoberta de maneira independente. A reinvenção da roda não parou por aqui. Backus e Naur, dois cientistas de computação que ignoravam

tanto o trabalho do matemático Schützenberger quanto o do linguista Chomsky, repetiram a descoberta dos dois cientistas. Por isso, regras de gramática como as mostradas na figura 1.1 da página 3 são conhecidas, em alguns círculos, como formas de Backus-Naur. Elas também são chamadas de DCG (Definite Clause Grammar) em reconhecimento pelo trabalho de Fernando Pereira, um linguista português que observou a relação existente entre certas sentenças da lógica (cláusulas de Horn) e as regras de produção. Fernando Pereira também criou a notação em que os símbolos terminais aparecem entre colchetes e introduziu os argumentos que forçam a concordância e modelam a semântica. O cientista português deu o nome de gramática de cláusulas definidas (DCG) ao sistema modificado por ele.

A fidelidade histórica e o reconhecimento do mérito exigem que citemos também o trabalho do cientista indiano PaaNini, que dois mil e quinhentos anos antes dos cientistas modernos criou as regras de produção. Eis o que [Anirban Dash] diz sobre PaaNini:

> *PaaNini's work is devoted to the description of Sanskrit language. At the outset, it must be pointed out that, PaaNini's avowed goal was to provide an adequate descriptive grammar for Sanskrit and not to make a semantic analysis of the language. As a result, PaaNini focused only on deriving grammatically correct phrases and sentences, and not on the derivational process involving a number of syntactical, morphological and phonological operations. Thus, PaNini's grammar is primarily a derivational grammar.*

A descoberta das regras de produção teve profundo impacto na eletrônica e na computação. Aliás, o impacto na moderna eletrônica foi tão grande quanto na linguística. Todos os computadores e circuitos integrados são projetados com gramática derivacional. A contribuição de linguistas e filósofos à eletrônica e à informática não parou nas regras de produção. [Quine], um antigo professor da Universidade de São Paulo e autor de um livro em português, desenvolveu o algoritmo fundamental da eletrônica digital. A propósito, [Quine] era também professor de Harvard. Ao visitar a USP, notando que os estudantes de filosofia não conseguiam ler livros em latim, grego

e alemão, Quine nos presenteou com uma obra de primeira qualidade, digna dos melhores livros de filosofia escritos em outras línguas. Claro que a obra foi traduzida rapidamente para inglês e é nessa língua que os estudantes do mundo inteiro aprendem o sentido da nova lógica.

Além de descobrir o caráter produtivo da linguagem, Chomsky e Schützenberger descobriram uma hierarquia existente entre as línguas. Para eles, as linguagens são classificadas da seguinte forma:

- Linguagem Regular. A gramática regular pode ser analisada por um processador sem pilha. Este processador é conhecido como autômato finito.

- Linguagem livre de contexto. A gramática da página 3 é livre de contexto. Nesse tipo de gramática, só há um símbolo à esquerda da seta (propriedade exibida por todas as regras da página 3). Então, a gramática é livre de contexto se todas as suas regras têm a forma $\alpha \rightarrow \beta_1, \beta_2, \beta_3 \ldots$ As gramáticas livres de contexto precisam de um processador com pilha.

- Linguagem dependente de contexto. Aqui, as regras apresentam mais de um termo do lado esquerdo: $\alpha_1, \alpha_2 \rightarrow \beta_1, \beta_2, \beta_3 \ldots$ Essas linguagens são analisadas por uma máquina de Turing linear finita não determinística. Essas máquinas (fictícias) possuem um alfabeto finito, uma cabeça de leitura/escrita que pode ler ou escrever um símbolo por vez e um número finito de estados. Uma característica importante dessas máquinas é que são não-determinísticas, ou seja, podem voltar atrás, quando tomam um caminho errado na derivação.

- Linguagens recursivamente enumeráveis. Esse tipo de linguagem precisa de uma máquina de Turing para ser analisada.

1.3 DSL — Linguagens de Domínio Específico

Conforme vimos, uma linguagem pode ser definida por uma gramática. Uma linguagem de domínio específico permite restringir essa gramática de modo

que sua implementação se torne mais eficiente, mais expressiva ou mais clara em um domínio dado. As linguagens de domínio específico permitem também desenvolver uma programação orientada para o problema e discutir essa programação; em outras palavras, elas permitem a meta-programação.

Para entender melhor as DSLs, convém entender o que é problemática. O filósofo greco-macedônio Aristóteles afirmou que o processamento do conhecimento envolve três fases:

- A arte da definição. Aqui, dizemos o que são as coisas. Na computação, a definição consiste em declarar variáveis ou registradores.
- A arte da enunciação estabelece a problemática, ou seja, lança a hipótese de que existe uma relação entre dois conceitos.
- A arte da inferência estabelece a relação de causa e efeito que porventura existe entre os conceitos da problemática. Dessa forma, a inferência resolve a problemática.

As DSLs foram concebidas para responder a uma problemática específica. Nesse aspecto, elas se contrapõem às linguagens de propósito geral.

As linguagens de domínio específico alcançaram enorme popularidade nos últimos tempos, pois permitem que pessoas quase leigas ou com treinamento deficiente consigam operar nos meandros intrincados da informática moderna. Entre as linguagens de domínio específico, está a LaTeX, ferramenta de processamento de texto utilizada para redigir essa tese. Outra linguagem de domínio específico é a **Postscript**, criada para produção de documentos complexos. Vamos examinar de perto a **Postscript**, uma vez que pode ser utilizada na micro-eletrônica.

1.4 Postscript

Postscript é uma linguagem de domínio específico utilizada para instruções enviadas para impressoras. A linguagem foi baseada na Forth, uma linguagem que hoje é utilizada apenas em dispositivos de *plug-and-play* (instalação

de equipamentos). Apesar do uso da Forth ser restrito à instalação de equipamentos, **Postscript** é ainda largamente utilizada em dispositivos gráficos. O autor dessa tese criou uma versão da **Postscript** que pode ser utilizada diretamente em documentos, como essa tese. Vamos fazer um pequeno tutorial dessa linguagem para mostrar como são as linguagens de domínio específico.

O nome **Postscript** vem do fato de que os comandos são sufixos dos argumentos. Vamos imaginar que queiramos imprimir um texto, como o mostrado abaixo.

Domain Specific Languages

Para imprimir o texto acima, os comando são os seguintes:

```
\eps{
  %!PS-Adobe-2.0
  %%BoundingBox: 0 0 80 30
  /Times-Roman findfont
  20 scalefont
  setfont
  40 5 moveto
  (Domain Specific Languages) show
}
```

O primeiro passo, é estabelecer o tamanho da caixa gráfica onde o texto será desenhado. Isso foi realizado pelas seguintes linhas:

```
  %!PS-Adobe-2.0
  %%BoundingBox: 0 0 80 30
```

No caso, temos uma caixa de 80 por 30. O passo seguinte é escolher a fonte para o texto. Inicialmente, precisamos encontrar a fonte:

```
/Times-Roman findfont
```

Uma vez encontrada, a fonte é colocada em uma pilha, como todos os objetos da **Postscript**. A fonte que está na pilha tem dimensões minúsculas: 1×1.

Para redimensioná-la, vamos colocar o número 20 (a escala) na pilha e chamar a função `scalefont` que, como toda função da **Postscript**, pega seus argumentos da pilha. No caso, os argumentos são a fonte e a escala. Nesse ponto, a fonte está na pilha com o tamanho desejado. Chamamos então a função `setfont`, que pegará a fonte da pilha e instalará no desenhador.

```
/Times-Roman findfont
20 scalefont
setfont
```

O passo seguinte é mover a pena para a posição de coordenadas `(40,5)` e desenhar o texto. Para isso, colocamos os números 40 e 5 na pilha e chamamos a função `moveto`.

```
40 5 moveto
(Domain Specific Languages) show
```

O texto a ser desenhado é colocado entre parênteses e empurrado para a pilha. A função `show` faz o desenho.

Domain Specific Languages

O nome da linguagem **Postscript** tem origem na expressão latina POSTSCRIPTUM, que significa nota acrescentada ao fim de um texto. Como seu nome indica, a **Postscript** coloca as operações depois dos operandos:

```
GS>3 4 add =
GS> 7
```

Quando digitamos números, **Postscript** coloca-os em uma pilha. Examina-se a pilha com o comando `stack`, conforme pode ser visto abaixo. Observe que o número de objetos na pilha é indicado pelo *prompt* da **Postscript**; no exemplo abaixo, quando há dois objetos na pilha, o *prompt* indica esse fato: `GS<2>`. Para somar dois objetos do topo da pilha, usamos o operador `add`.

```
GS>3 4
GS<2>stack
4
3
GS<2>add
GS<1>stack
7
GS<1>=
7
```

No programa abaixo, mostramos como imprimir o resultado de um cálculo.

```
\eps{
%!PS-Adobe EPSF-3.0
%%BoundingBox: 0 0 400 200

/Times-Roman findfont
20 scalefont setfont

100 100 moveto
45 sin 8 string cvs
show }
```

0.707107

O programa que acabamos de ver calcula o seno de 45 graus, transforma o resultado em uma *string* de 8 caracteres, a qual é mostrada no terminal.

A principal aplicação da **Postscript** é, evidentemente, traçar gráficos. O programa abaixo mostra como isso é realizado.

```
\eps{
%!PS-Adobe EPSF-3.0
%%BoundingBox: 0 0 400 80
gsave   1 0.5 scale
0.4 setgray   70 100  48  0 360 arc fill
grestore
/Helvetica-Bold 14 selectfont
1.0 setgray
29 45 moveto (Hello, world!) show  }
```

A linguagem **Postscript** pode também ser usada para desenhar fórmulas estruturais de química orgânica, diagramas de ótica, partituras musicais e até mesmo obras de arte. Na figura figura 1.2, mostramos a diagramação do colesterol feita em **Postscript** por Philippos Apolinário[Philippos-2].

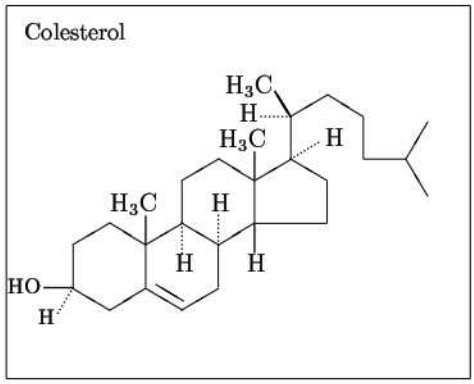


Figura 1.2: Diagramação do colesterol

A figura 1.3 mostra o resultado de um programa desenvolvido em **Postscript** por [Philippos] para traçar a trajetória de um raio de luz através de um prisma.

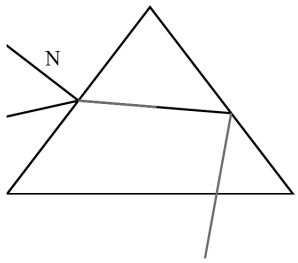


Figura 1.3: Prisma

1.5 Circuitos em Postscript

Postscript é também utilizada para desenhar circuitos elétricos e eletrônicos, sendo adequada até mesmo para diagramas precisos, utilizados na geração de máscaras de circuito integrado. Mostramos abaixo o desenho de um circuito elétrico `(RC)` e a codificação necessária para a sua construção.

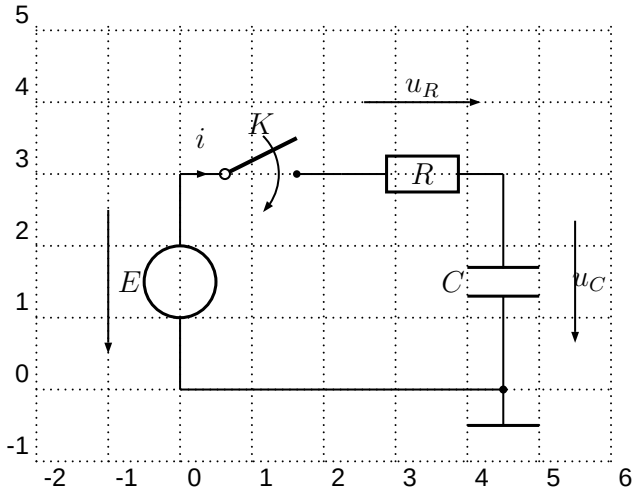


Figura 1.4: Circuito (RC)

```
\begin{pspicture}(-1.5, -1)(6,5)
\psgrid[subgriddiv=1,griddots=10]
\pnode(0,0){A}
\pnode(0,3){B}
\pnode(4.5,3){C}
\pnode(4.5,0){D}
\Ucc[tension,dipoleconvention=generator](A)(B){$E$}
\multidipole(B)(C)%
    \switch[intensitylabel=$i$]{$K$}%
    \resistor[labeloffset=0,tensionlabel=$u_R$]{$R$}.
\capacitor[tensionlabel={$u_C$},
     tensionlabeloffset=-1.2,tensionoffset=-1,
     directconvention=false](D)(C){$C$}
\wire(A)(D)
\ground(D)
\end{pspicture}
```

Considere o circuito da figura 1.5. O circuito da fonte de tensão foi descrito em uma linguagem de macros chamada **m4**. Os macros podem ser utilizados para gerar código em Postscript.

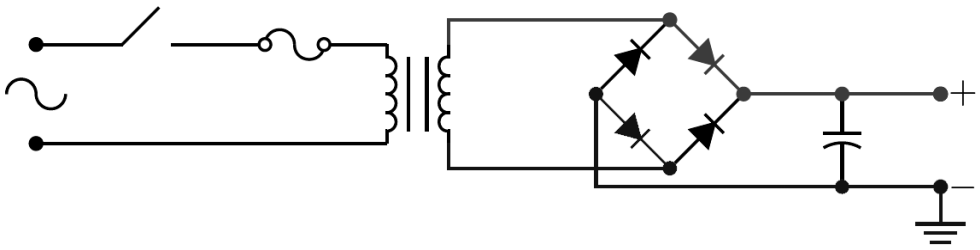


Figura 1.5: Fonte de tensão

```
% Pwrsupply.m4
.PS
cct_init
ifdef(‘m4pco’,‘resetrgb’)
  linewid = linewid*1.2
  down_
```

```
T:transformer
  line left_ elen_/4 from T.P1
  rgbdraw(0,0,1,fuse(left_ elen_/3,D))
  reversed('switch')
  gap(down_ to (Here,T.P2))
  { fuse(right_ 2*dimen_/5 at last []) }
  line to T.P2
  blen = dimen_/2
W: T.TS+(dimen_,0)
N: W+(blen,blen)
S: W+(blen,-blen)
E: S+(blen,blen)
  diode(from W to N)
  diode(from S to E)
G:gap(from E+(dimen_*4/3,0) down_ (E.y-S.y)*5/4); llabel(+,,-)
C:capacitor(down_ G.start.y-G.end.y from 0.5 between E and G.start,C)
setrgb(1,0,0)
  line from T.S1 to (T.S1,N) then to N; dot
  diode(to E); dot
  line from E to G.start; dot
  dot(at C.start)
resetrgb
setrgb(0,1,0)
  dot(at C.end)
  dot(at G.end)
  ground
  line to (W,Here) then to W; dot
  diode(to S); dot
  line to (T.S2,Here) then to T.S2
resetrgb
.PE
```

Esses dois exemplos devem ser suficientes para mostrar o poder das linguagens de domínio específico em geral, e da **Postscript** em particular. Em um artigo submetido à publicação, o autor do presente trabalho descreve um sistema de programação genética capaz de projetar e desenhar circuitos através da linguagem Postscript. Na figura 1.6, apresenta-se um modelo de linha de transmissão construído utilizando-se de elementos distribuídos. O diagrama foi gerado por um processo em tudo similar ao descrito em [Caparelli].

A linguagem **Postscript** é especialmente adequada para sistemas de programação genética por não exigir a construção de estruturas sintáticas complexas. Isso facilita o cruzamento e também as mutações (ver [Perkis]).

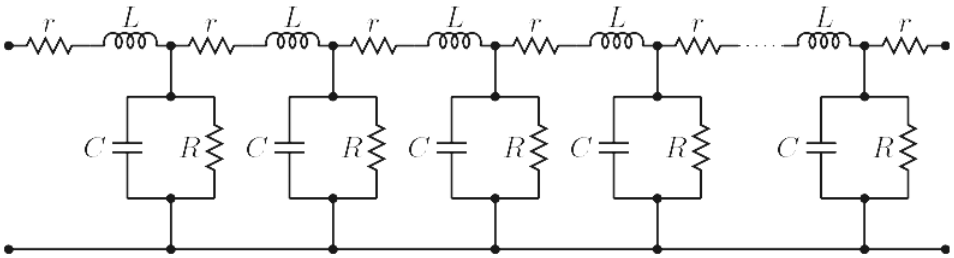


Figura 1.6: Modelo de linha de transmissão

No restante desta sessão, apresentaremos alguns circuitos elétricos e eletrônicos para exemplificar o poder da linguagem **Postscript**.

O desenho do transformador mostrado na figura 1.7 foi construído com a linguagem de macros **m4**. Essa linguagem, como dito anteriormente, gera o código **Postscript** necessário.

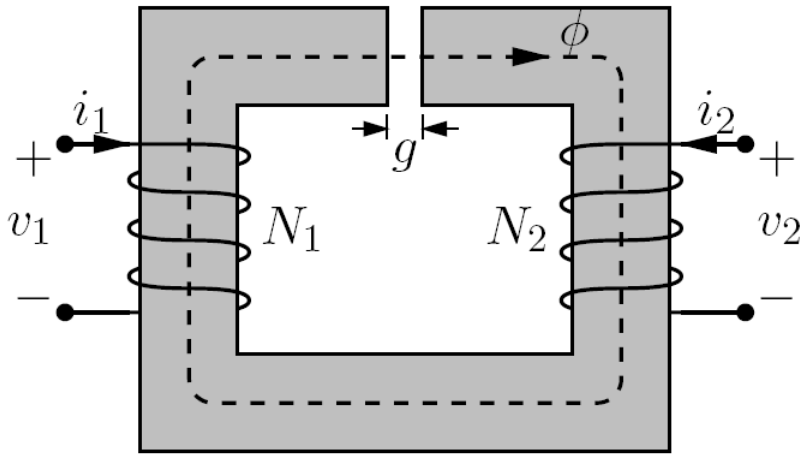


Figura 1.7: Transformador

A figura 1.8 abaixo mostra o diagrama eletrônico do estágio de amplificação de audio de um receptor de rádio.

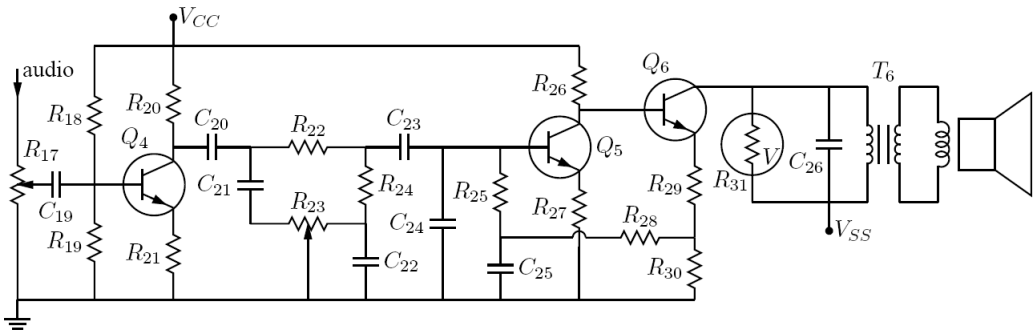


Figura 1.8: Estágio de amplificação de audio

Na parte esquerda da figura 1.9, podemos ver uma combinação de portas lógicas. Na parte direita da mesma figura, temos um diagrama mostrando como conectar os componentes eletrônicos discretos que implementam o circuito combinacional da esquerda.

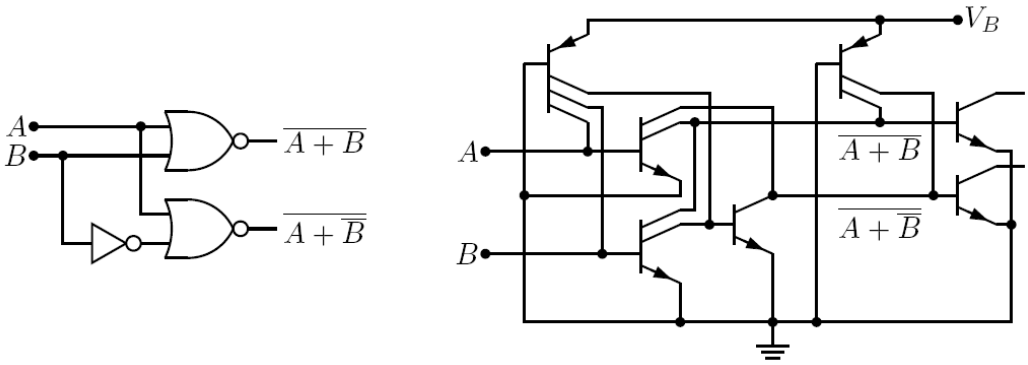


Figura 1.9: Portas Lógicas

A figura 1.10 exibe o diagrama de componentes discretos usado para especificar uma porta NAND com três entradas. A figura 1.11 mostra a estrutura de uma outra porta NAND, que utiliza a tecnologia CMOS.

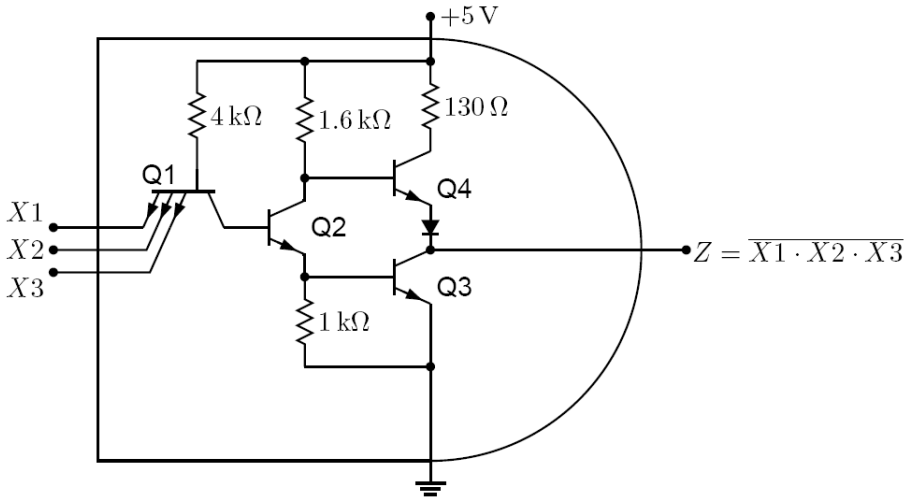


Figura 1.10: Porta Nand TTL

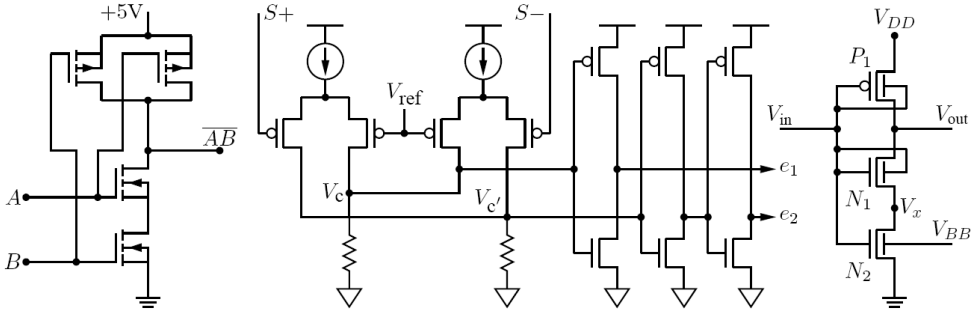


Figura 1.11: Porta Nand CMOS

A única finalidade da figura 1.12 é mostrar que se pode rotacionar diagramas. O circuito elétrico em si combina elementos capacitivos, indutivos e resistivos.

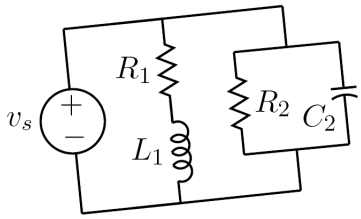


Figura 1.12: Rotação de diagramas

Na figura 1.13, vê-se um diagrama de um circuito que pode operar como oscilador senoidal trifásico. A figura 1.14 mostra um multivibrador biestável.

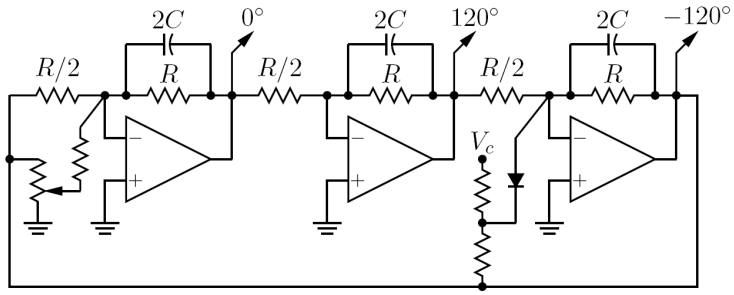


Figura 1.13: Oscilador trifásico

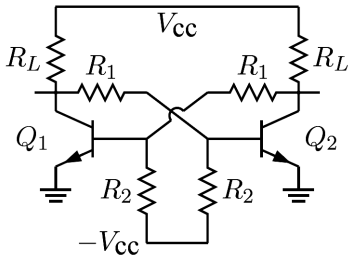


Figura 1.14: Multivibrador biestável

Na figura 1.15, tem-se um exemplo de diagrama elétrico de um circuito que contém um grupo gerador-motor. No motor, contatores possibilitam inversão de fase.

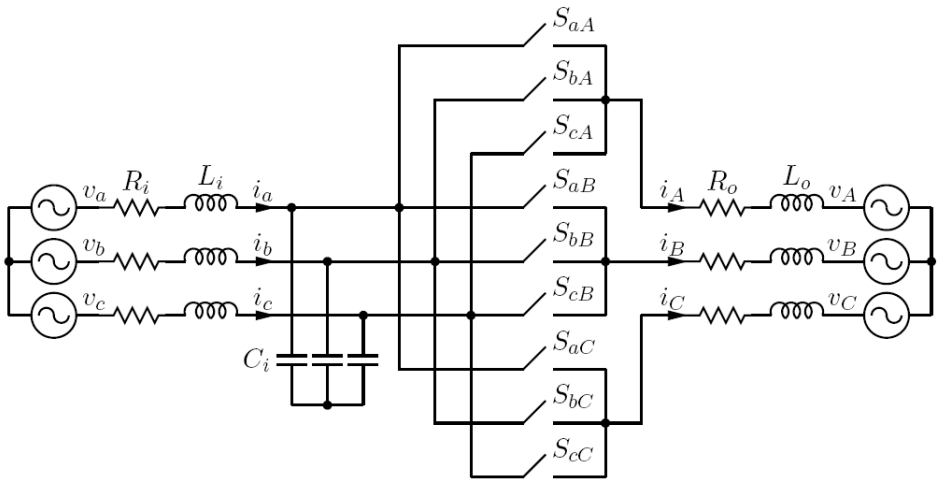


Figura 1.15: Grupo Gerador-Motor

Na figura 1.16, temos um temporizador (*timer*). Há dois LEDs, um vermelho e um verde. O vermelho é aceso quando o *timer* está ativo.

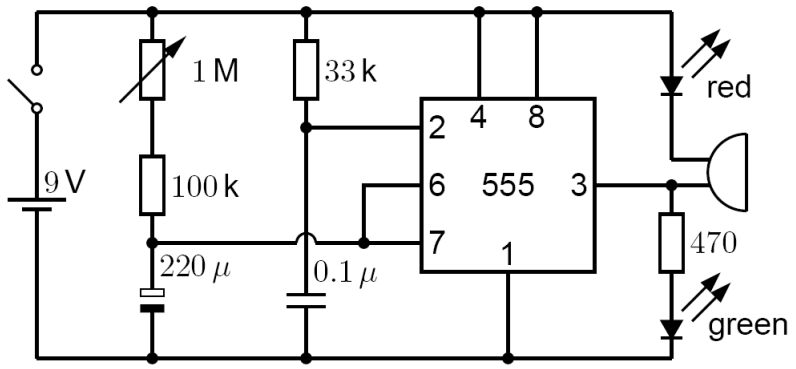


Figura 1.16: Temporizador

Nas figuras 1.17 e 1.18 seguintes, mostramos exemplos de circuitos lógicos sequenciais e combinacionais.

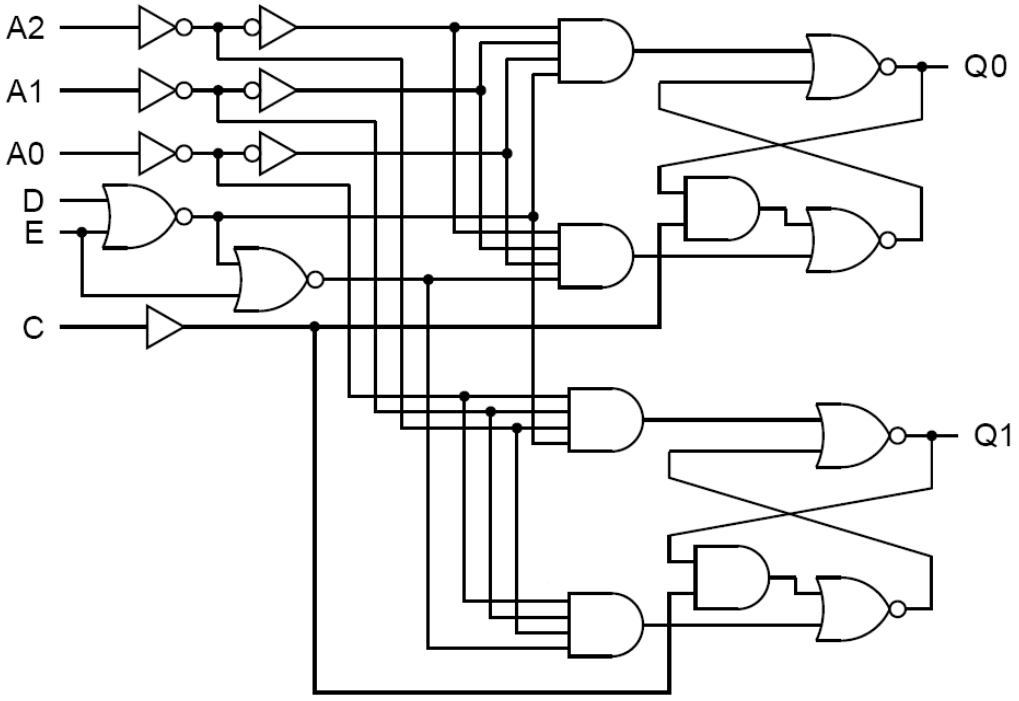


Figura 1.17: *Latch* de uso geral

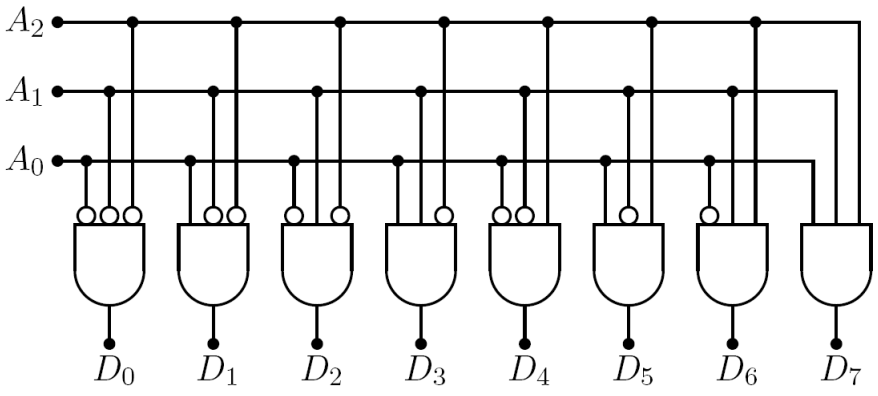


Figura 1.18: Decodificador lógico

A figura 1.19 mostra o diagrama de construção de um registrador de deslocamento para a direita.

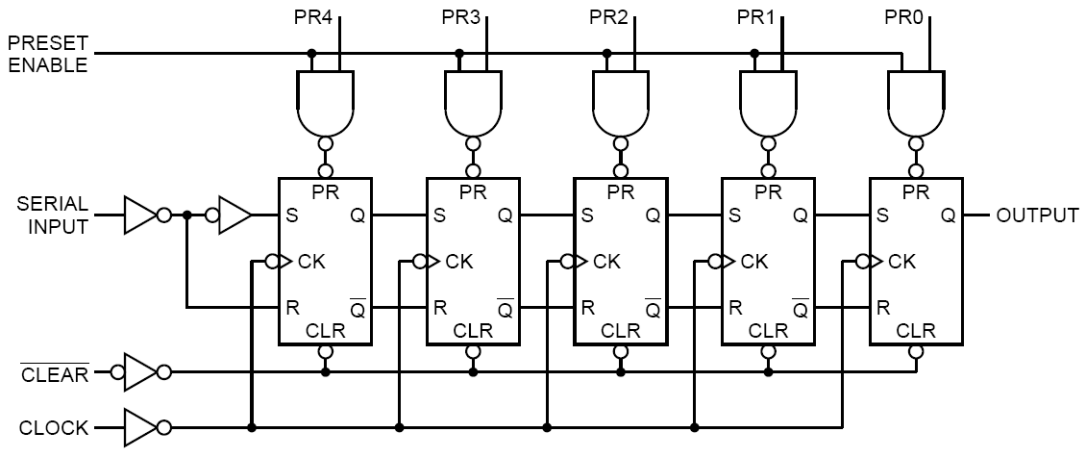


Figura 1.19: Registrador de Deslocamento para a Direita

Na figura 1.20, mostramos uma fonte de tensão com retificador e variador de velocidade, alimentando uma máquina síncrona.

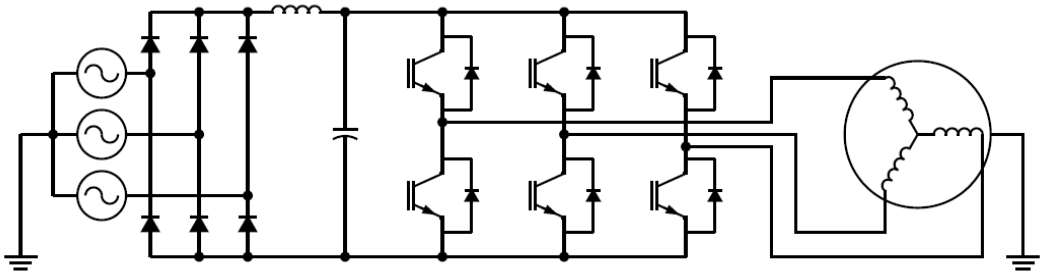


Figura 1.20: Máquina Síncrona

1.6 Obsolescência de programas

Além de linguagens de domínio específico, esta tese analisa também as causas da obsolescência de *software*, discute os problemas que ela acarreta e propõe um método para minimizar seus efeitos. De fato, o estudo da obsolescência de *software* é uma contribuição original do autor da tese e foi apresentado em um artigo publicado no International Journal of Computational Intelligence (ver [Caparelli]).

A sociedade industrial moderna tornou-se possível pela engenhosidade de Honoré Blanc. Antes dele, mecanismos complexos como mosquetes eram fabricados um por um. Cada parte era manufaturada, ajustada e limada para encaixar no objeto que o artesão estava montando no momento. Se um mosquete precisava de concerto, ele era enviado ao armeiro que tomava medidas e construía uma peça de reposição para o componente defeituoso.

Em 1790, Honoré Blanc fez uma reunião com um grupo de oficiais de alta patente, diante dos quais ele montou mosquetes com peças padronizadas retiradas de uma caixa. Na ocasião, Thomas Jefferson já tinha visitado a oficina de Blanc e tomado conhecimento de sua metodologia.

Dezoito anos depois da demonstração de Blanc, quando Jefferson era presidente dos Estados Unidos, autoridades militares, tendo tomado consciência da necessidade de defender o país, começaram a rearmar-se. Por volta de 1798, o secretário do tesouro Wolcott enviou um dos relatórios de Honoré Blanc a um certo Eli Whitney. Para divulgar a nova idéia, Whitney falsificou uma re-encenação da demonstração de Blanc. Ele fabricou manualmente cada peça e rotulou disfarçadamente os componentes que deveriam se encaixar na montagem de um dado mosquete. O golpe funcionou e o governo americano prometeu comprar entre dez e quinze mil mosquetes do esperto engenheiro. Acontece a que as peças de Whitney não eram padronizadas como as de Blanc, de modo que o armeiro americano levou oito anos para entregar a encomenda. Apesar disso, o entusiasmo de Whitney em promover as idéias de Blanc, colocou-as na base da indústria manufatureira moderna.

Com o advento dos computadores, os fabricantes dessas máquinas precisavam de dois subsistemas — *hardware* e *software*. Em princípio, *software*

podia ser projetado para funcionar em qualquer *hardware*. O problema é que limitações tecnológicas, tais como velocidade e memória, impuseram sérias restrições na indústria de equipamentos digitais. Quando essas limitações são eliminadas pelo progresso tecnológico os programadores enfrentam um dilema:

- Eles podem conservar suas linguagens, métodos e bibliotecas de modo a executar software antigo nas novas máquinas. Nesse caso, o sistema será privado da funcionalidade que as melhorias tecnológicas tornaram disponível.

- Eles podem também escrever um sistema mais poderoso, com novos recursos e mecanismos de segurança. Este é, em geral, o caminho seguido.

A conclusão da discussão acima é que *software* torna-se obsoleto, e necessita de substituição constante. A dificuldade é que não há mão de obra suficiente para re-escrever aplicações reiteradamente.

Em [Caparelli], este autor apresenta uma análise suscinta da obsolescência de *software*, e faz algumas propostas de como remediá-la, de modo que a grande contribuição de Honoré Blanc à indústria possa ser aplicada ao desenvolvimento de aplicações computacionais. É claro que ninguém pode esperar uma solução de curto prazo para esse problema, já que qualquer solução envolve a eliminação de hábitos enraizados e métodos prediletos. De fato, o próprio Honoré Blanc não teve muito sucesso em mudar os processos de manufatura de seu tempo. De acordo com [Lienhard], desde que Honoré Blanc usou trabalhadores não especializados, *he had made factories independent of government control over crafts*. Por isso as autoridades constituídas fecharam o negócio de Blanc.

Linguagens de computador se tornam obsoletas devido às restrições impostas pelo hardware subjacente. Por exemplo, o estado da tecnologia pode impor limitações de memória que impedem a implementação de estruturas de dados dinâmicas. Por esta razão, uma linguagem de computador pode precisar de reutilizar variáveis. Esta é a principal razão para a ampla utilização de atualizações destrutivas em linguagens convencionais de computador.

Velocidade ou razões de segurança podem impedir a implementação de otimizações globais ou de chamada de cauda(TCO – *Tail Call Optimization*). Em poucas palavras, o estado da arte da tecnologia faz com que os projetistas de linguagens evitem sistemas matemáticos e criem métodos formais que são mais fáceis de implementar e empregar. Quando avanços tecnológicos tornam as modificações desnecessárias, as linguagens de computador utilizadas são abandonadas ou atualizadas, e o profissional de informática não consegue mais compilar a safra de *software* anterior às mudanças (*vintage software*).

1.7 Programação funcional

Linguagens limitadas pela tecnologia (TCL – Technologically Conditioned Languages) são aquelas cuja sintaxe e semântica são determinadas pelo estágio de desenvolvimento tecnológico dos computadores. O modelo de memória, a falta de coleta de lixo, a representação deficiente de tipos abstratos de dados, a estratégia de execução e a orientação para problemas numéricos foram impostos aos projetistas de linguagens pelas limitações do *hardware* de que dispunham. Se John Backus tivesse tentado dotar FORTRAN (a primeira linguagem de alto nível) com melhores recursos de processamento de dados, essa linguagem seria lenta demais para as aplicações em vista, e não poderia ser utilizada nos computadores fabricados em 1950. É interessante notar que [John Backus] deixou claro que não queria FORTRAN com as características que tem. O projeto do cientista americano foi ditado inteiramente pelo *hardware* disponível, e pelo conhecimento que se tinha, na época, sobre técnicas de compilação. O que Backus realmente queria era uma linguagem funcional baseada na lógica combinatória (um sistema formal criado pelo matemático ucraniano [Moses Schonfinkel]). Aqui está o que Backus escreveu sobre FORTRAN e linguagens similares num discurso proferido quando ele recebeu a prestigiosa medalha de Turing:

> *Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-*

> *time style of programming. . . , their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.*

Podemos ver pelo discurso de Backus, que ele não gostava de linguagens limitadas pela tecnologia, nem mesmo a que ele propôs.

Existem linguagens de computador cujas propriedades são ditadas não pela própria tecnologia, mas pela maneira coma as pessoas a percebem. A maioria dessa linguagens orientadas ao homem são adaptações do λ-Calculus (veja [Henk Barendregt]), enquanto outras vêm de evoluções da lógica simbólica. De agora em diante, referir-nos-emos a essa família de linguagens como linguagens lógicas.

No próximo capítulo, faremos uma apresentação didática do λ-Calculus de modo a facilitar o entendimento do resto desse trabalho.

Capítulo 2

λ-Calculus

Neste capítulo, apresentaremos o λ-Calculus, um sistema formal criado pelo filósofo e matemático americano [Alonzo Church]. Nosso texto é, basicamente, um *potpourri* dos seguintes livros:

- The Lambda Calculus, Its Syntax and Semantics, Henk Barendregt. North-Holland.
- The Haskell Road to Logic, Math and Programming, Kees Doets e Jan van Eijck. College Publications.
- Introduction to Functional Programming, Richard Bird e Philip Wadler. Prentice Hall.
- Clean, The Computer Language, Auctore Ignoto.
  Conexões em `clean.cs.ru.nl` e `www.discenda.org`, ambas consultadas em 03/01/2010.

2.1 Gramática

A gramática do λ-Calculus tem poucas regras, que definem expressões λ. Essa gramática é apresentada abaixo na notação de Backus-Naur, conforme é costumeiro em Ciência de Computação. Embora expressões λ mostrem

variações sintáticas, vamos nos concentrar na forma utilizada pelos implementadores da linguagem Haskell. A gramática da linguagem Clean é semelhante à de Haskell e não deve apresentar problemas ao estudioso. A sintaxe da Lisp, que é mais concisa e flexível, será estudada em um capítulo a parte.

$< \text{expr} >$	::=	$< \text{id} >$	x, y, z, u, v
constantes		$< \text{constant} >$	π
abstrações		$\lambda\ x_1\ x_2 \ldots \rightarrow < \text{expr} >$	$\lambda\ x \rightarrow \sin\ x$
		$(\lambda\ x_1\ x_2 \ldots \rightarrow < \text{expr} >)$	$(\lambda\ x \rightarrow \sin\ x)$
aplicações		$< \text{expr} > < \text{e} >_1 < \text{e} >_2 \ldots$	$\sin\ x$
		$(< \text{expr} > < \text{e} >_1 < \text{e} >_2 \ldots)$	$(\sin\ x)$
definições		$< \text{id} > < \text{e} >_1 \ldots = < \text{e} >$	$f\ a = a \times a$
		$< \text{id} > < \text{e} >_1 \mid < \text{e}_c > = < \text{e} >$	$f\ a \mid a > 0 = a \times a$

Uma função abstrata é uma expressão onde o identificador `<id>` é o nome de um parâmetro, podendo ser ele *bound* (com valor) ou *unbound/free* (sem valor). Uma expressão do tipo $(fn\ \ x)$ é denominada aplicação. As aplicações são associativas à esquerda, ou seja, `f x y= (f x) y`. As constantes incluem `True` e `False` (valores booleanas), números e algumas funções como `(+)`, `(-)`, `sin`, `tail` e `head`.

A gramática definida acima implementa apenas expressões prefixas, que são as mais flexíveis. Entretanto, por conveniência, linguagens como Clean e Haskell utilizam notação infixa para operações aritméticas e construção de listas. Assim, escreve-se $main = print 2 \times 3 + 4 \times 5$ em Haskell; na notação prefixa, essa expressão seria escrita como `(+ (* 2 3) (* 4 5))`. Nos próximos capítulos, veremos a notação prefixa de forma mais aprofundada.

Conforme [Michael Gordon], uma expressão aritmética como $(2 + 3) \times 5$ pode ser representada como uma expressão λ e seu valor 25 também pode ser representado como expressão λ. A simplificação de $(2 + 3) \times 5$ para 25 é realizada por um processo chamado de conversão em alguns círculos e redução em outros.

Há três tipos de conversão-λ, a saber, conversão-α, redução-β e conversão-η. Na próxima seção, vamos estudar essas três conversões em detalhes.

2.2 Redução β, conversão α e substituição

No caso de aplicações, o processo de avaliação de uma expressão λ é chamado de redução-β.

$$(\lambda x \ \rightarrow \ E)\ f \qquad\qquad \Rightarrow E[f/x]$$

A definição acima indica que a expressão $(\lambda x \ \rightarrow \ E)\ f$ pode ser re-escrita como $E[f/x]$. A notação $E[f/x]$ significa que todas as ocorrências de x em E devem ser substituídas por f. Tomemos um exemplo concreto.

$$(\lambda\ a\ b\ c\ \rightarrow\ b^2 - 4 \times a \times c)\ 2\ 8\ 3 \underset{\beta}{\rightarrow} 8^2 - 4 \times 2 \times 3 \ \Rightarrow\ 40$$

Tanto Haskell quanto Clean oferecem uma maneira conveniente de dar nomes a expressões-λ. Eis como definir a expressão acima, associando-a com o identificador *delta*:

```
1 delta a b c = b^2-4*a*c

3 del = λ a b c → b^2-4*a*c

5 main= print (delta 2 8 3)
```

No exemplo acima, a definição da linha 1 é, em tudo, similar à definição da linha 3. A linha 1, entretanto, é mais conveniente, pois dispensa o uso do λ. A linha 1 pode ser interpretada como significando: *Re-escreva* delta a b c *como* $b^2-4\times a\times c$. A re-escrita só ocorre quando uma expressão unificar com o lado esquerdo da equação mostrada na linha 1. O processo de unificação atribuirá valores às variáveis `a`, `b` e `c` de tal forma que a expressão-λ `delta 2 8 3` torne-se igual ao lado direito da equação da linha 1.

Qualquer expressão na forma $(\lambda\ V\ \rightarrow\ E)$ pode ser convertida em $(\lambda\ x\ \rightarrow\ E[x/V])$ desde que a substituição de V por x em E seja válida. Esse processo é chamado de conversão α.

Qualquer expressão na forma $(\lambda\ V\ \rightarrow\ (E\ V))$ pode ser reduzida a E, desde que não haja ocorrências livres de V em E. Esse processo recebe o nome de conversão η.

2.3 Haskell

A fim de proteger-nos contra a obsolescência e usar a rica herança da matemática e da lógica, decidimos trabalhar com uma linguagem funcional baseada no cálculo-λ. Há duas opções Clean e Haskell. Ainda que a Clean seja segura e mais elegante que a Haskell, e mais fiel ao cálculo-λ, este trabalho apresenta seus resultados em Haskell, considerando-se que ela é mais amplamente conhecida do que a Clean. De fato, Haskell é usada como ferramenta de aprendizado em vários cursos superiores. Entretanto, o leitor deve levar em consideração a possibilidade de trabalhar em Clean quando houver requisitos de segurança e desempenho.

Já que Haskell é um ramo da matemática, ela pode ser facilmente aprendida por aqueles que tem conhecimentos práticos de análise funcional, teoria de conjuntos e álgebra. No resto desta seção, o leitor interessado encontrará uma curta introdução à programação funcional.

Abaixo, está um pequeno programa que calcula os fatores primos de um número natural. A função computável `factors` é definida assim:

```
1 import System

3 -- Least divisor starting from k
4 ldf k n | rem n k==0 = k
5         | k^2 > n = n
6         | otherwise= ldf (k+1) n

8 factors 1 = []
```

```
 9 factors n = p : factors (n ÷ p) where
10   p= ldf 2 n

12 main = do
13   args ∈ getArgs
14   case args of
15     [n] → print $ factors (read n)
16     otherwise → putStrLn "e.g. usus: factors 57"
```

A função `getArgs` retorna os argumentos da linha de comando na forma de uma lista de **String**s (cadeia de caracteres). O primeiro argumento dessa lista é convertido em um valor numérico pela função `read`. Aqui é apresentado um exemplo de uso do programa acima:

```
C:\fp>ghc -O2 factors.hs --make
[1 of 1] Compiling Main
Linking factors.exe ...

C:\fp>factors.exe 5754
[2,3,7,137]
```

A fim de melhor entender como esse programa funciona, vamos remover a função `factors`.

```
1 -- File: args.hs
2 import System(getArgs)

4 main = do
5     a ∈ getArgs
6     print a
```

Eis como compilar e executar `args.hs`:

```
C:\fp>ghc args.hs --make
[1 of 1] Compiling Main
Linking args.exe ...
```

```
C:\fp>args 4 5 6 7
["4","5","6","7"]
```

Como pode ser visto no exemplo acima, a função `getArgs` produz os argumentos da linha de comando como uma lista de *strings*. Na função `main`, a variável `args` unifica-se com a lista `[n]`, e atribui a *string* `"5754"` à variável `n`. Finalmente, `(read n)` converte a *string* em um número inteiro.

A definição de `ldf` (menor divisor — *least divisor*) é bastante óbvia. As barras verticais introduzem condições; e.g. ldf k n | **rem** n k==0 = k significa: *Se o resto* **rem** *n k é igual a 0,* `k` *é divisor de* `n`. Considerando-se que `k` começa do menor primo, e é incrementado a cada interação, ele é o menor divisor.

A função `factors n` é definida como um conjunto de equações de re-escrita. Cada equação tem um lado esquerdo e um lado direito. Se a expressão unifica-se com o lado esquerdo, ela é re-escrita na forma do lado direito. Por exemplo, a expressão `factors 1` unifica-se com a primeira equação, e é re-escrita como `[ ]` (lista vazia, pois 1 não tem fatores maiores do que 1). A expressão `factors 8` unifica-se com a segunda equação; o lado direito constrói uma lista cujo primeiro elemento é `p` (o menor divisor de 8), e cuja cauda contém os fatores de `n` dividido por 2; já que `n` é 8, a cauda contém os fatores de 4.

Listas são sequências ordenadas de elementos. Em geral, uma lista é representada por seus elementos entre colchetes. Em Hasekell, o construtor de lista é dois pontos (:); por exemplo, a expressão `(3:[2,4])` constrói a lista `[3,2,4]`, cujo primeiro elemento é `3`, e cuja cauda é `[2,4]`. Os dois componentes de uma lista (head – cabeça e tail – cauda) são desmontados pelo processo de unificação; por exemplo, se o padrão `(x:xs)` unifica-se com `[1,2,3]`, a cabeça da lista é atribuída a `x`(`x=1`), enquanto a cauda é atribuída a `xs` (`xs=[2,3]`).

Matemáticos têm uma notação para representar conjuntos que foi adotada tanto por Clean quanto por Haskell; a tal notação é chamada de notação de Zermelo-Frankel, em honra aos dois matemáticos que fizeram importantes contribuições à teoria dos conjuntos.

Tanto em matemática, quanto em Haskell, [(i, i^2) | i ∈ [1..n]] representa uma lista de pares `(i, i^2)`, onde $i \in [1..n]$, e i^2 é o quadrado de `i`. Nota Bene, a maioria dos programadores Haskell prefere ← em vez de ∈.

O leitor deve ter notado que Haskell usa a palavra reservada '`do`' para introduzir uma sequência de comandos. Contudo, essa construção nada mais é do que uma alternativa à notação de Zermelo-Frankel.

Haskell tem um intérprete, útil para testar idéias, e um compilador, quando velocidade e eficiência estão na mesa de apostas. Examinando-se a seguinte sessão do intérprete, percebe-se que a notação-`do` é equivalente à notação de Zermelo-Frankel.

```
1 Prelude> [(x, x*x) | x ∈ [2..6]]
2    [(2,4),(3,9),(4,16),(5,25),(6,36)]
3 Prelude> do x ∈ [2..6]; return (x, x*x)
4    [(2,4),(3,9),(4,16),(5,25),(6,36)]
```

Por volta de 1932, o lógico americano [Alonzo Church] introduziu os conceitos de funções computáveis e de λ-Calculus no mundo da matemática. O λ-Calculus original, sem tipo, mostrou-se inconsistente, mas Church publicou dois outros artigos em 1936 (ver [Church-1] e [Church-2]), onde uma modificação da teoria original provou-se consistente. Por volta de 1990, matemáticos como Barendregt, Hughes, Plasmeijer e Peyton-Jones apoiaram-se no λ-Calculus em um esforço para criar uma linguagem de computação que pudesse resolver dois dos maiores problemas da Ciência da Computação: obsolescência de software e falta de clareza referencial.

A falta de clareza referencial tem a ver com funções que apresentam efeitos colaterais. Em código escrito em C, por exemplo, não basta olhar o texto do programa para entender o que ele faz, ou para encontrar um erro de codificação. Afinal, o processo computacional pode modificar os valores das variáveis, e, portanto, o desenvolvedor precisa de traçar a história da computação para localizar a causa do erro. Por outro lado, matemáticos não precisam de examinar o processo de prova utilizado por Arquimedes a fim de verificar além de qualquer dúvida que as demonstrações do grande geômetra

grego estão corretas. A matemática é referencialmente clara, enquanto C é referencialmente opaca.

Linguagens opacas escondem erros e tornam os programas obscuros. Esse fato dificulta a prova de correção de uma implementação, pois algoritmos são especificados em uma linguagem que é clara, mas implementada em outra, que é obscura. Ferramentas matemáticas podem ser utilizadas para provar que a especificação está correta, mas é quase impossível usá-las para demonstrar que a implementação também está certa.

2.4 Programação paralela

Visto que as linguagens funcionais são baseadas na matemática, elas tem a capacidade de efetivamente usar formas de combinação poderosas para construir novos programas a partir de programas existentes ([John Backus]). Esta capacidade permite-lhes acompanhar tanto a tecnologia de *hardware* quanto a de *software*. Por exemplo, máquinas modernas possuem múltiplos processadores, o que lhes permite processamento paralelo; programadores que utilizam linguagens tradicionais não são capazes de usar essa nova tecnologia na implementação de aplicações que explorem paralelismo de *hardware*. Abaixo, mostraremos um exemplo de quão facilmente um programa Haskell pode ser paralelizado em um PC que possua múltiplos processadores.

Criptografia é uma das mais importantes tecnologias da nossa sociedade baseada em computador. Ela é usada não somente para proteger privacidade e segredos industriais ou militares, mas também bens, visto que transações financeiras e comerciais apresentam uma dependência crescente da Internet.

Um dos métodos mais úteis de criptografia é a codificação de chave pública; este método é baseado em números primos grandes. Um número primo é divisível somente por ele mesmo. Verificar a primalidade de um número é um problema difícil. Então, uma abordagem frequentemente utilizada é usar computação paralela. Um teste simples (mas muito ineficiente) para descobrir se um número é primo consiste em verificar se ele é igual ao seu menor divisor. O programa abaixo aplica esse método em paralelo para achar uma lista de números primos.

```
1  import Data.Maybe
2  import Control.Parallel.Strategies
3  import Control.Parallel
4  import System

6  ld n= ldf 2 where
7    ldf k | rem n k == 0 = k
8          | k^2 > n = n
9          | otherwise= ldf (k+1)

11 prime n | n==1 = False
12         | otherwise= ld n==n

14 prim n= if (prime n) then n else 0

16 bigNums = [25431045773439..]
17 primlist n = (parMap rwhnf) prim (take n bigNums)

19 main= do
20    args ∈ getArgs
21    case args of
22      [n] → prt (primlist (read n))
23      otherwise → putStrLn "e.g. usus: pprims 5"

25 prt [] = return ()
26 prt (x:xs) | x==0 = prt xs
27 prt (x:xs)= do print x
28                prt xs
```

Aqui está a linha de comando usada para compilar e rodar o programa acima:

```
D:\par>ghc -O2 -threaded pprims.hs --make
[1 of 1] Compiling Main
Linking pprims.exe ...

D:\par>pprims.exe 400 +RTS -N2
25431045773453
25431045773461
25431045773477
25431045773503
25431045773813
```

Na linha 16, o programa atribui uma lista infinita de números inteiros à variável `bigNums`. A lista começa com `25431045773439`. Na linha 17, seleciona-se `n` números da lista de `bigNums` por meio de (take n bigNums); isso é sempre possível porque a lista de `bigNums` tem um número infinito de elementos; como pode ser visto, Haskell permite o uso de conceitos matemáticos como sequências infinitas a fim de raciocinar sobre programas. Quanto à forma `(parMap rwhnf)`, ela aplica `prim` aos elementos de `(take n bigNums)`, produzindo uma lista cujos elementos ou são números primos ou são substituídos por 0.

A função `parMap` é um exemplo do que Backus chama de formas de combinação poderosas. Ela aplica uma função a cada elemento de uma lista. O argumento `rwhnf` (*reduce to weak head normal form*) é uma estratégia de execução paralela; o leitor interessado deve consultar [Peyton Jones]. A figura 2.1 mostra como `parMap` distribui o processamento entre os dois núcleos de um computador de mesa. A máquina utilizada no experimento é um modelo antigo, a fim de mostrar que é possível explorar o paralelismo mesmo em *hardware* de safra antiga.

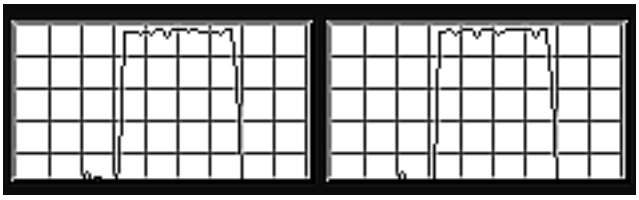


Figura 2.1: Uso de CPU no processamento paralelo

Para garantir a completitude, mostra-se abaixo uma implementação de criptografia de chave pública. Uma característica interessante da Haskell e da Clean é a possibilidade de automaticamente derivar métodos para diferentes classes de programas. Por exemplo, a criptografia pública tem duas chaves, uma para a codificação, e outra para a decodificação. A chave de codificação é tornada pública, e quem quiser enviar informações codificadas para o receptor pode usá-la. O receptor é a única entidade que tem a chave de decodificação privada. Portanto, somente o receptor pode acessar a men-

sagem. A linha 2 permite a Haskell ler e mostrar tanto as chaves públicas quanto as privadas, que são representadas por uma estrutura de dados. Um trabalho semelhante em linguagem convencional requer tecnologia de construção de compiladores e analisadores léxicos e sintáticos complexos. Aqui está uma sessão interpretada do programa de criptografia:

```
Prelude> :l "crypto.hs"
*Main> let (pr,pub)= genRSAKey prim1 prim2
*Main> pr
PRIV 64673808913334359238677993 1
24874541889742028088278190 5
*Main> pub
PUB 64673808913334359238677993 1
12934761782665854605904659 3
*Main> let code= ersa pub 12345678987654
*Main> code
95741030030096934463520585
*Main> drsa pr code
12345678987654
```

```
1  data RSAKey = PUB Integer Integer | PRIV Integer Integer
2                        deriving (Show, Read)

4  mInverse d f= loop (1, 0, f) (0, 1, d) where
5      loop (x1, x2, x3) (y1, y2, 0)= (0, y1)
6      loop (x1, x2, x3) (y1, y2, 1) | y2<0= (1, f+y2)
7      loop (x1, x2, x3) (y1, y2, 1)= (1, y2)
8      loop (x1, x2, x3) (y1, y2, y3)=
9        loop (y1, y2, y3) (x1-q*y1, x2-q*y2, x3-q*y3)
10         where q= x3 ÷ y3

12 expm m b k = ex b k 1 where
13   ex a k s
14     | k == 0 = s
15     | mod k 2 == 0 = ((ex (mod (a*a) m)) (k ÷ 2)) s
16     | otherwise = ((ex (mod (a*a) m)) (k ÷ 2)) ((s*a) `mod` m)
```

```
18 invm :: Integer → Integer → Integer
19 invm m a
20    | g /= 1 = error "No inverse exists"
21    | otherwise = x `mod` m
22  where (g,x) = mInverse a m

24 genRSAKey p q = (PRIV n d,PUB n e) where
25     phi = (p-1)*(q-1)
26     n = p*q
27     e = find (phi ÷ 5)
28     d = invm phi e
29     find x
30        | g == 1 = x
31        | otherwise = find ((x+1) `mod` phi)
32             where (g, _) = mInverse x phi

34 ersa (PUB n e) x = expm n x e
35 drsa (PRIV n d) x = expm n x d

37 prim1= 25431045773477
38 prim2= 25431045773503
```

Esta seção é sobre a linguagem Haskell, e não sobre criptografia. No entanto, o autor escolheu criptografia para mostrar Haskell em ação, pois trata-se de um problema interessante com um forte componente matemático, que revela os melhores recursos da programação funcional.

Como já mencionado, existem duas chaves, uma pública e uma privada. Seja e a chave pública, d a chave privada, e n o produto de p e q (dois números primos grandes). As chaves pública e privada devem obedecer à seguinte relação: $d \times e = 1 \bmod \phi(pq)$, i.e., $d \times e = 1 + k \times \phi(n)$. Nesse caso, o texto cifrado c é dado por $c = m^e \bmod n$.

Decifrar uma mensagem é possível porque $c^d = m^{ed} \bmod n$. Considerando-se que $d \times e = 1 + k \times \phi(n)$, tem-se:

$$m^{ed} = m^{1+k\phi(n)} = m(m^{\phi(n)})^k = m \bmod n = m, \text{ if } m < n$$

2.5 Prova da correção

Já que Haskell tem um forte embasamento na matemática, é possível utilizar princípios matemáticos para provar que um programa em Haskell está correto. Essa é uma característica bastante desejável em sistemas críticos, onde um erro pode ser fatal ou muito caro. Nos parágrafos seguintes, uma explicação detalhada sobre como isso pode ser feito é apresentada no caso da função `mInverse`.

Na vida real, os programas e circuitos eletrônicos não são óbvios. Por isso, engenheiros e cientistas da computação precisam da matemática para provar que seus programas estão corretos. Vamos provar que `mInverse` encontra o inverso multiplicativo de um número, i.e., se `mInverse d f == 1` e d<f, então existe d^{-1} tal que $d \times d^{-1} = 1 \bmod f$. Como se pode ver, esta propriedade é a base da criptografia de chave pública, conforme descrito na seção anterior.

A função `mInverse` se baseia em um *loop* para encontrar o inverso multiplicativo de seu primeiro argumento. Para provar que o *loop* está correto, é preciso procurar as relações que permanecem invariáveis em cada iteração. A essas relações é dado o nome de invariantes de laço. No caso do *loop* local da função `mInverse`, os invariantes são:

1. $f \times x1 + d \times x2 = x3$
2. $f \times y1 + d \times y2 = y3$

Ambos os invariantes são verdadeiros no início do *loop*, já que $(x1, x2, x3) = (1, 0, f)$ e $(y1, y2, y3) = (0, 1, d)$. Vamos supor que os invariantes sejam verdadeiros na iteração n. Neste caso, eles também são verdadeiros na iteração $n+1$. Afinal de contas, $f \times x1_{n+1} + d \times x2_{n+1} = f \times (x1_n - q \times y1_n) + d \times (x2_n - q \times y2_n) = (f \times x1_n + d \times x2_n) - q \times (f \times y1_n + d \times y2_n) = x3 - q \times y3 = x3_{n+1}$. O mesmo argumento pode ser usado para provar que, se a relação é válida para $(y1_n, y2_n, y3_n)$, então também é válida para $(y1_{n+1}, y2_{n+1}, y3_{n+1})$.

O resultado que demonstramos revela que, uma vez que o invariante é verdadeiro para a iteração 0, ele é verdadeiro para a iteração 1; já que ele é verdadeiro para a iteração 1, ele é verdadeiro para a iteração 2; e assim

por diante. Em particular, o invariante é verdadeiro para a última iteração. Contudo, na última iteração, $y_3 = 1$, e o segundo invariante toma a seguinte forma: $f \times y1 + d \times y2 = 1$. Então, $d \times y2 = 1 + (-y1) \times f$, e $d \times y2 = 1 \; mod \; f$, o que leva à conclusão de que $y2$ é o inverso multiplicativo de d.

Capítulo 3

Sistema de aquisição e processamento de sinais

Uma das contribuições deste trabalho é propor uma arquitetura para uma rede de *sensores* que possam ser utilizados no acompanhamento médico de pacientes e na monitoração das alterações climáticas.

Já que diferentes sinais exigem diferentes amplificações e filtragens de ruídos, tanto o amplificador como o filtro devem ser programáveis.

3.1 Hardware - Visão Geral

O diagrama de blocos do *hardware* de um dos sistemas desenvolvidos por este autor é mostrado na figura 3.1. São conectados sensores específicos para cada tipo de sinal, ligados a um módulo de condicionamento. Esse, por sua vez, envia o sinal para o módulo de conversão, que por fim envia os dados digitalizados ao transmissor. Chegando ao transmissor, o sinal é enviado ao módulo de recepção, que recompõe a informação e a envia para o módulo de comunicação com o computador principal.

O computador principal também é capaz de comunicar-se com os outros módulos, de modo a permitir o controle total do sistema de aquisição de dados.

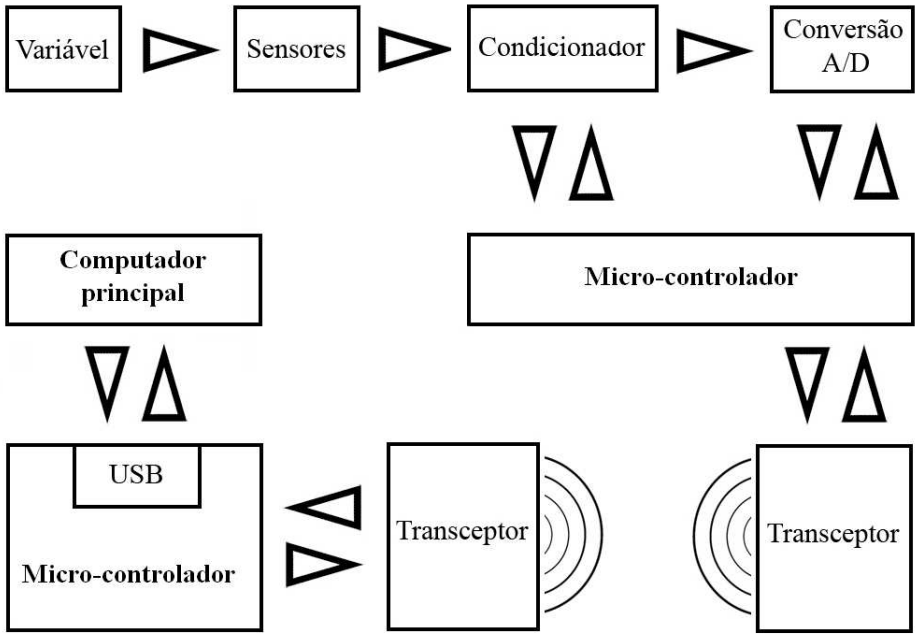


Figura 3.1: Diagrama de blocos do sistema

3.2 Variável

A variável representa qualquer tipo de informação de interesse, que se deseje coletar. Pode ser tanto uma informação de fluxo de combustível em um motor, pressão atmosférica em uma determinada altitude, pressão parcial de CO_2, temperatura, um biopotencial, ou qualquer outro tipo de informação.

3.3 Sensores

O sensor é um dispositivo que transfere energia entre dois sistemas. Ele deve ser capaz de detectar variações à sua volta (efetivamente, a variável de interesse), convertendo essas variações em sinais elétricos. Ao se trabalhar com sensores, é importante analisar os seguintes parâmetros:

- Exatidão — o quanto a medição se aproxima do valor real.
- Resolução — representa a menor grandeza que o sensor pode determinar
- Faixa de Operação — uma expressão da extensão total dos possíveis valores de medição.

No sistema desenvolvido, os sensores medem temperatura, pressão parcial de CO_2, e atividade elétrica de músculos, coração e cérebro, ou seja, podem lidar com sinais utilizados nos seguintes sistemas:

- Eletrocardiografia (ECG) — aquisição da atividade elétrica do coração ao longo do tempo captada e registrada por eletrodos de pele.
- Eletroencefalografia (EEG) — gravação da atividade elétrica de disparo de neurônios do cérebro.
- Eletromiografia (EMG) — gravação da atividade elétrica de ativação muscular.
- Termometria — série temporal da temperatura do corpo ou da temperatura ambiente sobre um único ponto de uma superfície.
- Capnografia — monitoração da pressão parcial de dióxido de carbono (CO_2) dos gases respiratórios. É utilizada no monitoramento de pacientes durante a anestesia e em cuidados intensivos.

O sistema foi projetado, no caso de sinais fisiológicos (ECG, EEG, EMG), para captação diferencial de sinais provenientes de pares de eletrodos Ag/AgCl. Para sinais de temperatura e pressão parcial de CO_2, a captação diferencial não é necessária.

3.4 Subsistema de Condicionamento de sinais

Os sinais captados pelos sensores geralmente possuem amplitudes baixas e são contaminados por todo tipo de interferência. Assim sendo, necessitam de

ser preparados para manipulação futura. O condicionamento envolve duas ações principais:

- Amplificação — A amplificação é o processo de elevação da magnitude do sinal, garantindo que ele esteja em valor adequado para as etapas de filtragem e digitalização.

- Filtragem — O processo de filtragem elimina componentes espectrais indesejadas, sejam elas originadas de interferência, ou pertinentes ao sinal medido [Lathi].

O sistema de condicionamento foi projetado para garantir a fidelidade de cada sinal coletado. Esse processo pode ser desmembrado em três estágios, de acordo com a figura 3.2

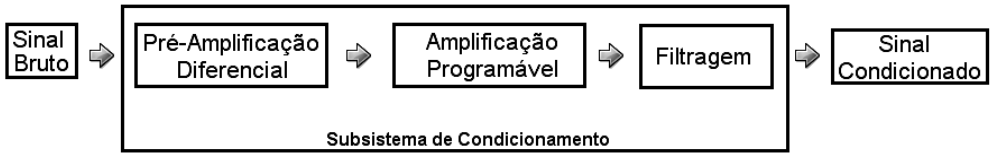


Figura 3.2: Estágios do subsistema de condicionamento

O primeiro estágio é responsável pela captação do sinal dos eletrodos em forma diferencial, realizando uma pré-amplificação fixa e adicionando um *offset* ao sinal resultante para que esse fique adequado ao processamento pelos estágios posteriores. A figura 3.3 ilustra o esquema elétrico desse estágio no sistema desenvolvido. Esse estágio é uma adaptação multicanal do estágio de amplificação denominado *High Precision Analog Front End* descrito pela TEXAS Instruments [Texas], e indicado para aplicações de alta precisão em sistemas médicos.

O sistema de realimentação funciona como um filtro passa-altas sintonizado em 0.05Hz, eliminando o *offset* gerado pela interface pele-eletrodo. A elevação da tensão média é obtida através da manipulação da tensão de referência dos amplificadores. Foi escolhido um amplificador de instrumentação

que possui uma alta razão de rejeição em modo comum, alta impedância de entrada, alta linearidade, baixo consumo e baixo ruído, sendo indicado pelo fabricante para utilização em aplicações médicas.

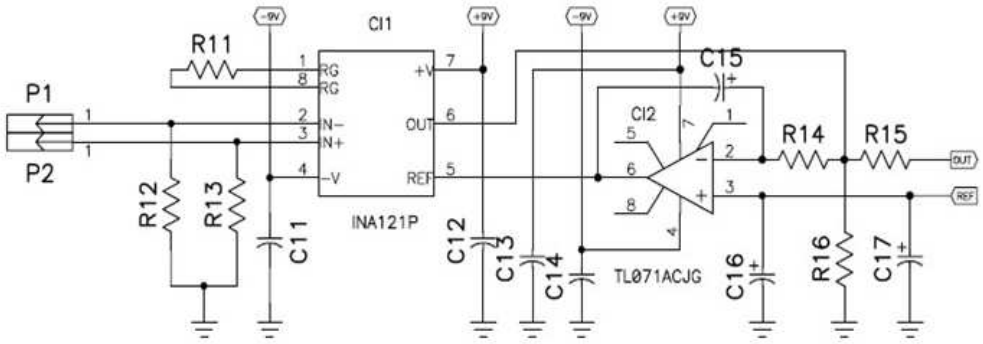


Figura 3.3: Esquema elétrico do estágio pré-amplificador

O segundo estágio é responsável pelo ajuste do condicionamento, permitindo que o sistema trabalhe com diferentes sinais. Isso é obtido através do uso de amplificadores programáveis. A figura 3.4 apresenta o esquema elétrico do amplificador utilizado no sistema desenvolvido.

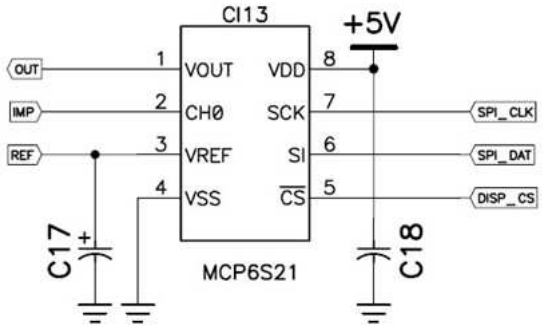


Figura 3.4: Esquema elétrico do estágio de amplificação programável

Esse estágio deve possuir ganho ajustável, para permitir amplificação máxima do sinal sem saturação do canal de aquisição, garantindo a melhor análise possível para cada tipo de sinal coletado.

O amplificador utilizado na implementação possui as seguintes características: baixo consumo, baixo *offset* (275μV max), baixo ruído, largura de banda de -3dB de 2Hz a 12MHz e ganho de tensão programável em 8 níveis (1, 2, 4, 5, 8, 10, 16 e 32 vezes) (erro `< 1%`), selecionáveis através de um barramento `SPI`, sendo apropriado para o sistema proposto.

O terceiro estágio é o responsável por realizar a filtragem *anti-aliasing*, eliminado o conteúdo espectral não desejado para a aplicação. Como o sistema trabalha com sinais que possuem diferentes conteúdos espectrais, a frequência de corte do filtro passa-baixa deve ser ajustável.

Uma frequência de corte máxima de 4KHz é suficiente para o sistema proposto, pois os sinais analisados não possuem conteúdo espectral acima desse valor (ver [Basha], [Berbari] e [Niedermeyer]). Na figura 3.5 é apresentado o diagrama elétrico do circuito de filtragem desenvolvido.

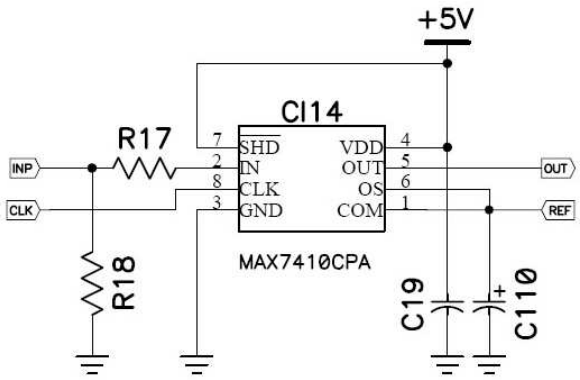


Figura 3.5: Esquema elétrico do estágio de filtragem

O processo de *aliasing* refere-se ao fenômeno de uma componente de alta frequência assumir a identidade de uma componente de baixa freqüência (ver figura 3.6). O *aliasing* distorce o espectro do sinal original. Para evitar este problema, o teorema de [Nyquist] define que a quantidade de amostras por unidade de tempo de um sinal amostrado no tempo deve ser maior que o dobro da maior frequência contida no sinal, de modo que este possa ser reproduzido fielmente.

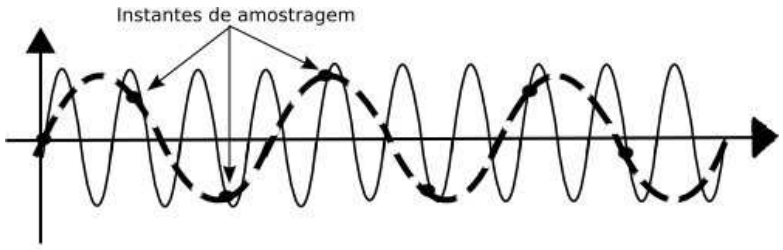


Figura 3.6: *Aliasing* de um sinal no domínio do tempo

No sistema desenvolvido, foi utilizado um filtro a capacitor chaveado, que possui característica *Butterworth* passa-baixa de quinta ordem (modelo mostrado na figura 3.7) e frequência de corte (sintonizável por *clock*) que pode variar de `1Hz` a `15kHz`, permitindo um controle preciso da frequência máxima do sinal a ser digitalizado.

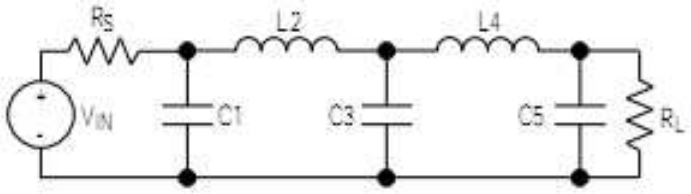


Figura 3.7: Modelo do filtro utilizado

3.5 Subsistema de Conversão A/D

O subsistema de conversão A/D é o responsável pela transformação dos sinais analógicos, provenientes do sistema de condicionamento, em sinais digitais. Esse processo é mostrado na figura 3.8 e ocorre em dois estágios:

- Multiplexação — Técnica que permite a combinação de vários sinais independentes num só sinal composto, destinado a ser transmitido por um meio físico comum. Na multiplexação por `divisão de tempo` o tempo é dividido entre as entradas em uma ordem pré-definida, e na saída tem-se um único sinal em um determinado instante.

- Conversão A/D — Transforma cada amostra do sinal em um valor com um determinado número de bits. O número de bits é determinado pela precisão da representação de cada amostra, ou seja, quanto maior o número de bits, maior a precisão com que a amostra é representada.

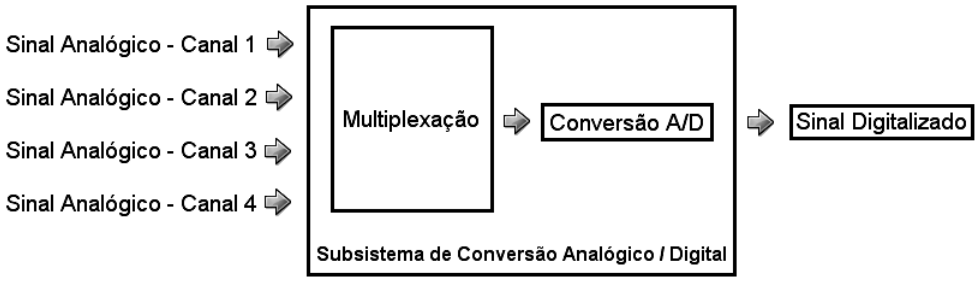


Figura 3.8: Estágios do subsistema de conversão analógico-digital

O primeiro estágio compreende a fase de multiplexação, onde vários canais analógicos são apresentados um a um ao conversor analógico digital, para que esse processe cada canal independentemente. O diagrama elétrico do estágio de multiplexação utilizado é apresentado na figura 3.9.

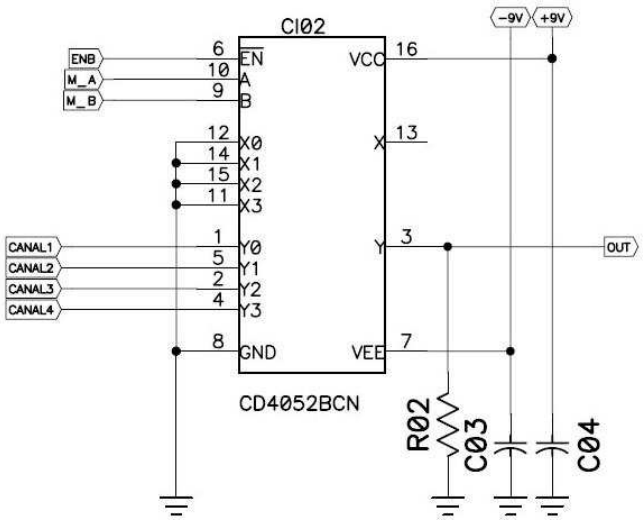


Figura 3.9: Diagrama elétrico do estágio de multiplexação

O multiplexador escolhido para compor o projeto possui alta velocidade de chaveamento e quatro canais selecionáveis.

O próximo estágio é o responsável pela conversão análogo-digital e seu esquema elétrico é apresentado na figura 3.10.

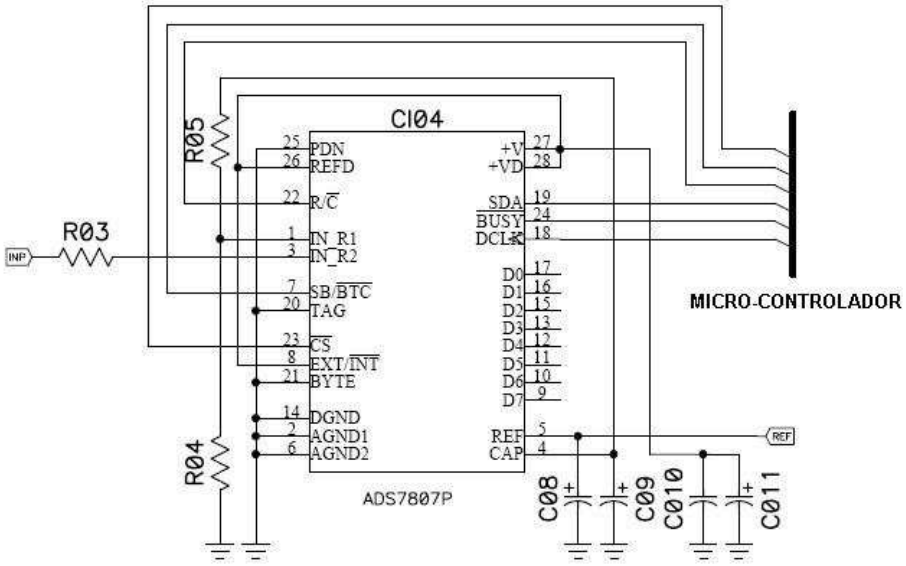


Figura 3.10: Diagrama elétrico do estágio de conversão A/D

O conversor utilizado no sistema desenvolvido efetua a aquisição e conversão de sinais com taxa máxima de 40kHz, codifica as amostras com 16 bits através do método de aproximações sucessivas (SAR)[Burr-Brown] e possui baixo consumo. Além disso, possui interfaces paralela e serial para comunicação com micro-controladores, facilitando sua incorporação ao sistema.

3.6 Controle dos subsistemas

O circuito de controle dos módulos que compõem os subsistemas de condicionamento de sinais e conversão A/D recebe comandos do computador principal através do sistema de transmissão (RF), permitindo o controle remoto de todas as variáveis envolvidas no processo. Esse circuito é responsável pelas

seguintes operações:

- Controle individual dos ganhos dos amplificadores.
- Controle da freqüência de corte dos filtros.
- Controle do processo de conversão análogo-digital (chaveamento do multiplexador, controle dos momentos de início e fim de conversão dos dados e transmissão dos dados para o transceptor).
- Empacotamento dos dados coletados e envio para transmissão.

As tarefas indicadas acima são executadas por um micro-controlador. A figura 3.11 mostra o diagrama elétrico do circuito de controle desenvolvido.

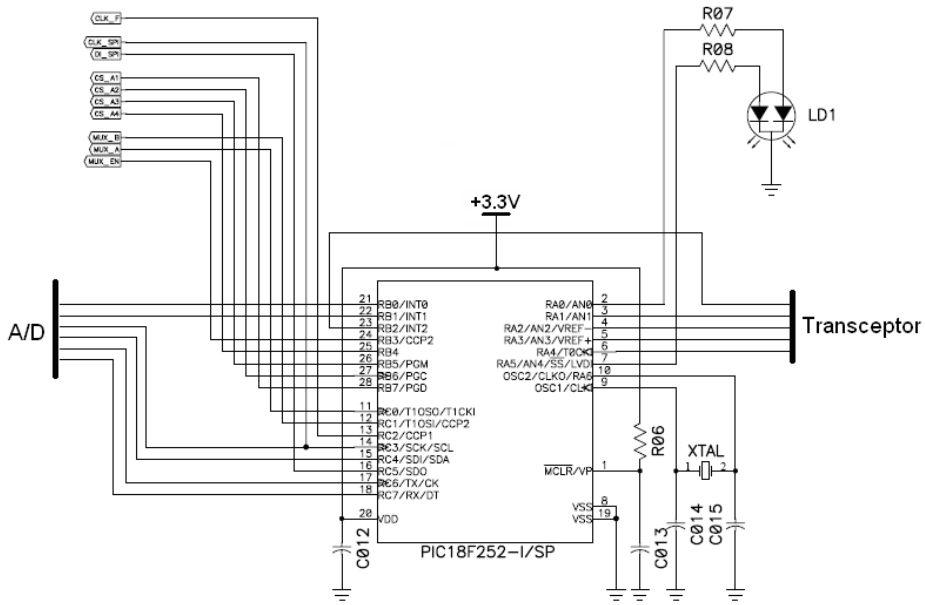


Figura 3.11: Diagrama elétrico do circuito de controle dos subsistemas

O micro-controlador escolhido para utilização nesse circuito possui alto desempenho, alimentação simples em 5V, baixo consumo (3,5mA típico), 32kb de memória de programação, 1536 bytes de memória de dados(*RAM*), 256bytes de memória *EEPROM*, barramento *SPI*, modulador *PWM* (para controle do *clock* dos filtros) e um *watchdog timer*.

3.7 Transceptores

Os módulos transceptores são responsáveis pela transmissão dos dados (sinais coletados e comandos), através de rádio frequência `(RF)`. Esses módulos devem possuir as seguintes características : comunicação duplex, vários canais (frequência de trabalho) selecionáveis (para evitar interferências e permitir o uso de mais de um aparelho em um mesmo ambiente), taxa de transmissão de dados mínima de `800Kbps` e um raio de comunicação mínimo de `50m` (sem barreiras). No projeto foram utilizados módulos transceptores que possuem as seguintes características:

- Comunicação Duplex — Trabalha como transmissor ou receptor sob o comando do micro-controlador.
- Taxa de transmissão de dados variável — A taxa de transmissão pode ser variada dinamicamente de 0 a 1Mbps.
- Canais selecionáveis — Possui 125 frequências de trabalho na faixa ISM (*Industrial, Scientific and Medical*) de 2,4GHz. A frequência de transmissão pode ser diferente da frequência de recepção. A seleção dos canais pode ser feita de forma dinâmica, permitindo a implementação de sistemas de comunicação com *frequency hopping*.
- Múltiplos receptores — O módulo possui dois receptores distintos, o que permite a recepção de dados provenientes de duas fontes diferentes.
- Potência de saída selecionável — A potência de transmissão de RF pode variar de -20dBm a 0dBm em degraus de 5dBm.
- Verificação dos dados transmitidos por CRC16 — Na transmissão, os dados podem ser automaticamente codificados com o código CRC16, permitindo na recepção a verificação da integridade dos dados.
- Endereço selecionável — O modulo é endereçável, permitindo a implementação de uma rede de transceptores.

A conexão do transceptor com o micro-controlador do circuito de condicionamento de sinais e conversão A/D se dá conforme o diagrama da figura 3.12.

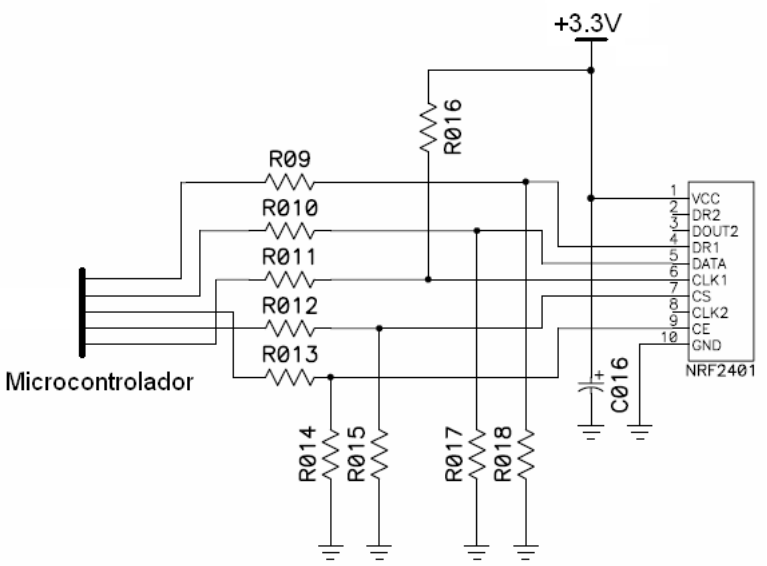


Figura 3.12: Conexão do transceptor ao circuito de condicionamento e conversão A/D

O diagrama de ligação do módulo transceptor utilizado no sistema de recepção dos sinais coletados é mostrado na figura 3.13.

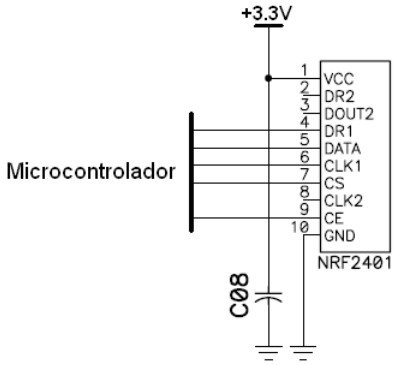


Figura 3.13: Conexão do transceptor ao circuito de recepção

Os transceptores são facilmente programados usando-se uma interface serial a 3 fios, onde a velocidade da comunicação é determinada pela velocidade do micro-processador. Esses transceptores trabalham com a modulação GFSK

(*Gaussian Frequency Shift Keying*), que é similar à modulação `FSK`, mas onde os dados antes de entrarem no modulador passam por um filtro `gaussiano`, de modo a reduzir a largura espectral dos mesmos. As características de taxa de transmissão, faixa de frequência e esquema de modulação são as mesmas utilizadas na tecnologia *Bluetooth*.

3.8 Controle do sistema de recepção

O circuito de controle do sistema de recepção tem a responsabilidade de receber os dados provenientes do sistema de aquisição de sinais e enviá-los ao módulo USB para comunicação com o computador principal. É responsável também por receber informações do computador principal e transmiti-las (via transceptores) ao módulo de aquisição. Estas informações podem ser instruções de configuração de parâmetros ou confirmação da execução de uma determinada ação requisitada. Essas tarefas são realizadas por um microcontrolador.

O diagrama elétrico do circuito de controle implementado é mostrado na figura 3.14.

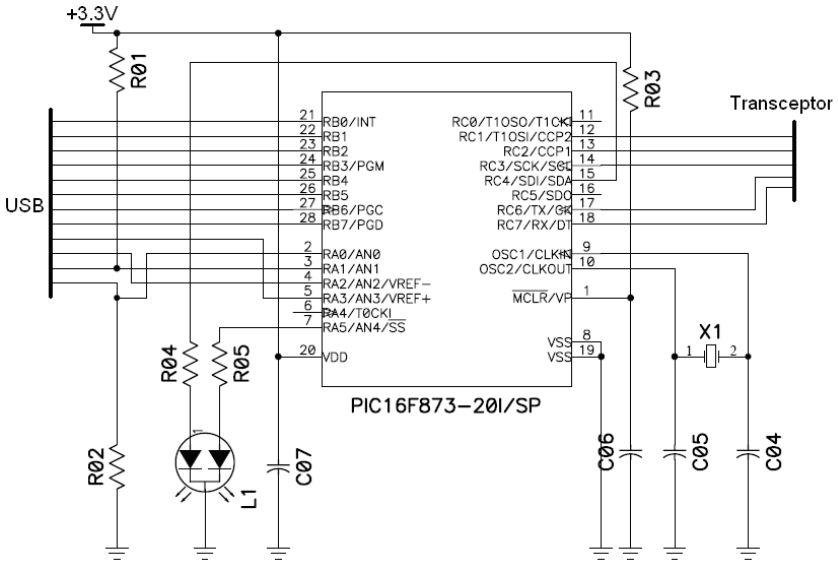


Figura 3.14: Diagrama elétrico do circuito de controle do sistema de recepção

3.9 Comunicação com o PC

O circuito de comunicação com o computador principal foi baseado em um módulo de comunicação USB. Esse módulo incorpora uma interface USB que é convertida em dois canais de I/O (serial e paralelo), que são configurados de forma independente. A taxa de transferência de dados pode chegar a 1Mbps no canal serial e 3Mbps no canal paralelo.

Todo o gerenciamento do protocolo USB é realizado pelo próprio módulo, portanto nenhuma programação específica de firmware USB é necessária. A tensão de alimentação do sistema de recepção é retirada do módulo, que por sua vez recebe a alimentação da interface USB. A conexão do módulo com o micro-controlador do sistema de recepção é mostrada na figura 3.15.

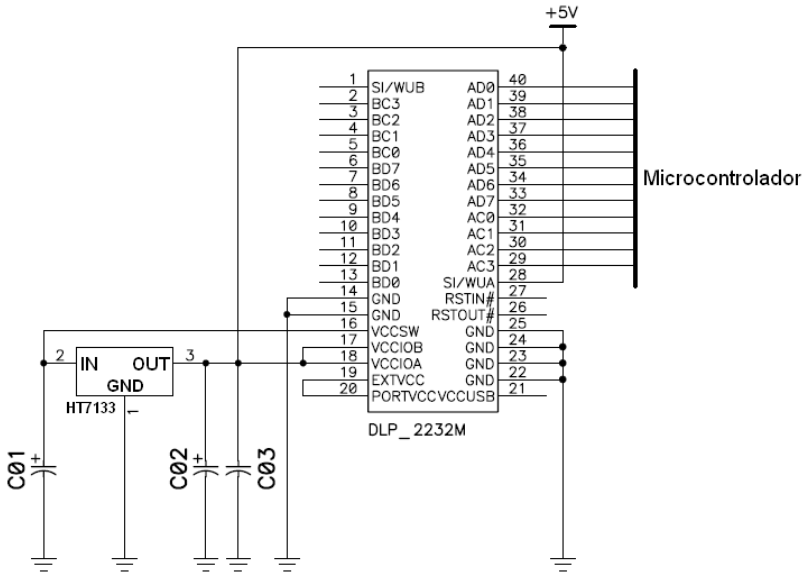


Figura 3.15: Diagrama elétrico de conexão do módulo USB

3.10 Alimentação do sistema

O sistema de aquisição (subsistemas de condicionamento de sinais e conversão A/D) é alimentado por uma bateria Ni-Cd. A conversão da tensão da bateria

nas tensões necessárias para o circuito é realizada através de reguladores de tensão e de um conversor DC/DC. O circuito de alimentação também proporciona a possibilidade de se recarregar a bateria sem a necessidade de desconectá-la do aparelho. O diagrama elétrico do circuito de alimentação é apresentado na figura 3.16.

A tensão de alimentação do sistema de recepção, como já explicado anteriormente, é retirada da interface USB do computador principal.

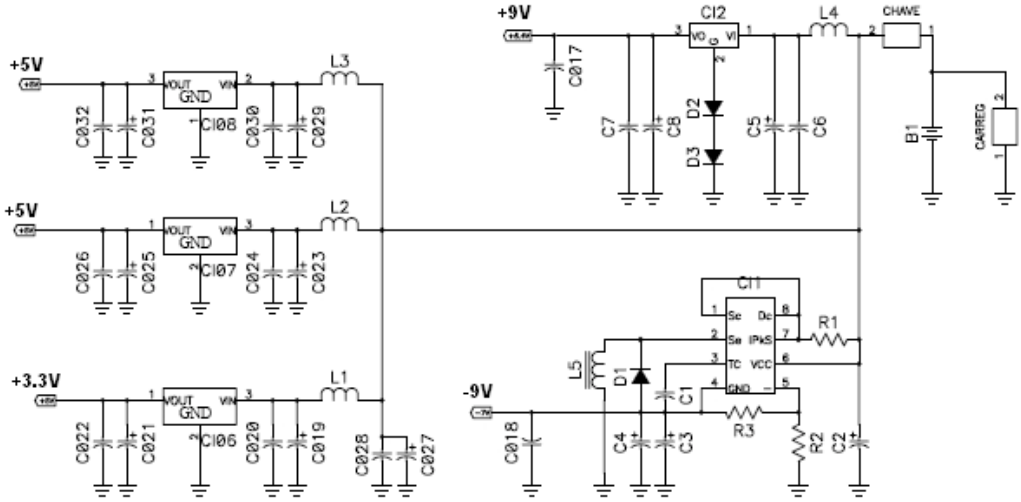


Figura 3.16: Diagrama elétrico do circuito de alimentação

3.11 Sistema simplificado

A fim de produzir um exemplo que ilustre as linguagens de domínio específico e algoritmos desenvolvidos no trabalho, tais como algoritmos de reconhecimento de padrões, processamento paralelo, calibração de *sensores*, linguagem de domínio específico para construção de interfaces gráficas, uma versão simplificada do hardware foi projetada e construída. A figura 3.17 mostra o esquema eletrônico deste sistema simplificado, que pode ser usado na aquisição de séries temporais de temperatura e capnogramas.

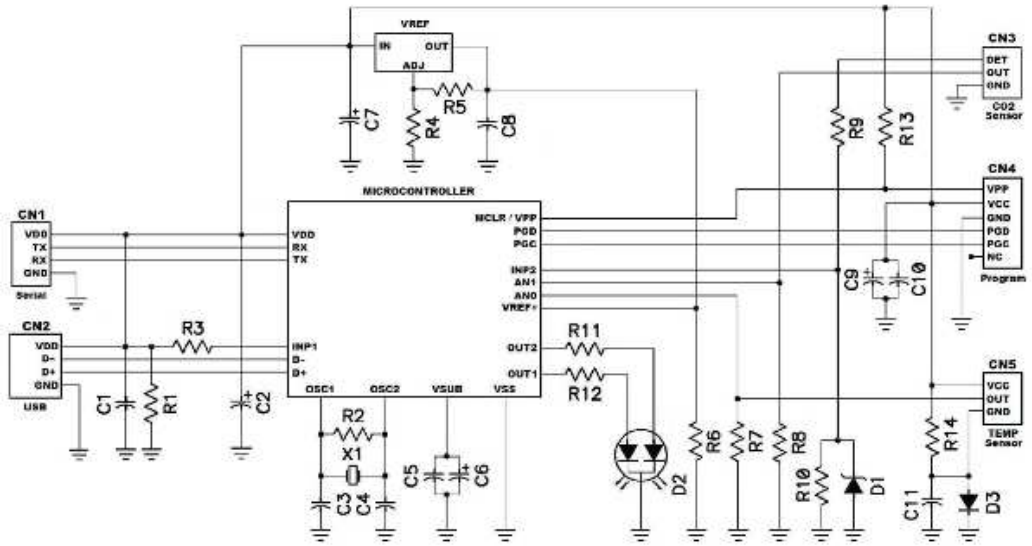


Figura 3.17: Sistema micro-controlado para termografia e capnografia

O autor do presente trabalho se interessou em medições de temperatura e CO_2 durante o estudo da contribuição de gases de efeito-estufa para a ilha de calor urbana, que é uma área metropolitana significativamente mais quente do que seu entorno.

Acredita-se que as ilhas de calor urbanas podem ter um impacto negativo sobre o aquecimento global e em mudanças climáticas. Após a concepção e construção de sensores para estudos de climatologia, ele foi contactado por médicos que estão estudando a correlação entre o aquecimento global e as doenças respiratórias. A partir dessas interações, veio a idéia de usar os mesmos sensores, para medir tanto a qualidade do ar quanto os padrões respiratórios de pacientes médicos com doenças relacionadas às mudanças climáticas.

A comunicação e controle de cada sensor é executada por um micro-controlador, como mostrado na figura 3.17. Um sensor tem muitos parâmetros que necessitam de ajuste, a fim de obter a leitura correta. Esses parâmetros são armazenados em uma memória EPROM, no interior do micro-controlador do sensor. Considerando-se que o número de vezes que se pode reescrever uma memória EPROM é relativamente pequeno, o micro-controlador envia os

parâmetros e dados para um programa de treinamento, que está hospedado em um computador principal. Vamos examinar o programa de comunicação que torna esse processo possível.

```
1  getTData outh n | n<1 = return ()
2  getTData outh n = do
3      tx ∈ sendMessage 1 "t"
4      let tp = filter (> ' ') tx
5      rawVal ∈ sendMessage 1 "x"
6      let raw = filter (> ' ') rawVal
7      let p= show (hex2dec raw)
8      threadDelay 500000
9      hPutStrLn outh (tp ++ "␣" ++ p)
10     getTData outh (n-1)
```

Na linha 3, o computador principal envia uma mensagem para os sensores pedindo a temperatura local; na linha 5, ele pede o valor bruto da temperatura. É claro que a calibração de um dado sensor requer meios de se medir um valor de referência da quantidade que precisa de ajuste. Este valor de referência pode ser obtido a partir de um sensor previamente calibrado. Naturalmente, o primeiro sensor deve ser ajustado manualmente. O programa grava um arquivo contendo os valores brutos de medições, e os valores calibrados dentro de um erro aceitável.

Os sensores devem aprender a produzir leituras corretas a partir de dados coletados. Isso pode ser feito através de computação evolutiva.

3.12 Computação evolutiva

Um algoritmo genético (AG) é um processo evolutivo de se estimar parâmetros. Programação Genética (PG) é uma metodologia de geração automática de programas. Tanto a Programação Genética quanto Algoritmos Genéticos foram inspirados na evolução biológica. Os programas e algoritmos evoluem através de processos estocásticos de mudança; alguns desses processos são explicados abaixo:

1. Criar a geração 0 pela aplicação aleatória de construtores de programas.
2. Atravessar várias gerações até que apareçam indivíduos competentes.
   (a) Executar cada um dos programas da população e calcular sua aptidão para resolver o problema apresentado.
   (b) Seleção — utilizar uma medida de aptidão para classificar a população.
   (c) Sobrevivência — fragmentos bem classificados sobrevivem na próxima geração.
   (d) Cruzamento *(crossover)* — produzir prole por re-combinação de fragmentos de programa.
   (e) Mutação — gerar programas mutantes através de busca heurística.

Abaixo, apresentamos um programa que realiza a aproximação de dados através de polinômios de [Bernstein], a fim de calibrar os sensores.

```
1  {- ghc gaCap.hs -O2 --make -}
2  {- Execute: gaCap.hs out.txt -}
3  import Data.Array.IO
4  import System.Random
5  import Data.Array
6  import Control.Parallel
7  import Data.Bits
8  import System
9  import Control.Monad.ST
10 import Data.Array.ST
11 import System.IO

13 type Sta s = STArray s Int Double
14 type SLD = [(Int, Double)]
15 type AD = Array Int Double
16 type POP= IO (IOArray Int (AD, Double))

18 (psz, thr, inf, sup, npar)= (30, 0.01, 0.0, 500.0, 2)
19 (alpha, order, ger) = (0.5, 3, 100)
```

```
21 main = do
22    let ind= listArray (0, order) [0.0 ..]
23    arr ∈ newArray (1,psz) (ind, 100.0) :: POP
24    args ∈ getArgs
25    case args of
26      [fn] → do
27         contents ∈ readFile fn
28         let dataset= rd (words contents) []
29         xs ∈ rnList (0.0, 1.0)
30         xs0 ∈ gen0 (ind,100.0) dataset arr psz xs
31         ind ∈ readArray arr 1
32         (bb, xs1) ∈ evolve dataset ind ger arr xs0
33         print bb
34      otherwise → putStrLn "e.g. usus: gbVec training.set"

36 rd [] acc= acc
37 rd [x] acc= acc
38 rd (x1:x2:xs) acc= rd  xs ((read x1, read x2):acc)

40 gen0 (b,fb)  s a i xs | i ≤ 1 = do
41       writeArray a i (b,fb)
42       return xs
43 gen0 bfb  s a i xs = do
44   let ind = listArray (0, order)
45               [ inf + x*(sup-inf) | x ∈ take (order + 1) xs]
46   writeArray a i (ind, fitness s ind)
47   gen0 bfb  s a (i-1) (drop (order + 1) xs)
```

Na linha 29, uma lista infinita de números aleatórios é construída para fornecer os elementos necessários para preencher o vetor da população. Cada par na população é uma tupla contendo um conjunto de parâmetros e sua aptidão.

O programa `gen0` percorre o vetor de população, inserindo, na população, indivíduos gerados aleatóriamente. A função de aptidão (linha 65) calcula o erro de estimação cometido por um dado indivíduo.

```
48 -- Triângulo de Pascal
49 pascal :: [[Double]]
50 pascal  = iterate (λrow → zipWith (+) ([0.0] ++ row)
51                                       (row ++ [0.0])) [1.0]
52 -- Números binomiais
53 bi     :: Int → Int → Double
54 bi n m = pascal !! n !! m

56 -- Polinômios de Bernstein
57 bernstein :: (Int,Int) → Double → Double
58 bernstein (i,n) t = (bi n i) * (t ** (fromIntegral i)) *
59         ((1.0 - t) ** (fromIntegral (n - i)))

61 rnList :: (Double, Double) → IO [Double]
62 rnList r=getStdGen≫=(λx→return(randomRs r x))

64 fitness s c = errSum (poly 0 0.0) s 0.0 where
65   poly i p x | i > order = p
66   poly i p x = poly (i+1) (p+(c!i) * bernstein (i,order) x) x
67   errSum p [] acc = acc
68   errSum p ((x,y):xs) acc = errSum p xs (acc+(y - p x)^2)
```

A função `cross` seleciona dois indivíduos da população, e força um cruzamento de seus conteúdos genéticos, produzindo uma prole.

```
70 map2 f p q= listArray (0, order) (loop 0) where
71   loop i | i > order = []
72   loop i =  f (p!i) (q!i) : loop (i+1)

74 cross (p1,p2) rnl | bounds p1 /= bounds p2 = (p1, rnl)
75 cross (p1, p2) (r:rnl)  = (offspring, rnl) where
76   offspring = map2 (λ x y → x + beta*(y-x)) p1 p2
77   beta = -alpha + r * (1.0 + 2.0 * alpha)
```

O programa `evolve s (b, fb) i pop xs` é um *loop* com cinco argumentos: o conjunto de treinamento, o melhor indivíduo encontrado até agora, o contador de gerações, a população, e uma lista infinita de números aleatórios. Ele procura dois pais potenciais na população, gera filhos, e insere-os na população se eles estão aptos o suficiente para resolver o problema proposto.

O programa pára quando encontra um indivíduo capaz de resolver o problema dentro de uma determinada precisão. Se um indivíduo apto não é encontrado após 100 gerações, `evolve` destrói a população inteira, e começa tudo novamente; o novo começo, também conhecido como *dilúvio*, é mostrado nas linhas 113, 114 e 115.

```
78  best i b pop xs | i > psz= return (b, xs)
79  best i b pop (r:xs) = do
80     (ix, ifit) ∈ readArray pop i
81     (bx, bfit) ∈ readArray pop b
82     if ifit < bfit && r < 0.2
83       then best (i+1) i pop xs
84       else best (i+1) b pop xs

86  findworse i i1 pop | i > psz= return i1
87  findworse i i1 pop = do
88     (_, fi) ∈ readArray pop i
89     (_, f1) ∈ readArray pop i1
90     if fi > f1 then findworse (i+1) i  pop else
91          findworse (i+1) i1 pop

93  mutt s (arr, fm) = runST $ do
94    starr ∈ thaw arr
95    grad ∈ loop (bounds arr) starr []
96    newarr ∈ freeze starr
97    let t= 0.02
98    let m2 = accum (λ c g → c − t∗g) newarr grad
99    return (m2, fitness s m2)
100  where
101    loop::(Int, Int)→Sta s→ SLD →ST s SLD
102    loop (i, n) arr acc | i > n= return $ reverse acc
103    loop (i, n) arr acc = do
104       old ∈ readArray arr i
105       let dx= 0.01
106       writeArray arr i (old+dx)
107       xx ∈ freeze arr
108       let f1 = fitness s xx; writeArray arr i old
109       let g  = (f1 − fm) / dx
110       loop (i+1, n) arr ((i, g):acc)
```

```
112 evolve s (b, fb) i pop xs | fb < thr = return ((b, fb), xs)
113 evolve s bfb i pop xs | i < 1 = do
114   xs0 ∈ gen0 bfb s pop psz xs
115   evolve  s bfb ger pop xs0
116 evolve  s b i pop (r:r1:r2:xs) = do
117   xs2 ∈ parents 10 s pop xs
118   (ib, xs5) ∈ best 1 2 pop xs2
119   bb ∈ readArray pop ib
120   evolve  s bb (i-1) pop xs5
```

É fato bem conhecido que, isoladamente, cruzamento pode produzir populações degeneradas, i.e., populações que não evoluem para a solução do problema em questão. Portanto, parte dos filhos deve sofrer mutação antes da inserção na população. Nas linhas de 78 a 110, pode-se ver a função de mutação, juntamente com duas outras funções: `best` e `findworse`. A função `best` encontra o indivíduo mais apto, i.e., o indivíduo que chega na solução com o menor erro. Na verdade, para evitar que o algoritmo seja capturado por ótimos locais, a função `best` pode trocar o individuo mais apto por outro não tão bem adaptado; a decisão de escolher um indivíduo sub-ótimo é realizada na linha 82, e é baseada no valor de um número aleatório. O filho é inserido na população se a função `findworse` pode encontrar um lugar ocupado por um indivíduo não adequado para o trabalho.

```
121 parents i s pop xs | i < 1 = return xs
122 parents i s pop (r1:r2:xs) = do
123     let i1 = 1+truncate (r1*(fromIntegral psz))
124     let i2 = 1+truncate (r2*(fromIntegral psz))
125     (t1, ft1) ∈ readArray pop i1
126     (t2, ft2) ∈ readArray pop i2
127     let (m1, xs1) = cross (t1, t2) xs
128     let (m2, xs2) = cross (t1, t2) xs1
129     let (im1,f1) = mutt s (m1, fitness s m1)
130     let (im2,f2) = mutt s (m2, fitness s m2)
131     iw1 ∈ findworse 1 2 pop
```

```
132     writeArray pop iw1 (im1, f1)
133     iw2 ∈ findworse 1 2 pop
134     writeArray pop iw2 (im2, f2)
135     parents (i − 1) s pop xs2
```

Em geral, sensores não produzem saída de resultados corretos. Portanto, a fim de calibrar um sensor, é preciso de uma nuvem de dados que contenha as medições do sensor, e os correspondentes valores corretos. O processo de calibração constrói uma função f :: Measurements →Values que minimiza

$$\sum_i (v_i - m_i)^2$$

onde cada ponto (m_i, v_i) pertence à nuvem de dados, m_i é uma medida, e v_i é o valor correspondente.

3.13 Programação Memética

Otimização refere-se a minimizar ou maximizar uma função escalar objetiva de valores reais, pela escolha de parâmetros dentro de um conjunto permitido. No presente caso, a função objetiva mede o erro de uma expansão polinomial que calibra um sensor a fim de obter medidas corretas a partir de dados brutos. Na figura 3.18, valores de referência são mostrados como pequenos quadrados preenchidos. A curva de calibração, encontrada por um sistema de programação genética, é dada pela linha contínua.

A curva de calibração é uma expansão em series que utiliza polinômios de Bernstein como funções de base. Um modelo similar foi usado em um estudo climatológico da movimentação zonal de plasma sobre Jicamarca, na região F, (vide [Fejer]). Nesse estudo, foi descrito resumidamente as propriedades básicas dos polinômios de Bernstein. Para pronta referência, o leitor encontrará a mesma descrição abaixo.

Os polinômios de [Bernstein] receberam esse nome em honra ao matemático ucraniano Sergei Bernstein que usou-os para provar o teorema de aproximação de Weierstrass. Eles podem representar funções monótonas seccional-

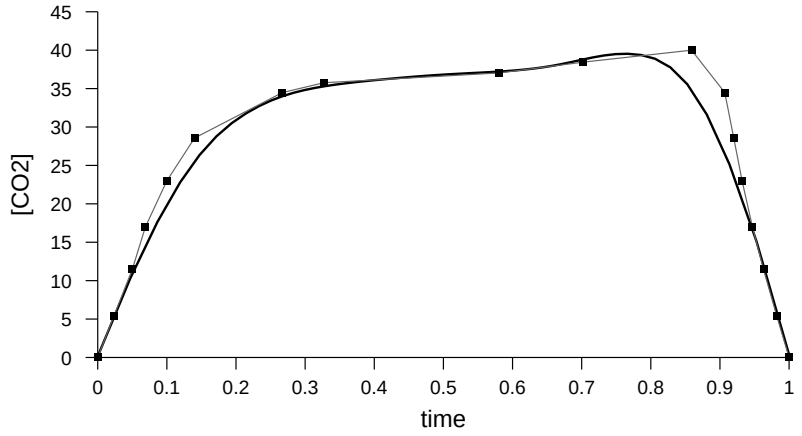


Figura 3.18: Capnograma normal e curva de calibração

mente suaves, que têm derivadas esquerda e direita em cada ponto. Essa representação não apresenta o fenômeno de Gibbs. O fenômeno de Gibbs é o comportamento peculiar da série de Fourier de uma função diferenciável periódica seccionalmente contínua em pontos onde ocorrem saltos de continuidade; perto do salto, a enésima soma parcial da série de Fourier apresenta um sobresalto que não desaparece com o aumento da freqüência, mas se aproxima de um limite finito.

Como [Fejer] diz, os polinômios de Bernstein fornecem aproximações globais de nuvens de pontos de dados em contraste com aproximações locais dadas por *splines* e outros métodos populares. Eles são particularmente úteis na modelagem de conjuntos de dados incompletos e ruidosos, que não aceitam bem aproximações locais.

O algoritmo de aprendizagem é baseado no cálculo e correção do erro cometido pela função de predição. O erro de um determinado conjunto de exemplos é calculado pela função `errSum`, que implementa a expressão abaixo, onde e é um exemplo, v_c é o valor previsto, e v é o valor determinado.

$$\sum_e (v_c(e) - v(e))^2$$

A função `mutt` atualiza os pesos pelo método do gradiente descendente de tal forma a reduzir o erro a um mínimo. Gradiente descendente é um algoritmo de otimização que encontra um mínimo local de uma função escolhendo passos proporcionais ao gradiente – com sinal trocado – da função em um dado ponto. Em cada passo do método de gradiente descendente, o peso é atualizado de acordo com a seguinte fórmula:

$$\omega_{n+1} = \omega_n - \gamma \nabla \mathbf{error}(\omega)$$

Nessa fórmula, se $\gamma > 0$ é um número suficientemente pequeno, $\mathbf{error}(\omega_{n+1}) < \mathbf{error}(\omega_n)$. Começando-se com ω_0, a sequência $\omega_0, \omega_1, \omega_2 \ldots$ converge para um mínimo. O mínimo será atingido quando o gradiente torna-se zero (ou próximo de zero, em situações práticas).

Se o método do gradiente descendente for o único método de otimização utilizado para encontrar a curva de calibração, corre-se o risco de captura por um mínimo local. O uso de programação genética impede que o gradiente descendente fique preso em mínimos locais. Por outro lado, o gradiente descendente acelera a convergência, que tornar-se-ia muito lenta em um sistema puramente evolucionário. Essa combinação de uma abordagem evolutiva com procedimentos de busca com melhoria local é chamada de algoritmos meméticos. A idéia vem de uma teoria de [Richard Dawkins], que acredita que a evolução não é exclusiva de sistemas biológicos, mas pode ser aplicada a qualquer sistema complexo, que apresente os princípios de herança, variação e seleção.

Capítulo 4

Lisp

Segundo [Richard Gabriel], Inteligência Artificial (AI) é uma espécie de mágica. Os mágicos de palco fazem um elefante desaparecer no ar, uma corda dançar ao som de flauta e outras coisas desse jaez. Os magos da Inteligência Artificial, por outro lado, produzem aplicações que resolvem difíceis problemas matemáticos, derrotam o campeão mundial de xadrez, ou controlam um robô que vai explorar a superfície de Marte.

Tanto mágicos de palco quanto magos da Inteligência Artificial mostram-se reticentes em revelar o segredo de seus truques. Contudo, cientistas como [Joshua Lederberg] (prêmio Nobel de medicina), Herbert Simon (prêmio Nobel de economia), [Douglas Hofstadter] e [Shiro Kawai] acham que conhecem o segredo da inteligência artificial: uma linguagem de programação chamada Lisp.

Não há muitos programadores Lisp. Apenas 1% dos engenheiros e cientistas de computação trabalham com Lisp. Há uma boa razão para a falta de popularidade da Lisp: a linguagem é hermética e misteriosa. Até mesmo uma simples expressão aritmética tem o aspecto de tecnologia alienígena quando expressa em Lisp. A propósito, comparar Lisp com tecnologia alienígena não foi idéia nossa, mas de um médico americano chamado [Barski]. Este Barski, que além de importante pesquisador, é um exímio desenhista e humorista, chegou mesmo a criar uma personagem alienígena que teria trazido Lisp para a terra.

Para entender porque Lisp encontra tantos praticantes entre cientistas e artistas, mas poucos entre programadores médios, vamos fazer uma rápida comparação entre um programa em Lisp e um programa em Clean. Por exemplo, o programa abaixo mostra como calcular fatorial em Clean:

```
module factorial
import StdEnv

fac 0 = 1
fac n = n*fac(n-1)

Start= fac 5
```

Lisp tem três dialetos em uso ativo: Common Lisp, Elisp e a Scheme. Neste capítulo, vamos descrever dois dialetos, apontando os pontos fortes de cada um. Inicialmente vamos descrever a Common Lisp, assim como o Maxima, um impressionante exemplo de inteligência artificial.

Em Lisp, não se usa operações infixas. Todo operador é prefixado. Eis como definir a função que calcula o fatorial de um número:

```
(defun fac(n)
   (if (<= n 1) 1
      (* n (fac (- n 1)))))
```

Como já dissemos antes, não há muitos magos da AI, nem programadores Lisp. Somente 1% dos profissionais usam Lisp. O problema é que este pequeno grupo é responsável diretamente ou indiretamente por 95% dos sistemas computacionais existentes, segundo [Drew McDermott]. Dois exemplos tornarão claro o que pretendemos dizer com isso.

Emacs foi o primeiro editor de texto em tela inteira. Esta famosa ferramenta foi escrita em Lisp por Richard Matthew Stallman (aka RMS). Programadores Lisp também inventaram o mouse, o monitor de computador e tudo mais que precisavam para fazer o Emacs funcionar. Naturalmente, depois do Emacs, outras pessoas foram capazes de clonar editores em `C`, `Pascal` ou `Clean`.

Considerando a importância do Emacs na história da computação, assim que adquirirmos noções básicas de Lisp, examinaremos detalhadamente o funcionameno desse editor de textos.

Um dos primeiros programas de inteligência artificial foi o Maxima, que até hoje permanece um dos mais impressionantes sistemas que a tecnologia de computadores criou. Maxima consegue resolver difíceis problemas de engenharia e matemática com um desempenho superior ao da maioria dos profissionais humanos. Assim, Maxima é um dos melhores exemplos do que Lisp pode fazer.

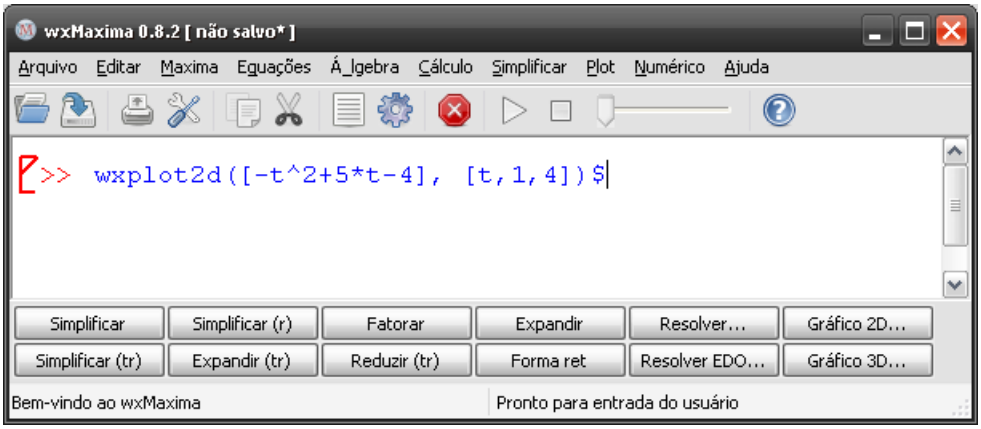
4.1 Maxima

Maxima é um sistema com inteligência artificial, construído sobre a linguagem Lisp, bem testado e que possui uma robusta base de conhecimentos sobre matemática. Isto dito, vamos fazer uma breve introdução ao Maxima.

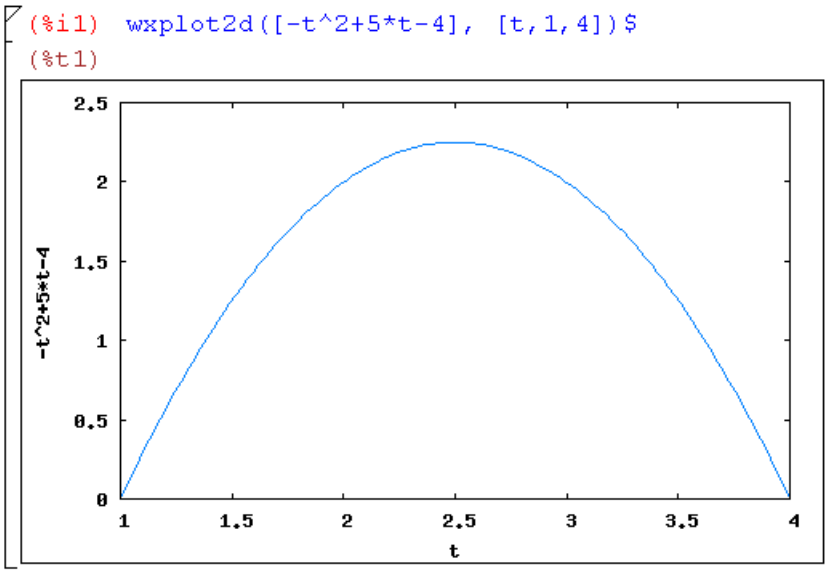
Figura 4.1: Janela do wxmaxima

Maxima é um dos melhores sistemas existentes para processamento de matemática. Além do mais, é gratuito. Procure na internete e instale uma cópia do Maxima em seu computador. Assim que você executar o ícone do **wxmaxima**, uma janela como a mostrada na figura 4.1 aparecerá na tela. No topo do painel da janela, há um traço reto. Este traço é o cursor horizontal. Se você apertar a tecla `F7` ou clicar com o mouse sobre a engrenagem

que aparece na barra de ferramentas, uma célula de cálculo surgirá no local do cursor horizontal. Digite o comando que traça o gráfico da função $f(t) = -t^2 + 5t - 4$, conforme mostrado na figura abaixo.



Para executar o comando, aperte *shift/Enter* (mantenha a tecla *shift* abaixada e aperte a tecla *Enter*). O Maxima apresentar-lhe-á o gráfico abaixo.



Além de traçar gráficos, Maxima é capaz de realizar praticamente qualquer operação matemática que possa ser útil a um engenheiro. Com o Maxima podemos achar a solução de problemas complexos relacionados a cálculo, geometria analítica ou álgebra linear. Inicialmente, vejamos como Maxima

pode ser utilizado para realizar integrações, diferenciar funções e calcular limites.

```
(%i1) integrate(x*sin(x), x);
```

(%o1) $$sin\,(x) - x\,cos\,(x)$$

```
(%i2) integrate(1/(1+x^3), x);
```

(%o2) $$-\frac{log\,(x^2 - x + 1)}{6} + \frac{atan\left(\frac{2\,x-1}{\sqrt{3}}\right)}{\sqrt{3}} + \frac{log\,(x+1)}{3}$$

```
(%i3) diff(sin(x)/cos(x), x);
```

(%o3) $$\frac{sin\,(x)^2}{cos\,(x)^2} + 1$$

```
(%i4) trigsimp(%);
```

(%o4) $$\frac{1}{cos\,(x)^2}$$

```
(%i5) limit( (2*x+1)/(3*x+2),  x,inf );
```

(%o5) $$\frac{2}{3}$$

```
(%i6) limit( sin(3*x)/x,  x,0);
```

(%o6) $$3$$

Para se obter a expressão de entrada na notação matemática, basta prefixar um apóstrofo à linha de comando do Maxima. No exemplo abaixo, mostramos o comando `integrate(1/(1+x^3), x)` tanto prefixado com apóstrofo, como sem o prefixo.

```
(%i3) 'integrate(1/(1+x^3), x);
```

(%o3) $$\int \frac{1}{x^3+1} dx$$

```
(%i4) integrate(1/(1+x^3), x);
```

(%o4) $$-\frac{log\left(x^2-x+1\right)}{6}+\frac{atan\left(\frac{2\,x-1}{\sqrt{3}}\right)}{\sqrt{3}}+\frac{log\left(x+1\right)}{3}$$

Pode-se também calcular integrais definidas com Maxima, conforme mostrado no exemplo abaixo. Observe também que, aplicada a várias expressões entre parênteses, o apóstrofo impede a avaliação de todas elas.

```
(%i4) x0:5$
      x1:7$
      integrate(x^2, x, x0, x1);
```

(%o6) $$\frac{218}{3}$$

```
(%i8) 'integrate(x^2, x, x0, x1) = integrate(x^2, x, x0, x1);
```

(%o8) $$\int_5^7 x^2 dx = \frac{218}{3}$$

```
(%i9)  '(sqrt(aa) + bb);
```

(%o9) $$bb+\sqrt{aa}$$

Maxima possui várias funções para expandir e fatorar expressões. Para avaliar expressões, prefixamos dois apóstrofos a elas. A entrada `id:expr` associa um identificador `id` a uma expressão `expr`. Maxima associa automaticamente identificadores de entrada e saída com as expressões calculadas ou digitadas pelo usuário; os identificadores de entrada e saída são apresentados entre parênteses no início da linha. Conforme mostrado abaixo, a última entrada é associada ao identificador `%`. Exemplos:

```
(%i1) expand((x+y)^4);
```

(%o1) $$y^4 + 4\,x\,y^3 + 6\,x^2\,y^2 + 4\,x^3\,y + x^4$$

```
(%i2) factor(%o1);
```

(%o2) $$(y + x)^4$$

```
(%i3) expand(%);
```

(%o3) $$y^4 + 4\,x\,y^3 + 6\,x^2\,y^2 + 4\,x^3\,y + x^4$$

```
(%i4) xx:'integrate(x*sin(x), x)$
```

```
(%i5) xx;
```

(%o5) $$\int x\,sin\,(x)\,dx$$

```
(%i6) ''xx;
```

(%o6) $$\int x\,sin\,(x)\,dx$$

```
(%i7) ''''xx;
```

$$(\%o7) \qquad sin\,(x) - x\,cos\,(x)$$

```
(%i8) [xx, ''''xx];
```

$$(\%o8) \qquad [\int x\,sin\,(x)\,dx, sin\,(x) - x\,cos\,(x)]$$

Em Maxima, podemos definir novas funções computacionais. Uma função computacional é aquela que, dado um argumento, associa um único valor a ele. Fornecemos abaixo exemplos de como definir funções.

```
(%i1) fn_1(x) := aa - bb*x;
```

$$(\%o1) \qquad fn\_1\,(x) := aa - bb\,x$$

```
(%i2) fn_1(10);
```

$$(\%o2) \qquad aa - 10\,bb$$

```
(%i1) fat(n) := if n=0 then 1 else n*fat(n-1)$
```

```
(%i2) fat(40);
```

$$(\%o2) \qquad 815915283247897734345611269596115894272000000000$$

A linguagem Lisp teve origem em um sistema formal matemático chamado Cálculo Lambda. Este sistema tem por finalidade definir e avaliar funções. Antes de explicar o que é cálculo lambda, precisamos saber o que é sistema formal. Um sistema formal consiste em uma linguagem recursiva (ver definição de linguagem na página 1) e um conjunto de regras de inferência.

A linguagem é recursiva se podemos determinar a validade de uma fórmula com um número finito de passos. No caso do cálculo lambda, as regras de gramática poderiam ser as seguintes (a sintaxe de uma linguagem é arbitrária; isto significa que o cálculo lambda pode ter mais de uma gramática):

$$
\begin{array}{lll}
< \text{expr} > & ::= & < \text{id} > \qquad\qquad x, y, z, u, v \\
 & & < \text{constant} > \qquad\qquad \pi \\
 & & \lambda([x_1, x_2 \ldots], < \text{expr} >) \qquad\qquad \lambda([x], \sin(x)) \\
 & & < \text{expr} > (< \text{e} >_1, < \text{e} >_2 \ldots) \qquad\qquad \sin(x)
\end{array}
$$

Podemos ver abaixo como expressões lambda são usadas para definir funções. Na linha `(%i4)` definimos uma expressão capaz de calcular a norma de um vetor. Na linha `(%i6)`, mostramos como utilizar `norma`.

```
(%i1) lambda([x,y], x*x+y*y);
(%o1)                      lambda([x, y], x x + y y)
(%i2) %o1(3,4);
(%o2)                                 25
(%i3) lambda([x,y], x*x+y*y)(3,4);
(%o3)                                 25
(%i4) norma:lambda([x,y], x*x+y*y);
(%o4)                      lambda([x, y], x x + y y)
(%i5) norma;
(%o5)                      lambda([x, y], x x + y y)
(%i6) norma(3,4);
(%o6)                                 25
(%i7) fn(x,y) := x*x+y*y;
(%o7)                        fn(x, y) := x x + y y
(%i8) fn(3,4);
(%o8)                                 25
```

Agora vamos examinar algumas expressões com matrizes. Inicialmente, vamos criar uma matriz e armazená-la na variável `mm`:

```
(%i1) mm:matrix([1,2,5],[2,3,7],[2,4,7])$
```

Para colocar `mm` no formato LaTeX e imprimi-lo em textos como esse, usamos o comando `tex(mm)`, conforme mostrado abaixo.

```
(%i1) mm:matrix([1,2,5],[2,3,7],[2,4,7])$
(%i2) tex(mm);
```

$$\begin{pmatrix} 1 & 2 & 5 \\ 2 & 3 & 7 \\ 2 & 4 & 7 \end{pmatrix}$$

```
(%i3) tex(mm . mm^^-1);
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Em linguagens de programação convencionais, temos comandos de bloco como `if-then-else` e `while-do`; estes comandos também são encontrados no Maxima. Quando Gauß era criança, seu professor pediu-lhe que somasse todos os números inteiros entre 1 e 100. Gauß observou que a soma do primeiro e do último número era 101; da mesma forma, a soma do segundo e do penúltimo era 101 também; resumindo, era possível organizar os números em 50 pares de números cuja soma era 101; o jovem Gauß chegou à conclusão de que o resultado é dado por $50 \times 101 = 5050$. Eis um programa para realizar tal soma sem utilizar o subterfúgio de Gauß:

```
(%i4) block([S], S:0, for i:1 while i<=100 do S:S+i, return(S));
(%o4)                                5050
```

É possível modificar a precisão de um número em ponto flutuante. Isto, porém, só é implementado para os chamados números *big float*. Um número

big float é representado na notação científica, onde a potência de dez é introduzida pela letra b; exemplos de números do tipo *big float*: $1.0b0 = 1.0 \times 10^0$, $23.0b3 = 23.0 \times 10^3$, $3b8 = 3 \times 10^8$, etc. Mostramos abaixo como alterar a precisão dos números do tipo *big float*.

```
(%i1) fpprec:40$
(%i2) 4b0 * atan(1b0);
(%o2)             3.1415926535897932384626433832795028841b0
(%i4) 1b0/3;
(%o4)            3.3333333333333333333333333333333333333b-1
(%i5) pi;
(%o5)                                  pi
(%i6) fpprec:120$
(%i7) pi:4*atan(1b0);
(%o7) 3.14159265358979323846264338327950288419716939937510\
582097494459230781640628620899862803482534211706798214808\
51328230665b0
```

A maior parte das entidades da Lisp em geral e do Maxima em particular são expressões. Uma expressão é qualquer fórmula que representa um valor. Fornecemos abaixo alguns exemplos de expressões em Maxima. Note que até `if-then-else` e laços são expressões.

```
(%i1) x:3$
(%i2) (x:x+1, x:x^2);
(%o2)                          16
(%i3) (if x>17 then 2 else 4);
(%o3)                           4
(%i4) (if x>17 then x:2 else y:4, x+y);
(%o4)                          20
(%i5) (x:1, for i from 1 thru 5 do x:x*i);
(%o5)                        done
(%i6) x;
(%o6)                          120
(%i7) (x:1, for i from 1 thru 10 step 2 do x:x*i);
(%o7)                        done
(%i8) x;
(%o8)                          945
```

Uma expressão complexa é especificada em Maxima somando-se a parte real com a parte imaginária multiplicada por `%i`.

```
(%i1) solve(x^2-4*x+13=0, [x]);
(%o1)          [x = 2 - 3 %i, x = 3 %i + 2]
```

Fatorar inteiros é de grande utilidade em criptoanálise. Maxima pode fatorar inteiros facilmente.

```
(%i1) factor (12345678);
```

(%o1) $$2 \times 3^2 \times 47 \times 14593$$

Maxima também pode simplificar somatórias. Abaixo, no primeiro exemplo, apenas escrevemos a somatória no formato LaTeX. Em seguida, calculamos o limite da somatória.

```
(%i1) tex(sum(1/2^k, k, 1, inf));
```

$$\sum_{k=1}^{\infty} \frac{1}{2^k}$$

```
(%i2) sum(1/2^k, k, 1, inf), simpsum;
(%o2)                          1
(%i3) sum(i^2, i, 1, 7);
(%o3)                         140
```

4.2 Expressões S

Maxima utiliza expressões aritméticas não muito diferentes de linguagens como Clean ou C. Programadores Lisp, entretanto, preferem representar expressões com todas as operações prefixadas. Por exemplo, a expressão $2\times3\times5+5\times4\times6+8$ é expressa assim: `(+ (* 2 3 5) (* 5 4 6) 8)`. Esta maneira de apresentar expressões é chamada notação prefixa de Cambridge,

sendo Cambridge a pequena cidade do estado americano de Massachusetts onde fica o MIT, instituto em que a linguagem Lisp foi inventada.

Lisp interage com o programador através do laço `read-eval-print` (lê, calcula o valor e imprime). No terminal, aparece um sinal de prontidão (um asterisco), indicando que Lisp está esperando uma expressão.

```
* |
```

Suponhamos que desejemos fazer uma soma: $1 + 2 + 3 + 4 + 5$. Neste caso, digitamos a expressão `(+ 1 2 3 4 5)` e apertamos a tecla `Enter`. Lisp apresenta o valor da expressão.

```
* (+ 1 2 3 4 5)

15
* |
```

Toda expressão Lisp tem a forma ($< op >$ a_1 a_2 $a_3 \ldots$), onde $< op >$ é uma operação e a_1, a_2, $a_3 \ldots$ são os argumentos. No exemplo dado, a operação é a soma ($+$) e os argumentos são os números 1, 2, 3, 4 e 5. O argumento de uma expressão pode ser outra expressão. Por exemplo, se quisermos calcular $2 \times 3 + 4 \times 5$, digitamos `(+ (* 2 3) (* 4 5))`.

```
* (+ (* 2 3) (* 4 5))

26
* |
```

Em Lisp, denomina-se uma expressão S qualquer lista de elementos entre parênteses, em que o primeiro elemento é uma operação e os outros elementos são os argumentos requeridos pela operação. Normalmente, Lisp avalia todas as expressões S que recebe, aplicando a operação aos argumentos. Para suspender a avaliação, devemos prefixar a lista com um apóstrofo.

```
* '(+ (* 2 3) (* 4 5))

(+ (* 2 3) (* 4 5))
* |
```

Como toda linguagem de programação, Common Lisp permite definir funções, ou seja, criar novas operações. Em Lisp, funções são definidas com

o auxílio do macro `defun`, que tem, no mínimo, três argumentos; o primeiro argumento é o nome da função que está sendo definida; o segundo argumento são os parâmetros formais da função; finalmente, temos expressões que permitem calcular o valor da função. O esqueleto desse macro é mostrado abaixo.

```
(defun f_name (arg1 arg2 arg3)
  (op1 arg1 arg2)
  (op2 arg1 arg2)
  ...
  (opn arg1 arg2 arg3) )
```

Um exemplo concreto é dado pela função que calcula as raízes de uma equação do segundo grau.

```
1 (defun quadratic-roots(a b c)
2   (let* ( (delta (- (* b b) (* 4 a c)))
3           (d (sqrt delta))
4           (2a (* 2 a)))
5     (values (/ (+ (- b) d) 2a)
6             (/ (- (- b) d) 2a))  ))
```

Figura 4.2: Definindo uma função: qroots.lisp

Na figura 4.2, utilizamos a forma `LET*` para introduzir variáveis locais. Apresentamos abaixo a sintaxe do `LET*`. Podemos ver que esta estrutura tem duas partes: (1) Uma lista de variáveis locais com seus valores e (2) uma sequência de comandos que devem ser executados com as variáveis.

Cada variável local do `LET*` é introduzida por uma lista `(<var> <valor>)`. No exemplo abaixo, temos as seguintes variáveis:

- `(delta (- (* b b) (* 4 a c)))` — Δ em função de a, b, c.
- `(d (sqrt delta))` — $\sqrt{\Delta}$
- `(2a (* 2 a))` — $2 \times a$

Existem duas formas que introduzem variáveis locais, o `LET` e o `LET*` (*let* estrela). No caso do `LET*`, podemos utilizar uma variável anterior para calcular um valor. No exemplo abaixo, utilizamos `delta` para calcular o valor de `d`. No caso do `LET` (sem estrela) isso não é permitido.

```
(let* (   (delta (- (* b b) (* 4 a c)))
          (d (sqrt delta))
          (2a (* 2 a)))                    ;;Variáveis locais

   ;; Valores produzidos pelo let*
   (values (/ (+ (- b) d) 2a)
             (/ (- (- b) d) 2a))  ))
);close let
```

Depois da lista de variáveis, está o corpo do `LET`, que contém uma sequência de expressões. A última expressão é o valor do `LET`.

Voltemos ao programa da figura 4.2, página 80. A função `quadratic-roots` produz dois valores, que são as raízes da equação do segundo grau. Esses dois valores são empacotados pela forma `(values val-1 val-2)`. Vamos ver como utilizar esses valores. Se desejarmos usar apenas um dos valores, basta inserir a chamada da função no ponto de utilização. Suponhamos que a função esteja armazenada no arquivo `qroots.lisp`. Neste caso, temos:

```
* (load "C:/sbcl/clisp/qroot.lisp")

T
* (quadratic-roots 2 3 -35)

3.5
-5.0

* (multiple-value-call #'+ (quadratic-roots 2 3 -35))

-1.5
* (multiple-value-bind (x1 x2)
     (quadratic-roots 2 3 -35) (list x1 x2))

(3.5 -5.0)
*
```

No exemplo acima, a forma `multiple-value-call` aplica a função `#'+` aos dois valores produzidos pela função `quadratic-roots`, somando-os. A forma `multiple-value-bind` associa as variáveis `x1` e `x2` aos valores produzidos por `quadratic-roots`.

Os exemplos que acabamos de ver processam apenas números. Entretanto, Lisp torna-se realmente poderoso quando trabalha com expressões simbólicas. Uma expressão simbólica pode ser:

- **Atom** é uma expressão simbólica que não pode ser analisada, ou seja, que não pode ser quebrada em outras mais simples. Átomos não podem ser construídos a partir de outras expressões mais simples. As expressões atômicas mais úteis são os números e os símbolos.

- **Lista** é uma expressão simbólica que contém duas partes, o primeiro elemento (também chamado `car` da lista) e o que resta da lista quando se remove o primeiro elemento; o resto da lista depois da remoção do primeiro elemento também é chamado de `cdr`.

Listas podem ser, informalmente, consideradas sequências ordenadas de elementos entre parênteses. O termo *sequências ordenadas* significa que a ordem importa; assim, a lista `(a b c d)` é diferente de `(b a c d)`.

```
* (defparameter *vogais* '(a e i o u))

*VOGAIS*
* *vogais*

(A E I O U)
* (car *vogais*)

A
* (cdr *vogais*)

(E I O U)
* (car (cdr *vogais*))

E
```

Listas podem ser analisadas e construídas. Analisar uma lista é quebrá-la em componentes, ou seja, obter o `car` e `cdr`. As funções que analisam listas são chamadas *seletores*. Mostramos acima a lista de vogais e usamos os *seletores* `car` e `cdr` para acessar os diferentes elementos desta lista.

Os programas em Lisp são listas, ou seja, sequências de elementos entre parênteses. O primeiro elemento de uma expressão que faz parte de um programa é uma operação; o que resta da expressão com a retirada do primeiro elemento são os argumentos. Conforme vimos, para que Lisp não considere uma expressão como parte de um programa é preciso precedê-la de um apóstrofo, o qual é denominado *quote*. Quando definimos o parâmetro `*vogais*`, na página 82, tivemos o cuidado de prefixar *quote* à lista de vogais; se não tivéssemos feito isso, Lisp consideraria o átomo `a` como sendo uma operação e os átomos `(e i o u)` como sendo parâmetros.

Os seletores `car` e `cdr` permitem desmontar uma lista. Para construir listas, usamos `(cons x xs)`, onde o primeiro argumento é o `car` da nova lista, e o segundo argumento é o `cdr`.

```
* (defvar *xs*)

*XS*
* (setf *xs* () )

NIL
* (setf *xs* (cons 2 *xs*))

(2)
* (setf *xs* (cons 4 *xs*))

(4 2)
* (setf *xs* (cons 'a *xs*))

(A 4 2)
*
```

Cálculo lambda. Podemos utilizar a sintaxe do Lisp no cálculo lambda. Nesse caso, as expressões simbólicas serão expressões lambda:

$$
\begin{array}{lll}
< \text{expr} > & ::= & < \text{id} > \qquad\qquad\qquad\qquad\qquad\qquad x, y, z, u, v \\
 & & < \text{constant} > \qquad\qquad\qquad\qquad\qquad\qquad \pi \\
 & & (\lambda(x_1 \;\; x_2 \ldots) < \text{expr} >) \qquad\qquad (\lambda(x)(sin \;\; x)) \\
 & & (< \text{expr} > \;\; < \text{e} >_1 \;\; < \text{e} >_2 \ldots) \qquad\qquad (\text{sin} \;\; x)
\end{array}
$$

Vamos ver alguns exemplos de processamento de lista. Inicialmente, escreveremos programas que percorrem listas aplicando `cdr` repetidamente.

Paradigma 1. Escreva um programa para somar os elementos de uma lista, mas apenas se eles forem números. Use a forma `cond`, que tem a seguinte estrutura:

```
(defun soma(xs)
    (Cond  ( (null xs) ;; Condição
             ;; A soma é zero:
                  0) ; Fim de lista vazia
                                                  } ;; Lista está vazia

           ( (numberp (car xs)) ;; Condição
             ;; (car xs)+(soma(cdr xs)):
               (+ (car xs) (soma (cdr xs)))
           ) ; Fim de (car xs) é número
                                                  } ;; (car xs) é número...

           ( T ;; Condição
             ;; Soma de xs= soma (cdr xs)
                  (soma (cdr xs))
           ))) ; Fim de outro caso
                                                  } ;; Outro caso
```

Para entender esse programa, façamos a análise de cada cláusula do `cond`. Devemos analisar três casos: 1 – xs= (); 2 – `(car xs)` é número; 3 – `(car xs)` não é número.

- `xs= ()`. Neste caso, a soma é zero. A cláusula que reconhece esta situação e produz a resposta desejada é

```
(  (null xs)   ;; Condição: xs=()?
      0        ;; Se a resposta for sim, soma é 0.
)
```

- O primeiro elemento de `xs` é número. Suponha, por exemplo, que `xs=(3 4 5 6)`. Neste caso, $\sum$ `xs`= `(car xs)`+$\sum$`(cdr xs)`, ou seja, $\sum$`(3 4 5 6)`= `3`+$\sum$`(4 5 6)`.

- O primeiro elemento de `xs` não é numérico: Abandone o elemento não numérico. Seja `xs=(a 3 4 5)`. $\sum$ `xs`= $\sum$`(cdr xs)`, ou seja, $\sum$`(a 3 4 5)`= $\sum$`(3 4 5)`.

Paradigma 2. Escreva um programa para somar os elementos numéricos de uma lista, mas utilize um parâmetro `&optional` para acumular resultados.

```
;; File: "downlist.lisp"
(defun  somar(xs &optional (S 0) )
    (cond ( (null xs)    ;; Se a lista estiver vazia
              S          ;; a resposta está em S
          )
          ( (numberp (car xs)) ;; Se elemento for número,
            (somar (cdr xs)      ;; retire-o da lista
                (+ (car xs) S) ) ;; e some-o ao acumulador
          )
          ( T                ;; Em qualquer outro caso,
             (somar (cdr xs))  ;; descarte o primeiro elemento
)   )     )
```

```
============ Testes ====================
* (load "downlist.lisp")

T
* (somar '(3 4 5 6))

18
* (trace somar)

(SOMAR)
* (somar '(3 4 5))
  0: (SOMAR (3 4 5))
    1: (SOMAR (4 5) 3)
      2: (SOMAR (5) 7)
        3: (SOMAR NIL 12)
        3: SOMAR returned 12
      2: SOMAR returned 12
    1: SOMAR returned 12
  0: SOMAR returned 12
12
```

O parâmetro `&optional (S 0)` recebe um valor inicial `0`. A medida que descemos pela lista, os elementos numéricos são retirados de `xs` e somados no acumulador `S`.

A forma `(cond ( (null xs)...) ((numberp...)...) (T...))` tem três casos, que descrevemos abaixo:

1. `(null xs)` — Lista vazia: a repetição terminou e parâmetro `S` contém a soma.

2. `(numberp (car xs))` — Elemento é número: retire-o da lista com `(cdr xs)` e some-o ao parâmetro `S` com `(+ (car xs) S)`.

3. `T` — Em qualquer outro caso, descarte o primeiro elemento de `xs` com `(cdr xs)`.

Paradigma 3. Utilize dois parâmetros opcionais para encontrar a média de uma lista de números. Imprima um *warning* para elementos não numéricos.

```
(defun media(xs &optional (S 0) (N 0))
   (cond ( (and (null xs) (= N 0)) 0)
         ( (null xs) ;; Iteração terminou
           (/ S N))
         ( (numberp (car xs))
           (media (cdr xs)
                  (+ (car xs) S) (+ N 1))  )
         ( T (format T "~:r element is not a number" (+ N 1))
             (terpri)
             (media (cdr xs)))  ))


(defun media-de-arquivo(fileName)
   (with-open-file (ifile (pathname fileName)
                       :direction :input
                       :if-does-not-exist Nil)
       (media (read ifile))
   )
)

====== Testes =================
* (load "downlist.lisp")

T
* (media '(3 4 rosa 5 6))
third element is not a number
11/2
* (media '(3 4 rosa 5 6.5))
third element is not a number
5.75
```

Paradigma 4. Encontrar o último cdr de uma lista. Isto significa que você deve encontrar uma lista contendo o último elemento de outra lista. Por exemplo, no caso da lista `(3 4 5 6)`, o último cdr é `(6)`.

```
(defun my-last (xs)
     (cond ( (not (consp xs)) '())
           ( (null xs) xs)
           ( (null (cdr xs)) xs)
           (T (my-last (cdr xs)) ) )))

========= Testes ===============
* (my-last '(3 4 5))

(5)
```

Paradigma 5. Encontrar o penúltimo cdr de uma lista.

```
(defun penultimo (xs)
     (cond ( (not (consp xs)) '())
           ( (null xs) xs)
           ( (null (cdr xs)) xs)
           ( (null (cddr xs)) xs)
           (T (penultimo (cdr xs)) ) )))
```

Paradigma 6. Encontrar o enésimo elemento de uma lista.

```
(defun enesimo (s k)
      (cond (  (null s) s)
            (  (<= k 1) (car s))
            (T (enesimo (cdr s) (- k 1))  )))

========= Testes ==========
* (enesimo '(a b c d) 1)

A
* (enesimo '(a b c d) 2)

B
```

Paradigma 7. Encontrar o número de elementos de uma lista.

```
(defun len (s &optional (i 0))
      (cond (  (not (consp s)) i)
            (T (len (cdr s) (+ i 1))  )))

 ======== Testes ==============
 * (load "D:/downlist.lisp")

T
 * (trace len)

(LEN)
* (len '(a b c))
  0: (LEN (A B C))
    1: (LEN (B C) 1)
      2: (LEN (C) 2)
        3: (LEN NIL 3)
        3: LEN returned 3
      2: LEN returned 3
    1: LEN returned 3
  0: LEN returned 3
3
*
```

Paradigma 8. Inverter uma lista.

```
(defun rev (s &optional (acc '()))
     (cond ( (not (consp s)) acc)
           ( (rev (cdr s) (cons (car s) acc) ) )))


======== Testes ==============
* (load "D:/Programs/lisp/mytut/code/tests/downlist.lisp")

T
* (rev '(3 4 5 6))

(6 5 4 3)
*
```

Paradigma 9. Descobrir se uma lista é um palíndrome. Em grego, palíndrome é uma frase que pode ser lida da esquerda para a direita ou da direita para a esquerda. Um palíndrome grego muito famoso é

> NIΨONANOMHMATAMHMONANOΨIN
> Lave os pecados, não apenas o rosto.

```
(defun rev (s &optional (acc '()))
     (cond ( (not (consp s)) acc)
           ( (rev (cdr s) (cons (car s) acc) ) )
     )
)

(defun palin (xs)
  (equal xs (rev xs)))

======== Testes ==============
* (load "D:/Programs/lisp/mytut/code/tests/downlist.lisp")

T
* (palin '(N I Y O N A N O M H M A T A M H M O N A N O Y I N))

T
```

Quando você precisa de construir listas, mantendo a ordem em que os elementos são visitados pelo programa, a melhor solução é a recursividade. Isto significa que seu programa vai fazer chamadas a si mesmo de dentro de um `cons`. A estrutura do programa será algo assim:
`(defun fn(s) (if (null s) '() (cons xxx (fn (cdr s))))).`

Paradigma 10. Escreva um programa que achate listas. Assim:

```
* (load "D:/Programs/lisp/mytut/code/tests/downlist.lisp")

T
* (flatten '(a (b (c d) e)))

(A B C D E)
```

Eis aqui o programa:

```
(defun flatten (s)
  (cond ( (not (consp s)) s)
        ( (consp (car s))
          (append (flatten (car s)) (flatten (cdr s))))
        ( (cons (car s) (flatten (cdr s)) ))  ))
```

Paradigma 11. Eliminar duplicatas consecutivas de uma lista de elementos. A ordem dos elementos deve ser mantida.

```
(defun eldup (xs)
   (cond ( (not (consp xs)) xs)
         ( (not (consp (cdr xs))) xs)
         ( (equal (car xs) (cadr xs))  (eldup (cdr xs)) )
         ( (cons (car xs) (eldup (cdr xs)))  )))

========= Testes ==============
* (load "D:/Programs/lisp/mytut/code/tests/downlist.lisp")

T
* (eldup '(3 a a v d f g s s d d d d f))

(3 A V D F G S D F)
```

Paradigma 12. Empacotar elementos contíguos repetidos de uma lista em sublista.

```
(defun ag (s &optional (x 'none) (acc '()))
     (cond ( (equal x 'none) (ag s (car s) acc))
           ( (null s) (cons acc s))
           ( (equal (car s) x) (ag (cdr s) x (cons x acc)) )
           ( (cons acc (ag (cdr s) (car s) (list (car s)) ) ) )))

======== Testes ==============
* (ag '(s a a a c d d e e f f g h h))

((S) (A A A) (C) (D D) (E E) (F F) (G) (H H))
```

Paradigma 13 Um dos sistemas mais simples de compactação de dados é o *código de comprimento*. Se a lista tem a forma `(a a a a b b b c c d d d e)`, ela é transformada em `((4 a)(3 b)(2 c)(3 d) (1 e))`. Use o programa anterior para escrever um codificador de comprimento corrido.

```
(defun ag (s &optional (x 'none) (acc '()))
     (cond ( (equal x 'none) (ag s (car s) acc))
           ( (null s) (cons acc s))
           ( (equal (car s) x) (ag (cdr s) x (cons x acc)) )
           ( (cons acc (ag (cdr s) (car s) (list (car s)) ) ) )))

(defun cod (xs)
  (loop for s in (ag xs) collect (list (length s) (car s))))

========= Testes =================
* (cod '(s a a a c d d e e f f g h h))

((1 S) (3 A) (1 C) (2 D) (2 E) (2 F) (1 G) (2 H))
*
```

Paradigma 14. Modifique o resultado do problema **L10** de tal maneira que um elemento sem duplicatas é simplesmente copiado na lista resultante. Apenas elementos com duplicatas são colocados na forma (N E).

```
(defun ag (s &optional (x 'none) (acc '()))
   (cond ( (equal x 'none) (ag s (car s) acc))
           ( (null s) (cons acc s))
           ( (equal (car s) x) (ag (cdr s) x (cons x acc)) )
           ( (cons acc (ag (cdr s) (car s) (list (car s))) ))))

(defun cod (xs)
  (loop for s in (ag xs)
       collect (if (= (length s) 1) (car s)
                   (list (length s) (car s)))))

* (cod '(s a a a c d d e e f f g h h))

(S (3 A) C (2 D) (2 E) (2 F) G (2 H))
```

Paradigma 15. Implemente o método de compressão com codificador de comprimento diretamente, i.e., não crie as sublistas.

```
(defun cns(n x)(if (= n 1) x (list n x)))

(defun codit (s &optional (x 'none) (acc 0))
     (cond ( (equal x 'none) (codit s (car s) acc))
           ( (null s) (list (cns acc x)))
           ( (equal (car s) x) (codit (cdr s) x (+ 1 acc)) )
           ( (cons (cns acc x) (codit (cdr s) (car s) 1 ) )  )))

* (codit '(s a a a c d d e e f f g h h))
(S (3 A) C (2 D) (2 E) (2 F) G (2 H))
```

Paradigma 16. Dado uma lista comprimida com o codificador de comprimento, construa a lista descompactada.

```
(defun cns(n x)  (if (= n 1) x (list n x)))

(defun codit (s &optional (x 'none) (acc 0))
     (cond ( (equal x 'none) (codit s (car s) acc))
           ( (null s) (list (cns acc x)))
           ( (equal (car s) x) (codit (cdr s) x (+ 1 acc)) )
           ( (cons (cns acc x) (codit (cdr s) (car s) 1 ))) ))

(defun ncons (n x) (loop for i from 1 to n collect x))

(defun decod (s)
   (loop for x in s
      append (if (consp x)  (ncons (car x) (cadr x))  (list x))))

* (defvar tst)

TST
* (setf tst (codit '(s a a a c d d e e f f g h h)))

(S (3 A) C (2 D) (2 E) (2 F) G (2 H))
* (decod tst)

(S A A A C D D E E F F G H H)
```

Paradigma 17. Escreva um programa que duplique os elementos de uma lista. E.G.:

```
(dup-elementos '(a b c d e)) => (a a b b c c d d e e)

(defun dup-elementos(xs)
   (loop for x in xs collect x collect x))
```

Paradigma 18. Escreva um programa que insere um elemento na posição n de uma lista dada. Por exemplo, `(insn 'a 3 '(1 2 3 4 5)) => (1 2 a 3 4 5)`.

```
(defun insn(elem k xs)
  (loop for x in xs for i from 0 to (length xs)
        if (= i k) collect elem collect x))
```

4.3 Loop

A filosofia da Common Lisp é oferecer ao programador todas as ferramentas e bibliotecas necessárias para realizar com mínimo esforço computações em qualquer área. Em suma, os projetistas da Common Lisp tentaram criar a linguagem mais completa possível. Isso, por um lado, facilita o desenvolvimento de aplicações. Por outro lado, torna a linguagem difícil de aprender e pouco elegante. Para perceber as vantagens e desvantagens da filosofia da Common Lisp, vamos examinar de perto uma de suas estruturas sintáticas. A forma `loop` realiza iterações, ou seja, repetições. Por exemplo, se quisermos repetir um comando cinco vezes, basta teclar:

```
* (loop repeat 5 do (print 2))
22222
```

É possível manter uma variável que enumere a repetição. Eis como fazê-lo:

```
* (loop for i from 1 to 5 do (print i))
12345
```

Podemos fazer a variável do `loop` ser decrementada, em vez de incrementada:

```
* (loop for i from 3 downto 1 do (print i))

3
2
1
```

Esse tipo de repetição existe em quase todas as linguagens procedurais. Por exemplo, ela existe na Matlab, na C, na OCAML, etc. Common Lisp, entretanto, permite que a variável percorra uma lista conectada.

```
* (loop for i in '(a b c d) do (print i))
ABCD
NIL
* (loop for i on '(a b c d) do (print i))
(A B C D)(B C D)(C D)(D)
```

No primeiro dos dois exemplos acima, visitamos cada elemento da lista. No segundo exemplo, produzimos as listas obtidas por eliminação repetida dos elementos de uma lista inicial; explicando melhor, o segundo exemplo percorre sucessivas de aplicações da função `cdr` à lista.

Da mesma forma que possui recursos para visitar os elementos de uma lista, `loop` pode visitar os elementos de qualquer sequência. Mostramos abaixo como visitar os elementos de um vetor e de uma *string*. Note que elementos tanto de *string*s quanto de vetores podem ser acessados por indexação. Mesmo assim, `loop` fornece recurso para acesso direto.

```
* (defparameter *v* #(34.0 45 67 18))

*V*
* (loop for i from 1 to 3 do (print (aref *v* i)))

45
67
18
NIL
```

```
* (loop for x across *v* do (print x))

34.0
45
67
18
NIL
```

O comando `loop` também fornece vários recursos para combinar as diferentes respostas que produz. O combinador `sum`, por exemplo, permite somar resultados da iteração:

```
* (loop for x across *v* sum x)


164.0
```

Vamos utilizar `sum` para calcular o produto interno de dois vetores.

```
* (defparameter *v* #(2 3 5 6))

*V*
* (defparameter *u* #(4 5 6 7 8))

*U*
* (loop for x1 across *v* for x2 across *u* sum (* x1 x2))

95
```

Resumindo, o comando `loop` sozinho já permitiria que se escrevesse qualquer tipo de programa. Common Lisp possui várias ferramentas parecidas com `loop`. Common Lisp não apresenta a elegância trazida pela simplicidade e pela ortogonalidade de suas ferramentas. Esta falta de ortogonalidade torna Common Lisp inapropriada para descrições de formalismos matemáticos e para demonstrações de propriedades de programas.

Explorando as potencialidades do loop. Iteração com `(loop ...)` é um método de programação poderoso, mas difícil de aprender. Vamos examinar alguns problemas e mostrar como resolvê-los utilizando apenas `loop`.

- Visitando cada elemento de uma lista.

```
* (loop for i in '(1 2 3) do (print i))

1
2
3
```

- Visitando cada `cdr` de uma lista.

```
* (loop for i on '(1 2 3) do (print i))

(1 2 3)
(2 3)
(3)
```

- Percorrendo um vetor. Vetores são prefixados com `#`.

```
* (defvar v1 #(8 4 5))

V1
* (loop for i across v1 do (format T " ~a " i))
 8  4  5
```

- Visitando cada chave de uma hashtable.

```
* (defvar h1 (make-hash-table))

#<HASH-TABLE :TEST EQL :COUNT 0 {23DAD1C9}>
* (setf (gethash 'edu h1) "edu500ac@yahoo.com")

"edu500ac@yahoo.com"
* (setf (gethash 'nero h1) "nerone@roma.gov")

"nerone@roma.gov"
* (loop for k being the hash-key of h1 do (print k))

EDU
NERO
NIL
*
```

- Visitando cada valor de uma hashtable.

```
* (loop for v being the hash-value of h1 do (print v))

"edu500ac@yahoo.com"
"nerone@roma.gov"
NIL
```

- Visitando cada par chave/valor de uma hashtable.

```
* (loop for k being the hash-key
     using (hash-value v) of h1  do (format t "~a ~a~%" k v))
EDU edu500ac@yahoo.com
NERO nerone@roma.gov
NIL
```

- Iteração com variáveis.

```
* (loop for i from 1 to 3 do (format T " ~a " i) )
 1  2  3
NIL
* (loop for i from 1 to 3 by 0.5 do (format T " ~a " i) )
 1  1.5  2.0  2.5  3.0
NIL
* (loop for i from 3 downto 1 by 0.5 do (format T " ~a " i) )
 3  2.5  2.0  1.5  1.0
NIL
```

- Iteração em intervalo aberto.

```
* (loop for i from 1 below 3 do (format T " ~a " i))
 1  2
NIL
* (loop for i from 3 above 1 do (format T " ~a " i))
 3  2
NIL
*
```

- Iteração com inicialização.

```
* (loop with a = '(1 2 3) for i in a
           do (format T " ~a " i))
 1  2  3
NIL
```

- Criando variáveis subordinadas.

```
* (loop for i from 1 to 3 for x = (* i i)
            do (format T " ~a ~a~%" i x))
 1 1
 2 4
 3 9
NIL
```

- Condição.

```
* (loop for i from 1 to 3 when (oddp i) do (print i))

1
3
NIL
```

- Condição de parada — `while`

```
* (loop for i from 1 to 3 while (< i 2) do (print i))

1
NIL
```

- Condição de parada — `until`

```
* (loop for i from 1 to 34 until (< 2 i) do (print i))

1
2
NIL
```

- Coletando elementos de lista.

```
* (loop for i from 1 to 3 collect (* i i))

(1 4 9)
```

- Concatenando listas.

```
* (loop for i from 1 to 3 append (list i i i))

(1 1 1 2 2 2 3 3 3)
```

- Contando elementos de lista.

```
* (loop for i from 1 to 3 count (oddp i))

2
```

- Somando os elementos de uma lista.

```
* (loop for i from 1 to 3 sum (* i i))

14
```

- Achando os valores máximo e mínimo.

```
* (loop for x from 0 to 10 by 0.1 maximize (-  (* x x) 80))

18.010033
* (loop for x from 0 to 10 by 0.1 minimize (- 80 (* x x) ))

-18.010033
```

- Visitando o `car` de duas listas em paralelo.

```
* (loop for x in '(a b c d e)
        for y in '(1 2 3 4 5)
        collect (list x y))

((A 1) (B 2) (C 3) (D 4) (E 5))
```

Conforme já vimos, em Common Lisp, funções são definidas com o auxílio do macro `defun`. O esqueleto básico desse macro é mostrado abaixo.

```
(defun name (lista-de-argumentos*)
  "Comentários e documentação opcional."
  expressões*)
```

Imediatamente depois da lista de argumentos, é possível introduzir uma *string* de comentários, que documente a função.

Além do macro `defun`, Common Lisp possui outras maneiras de definir funções, que explicaremos mais adiante. Por enquanto, vamos utilizar um editor de texto para produzir, compilar e executar o programa da listagem 4.1.

Listing 4.1: progs/fatorial.lsp

```
1 ;; Compile: (load "progs/fatorial.lsp")

3 (defun fat(n)
4   "Fatorial␣de␣um␣inteiro."
5   (if (< n 1) 1
6       (* (fat (- n 1)) n)) )
```

```
* (load "C:/SBCL/progs/fatorial.lsp")

T
* (fat 50)

30414093201713378043612608166064768844377641568960512000000000000
```

Para utilizar o programa, ponha o cursor no *prompt* da Lisp e carregue o arquivo. Depois disso, pode executar fatorial como mostrado acima.

Volume do universo. O exemplo apresentado na listagem 4.2 mostra uma função para converter anos luz em metros. O diâmetro do universo tem 40

bilhões de anos luz. Usando a referida função, podemos converter esse valor para metros e, em seguida, calcular o volume do universo em m^3. Este exercício aparentemente sem sentido é de extrema utilidade em criptografia (ver [John Allen Paulos]). Afinal, se um código não pode ser decifrado por um computador do tamanho do universo, utilizando componentes menores do que um próton e que foi ligado no *big bang*, ele é indecifrável para todos os efeitos práticos.

```
* (load "progs/anosluz.lsp")

T
* (anosluz->metros 40000000000)

37843200000000000000000000
* (let ( (r (anosluz->metros 20d9)))
          (* 4/3 pi r r r))

2.8376719107484924d79
```

Listing 4.2: progs/anosluz.lsp

```
1 ;; Compile: (load "progs/anosluz.lsp")

3 (defun anosluz→metros(a)
4     "Converte␣anos␣luz␣para␣metros"
5     (* a 365 24 60 60 300000 1000)
6 )
```

No cálculo do volume do universo, utilizamos a forma `LET*` para introduzir variáveis locais. Considerando que a forma `LET*` é de extrema importância no estilo funcional de programação, vamos repetir a explicação de seu funcionamento dada na página 81; desta vez, porém, trataremos a forma `LET` no contexto do cálculo do volume do universo. Nestes termos, reapresentamos abaixo a sintaxe do `LET*`. Podemos ver que esta estrutura tem duas partes: (1) Uma lista de variáveis locais com seus valores e (2) uma sequência de comandos que devem ser executados com as variáveis; note que, na página 81,

o corpo da `LET*` contém apenas o valor da expressão; aqui mostramos como introduzir uma série de comandos no corpo. Cada variável local do `LET*` é introduzida por uma lista `(<nome-da-variável> <valor>)`.

```
(LET* ( (r (anosluz->metros 20d9) )                    ⎫
        (v (* 4/3 pi r r r))                           ⎪
        (vm (* 4/3 pi (expt 1d-15 3))                  ⎬ ;;Variáveis locais
      )                                                ⎭

   ;; Coisas que deve ser feitas dentro do LET
   (format T "Se um computador com o volume do universo (~a m3)" v)
   (terpri)
   (format T "e componentes de ~a m3" vm)
   (format T "(menores do que um próton)" vm)
   (terpri)
   (format T "não conseguir decifrar um código, ")
   (format T "então o código é indecifrável.")
   (terpri)
   (format T "Memória do computador: ~a megabytes" (/ v vm 1d6))

   ;última coisa calculada é o valor produzido pelo LET*
   ;aqui, o valor produzido pelo LET* é o número de
   ;componentes do computador.
   (/ v vm)
)
```

Já explicamos que existem duas formas que introduzem variáveis locais, o `LET` e o `LET*` (*let* estrela). No caso do `LET*`, podemos utilizar uma variável anterior para calcular um valor. No exemplo, utilizamos `r` para calcular o valor de `v`. No caso do `LET` (sem estrela) isso não é permitido.

Corpo do let. Depois da lista de variáveis, está o corpo do `LET`, que contém uma sequência de comandos ou uma expressão de retorno, conforme já sabemos. Vamos examinar a diferença entre comando e expressão. Quando uma função computável calcula uma expressão, ela não produz efeitos co-

laterais. Por exemplo, ela não modifica o conteúdo de variáveis, de arquivos ou de dispositivos de entrada e saída. Por outro lado, comandos modificam variáveis, conteúdo de matrizes, arquivos e dispositivos de entrada e saída. Em `Common Lisp` temos duas classes de comandos:

Atribuição. O principal comando de atribuição chama-se `setf`. Fornecemos abaixo alguns exemplos de como utilizá-lo.

```
* (defun fat(n)
    (let ( (acc 1) )
      (loop for i from 1 to n do
        (setf acc (* i acc)))
      acc))
FAT
* (fat 5)

120
```

No exemplo acima, o fatorial é calculado pela atribuição destrutiva de novos valores à variável `acc`. Aliás, esta é a única maneira de realizar cálculos interativos em linguagens como Java e Python.

Entrada e saída. Comandos de entrada e saída são aqueles que realizam leitura e impressão de arquivos ou dispositivos interativos.

```
* (defun fat(n)
    (let ( (acc 1) )
      (loop for i from 1 to n do
        (print acc)
        (setf acc (* i acc)))
      acc))
FAT
* (fat 3)

1
1
2
6
```

No exemplo do volume do universo, o corpo de `LET*` tem apenas comandos de entrada e saída. Explicamos abaixo o funcionamento desses comandos.

- `(format T "e componentes de ~a m3" vm)` — imprime um texto (se seu primeiro argumento é `T`) ou constrói uma *string* (se seu primeiro argumento é `NIL`). Antes da impressão, `format` preenche todas as diretivas `~a` com valores que seguem o texto. No caso, a única diretiva `~a` é preenchida com o valor de `vm`, e Lisp imprime o seguinte texto: `e componentes de 4.188790204786392d-45 m3`

- `(terpri)` — salta uma linha durante o processo de impressão.

- `(/ v vm)` — resultado do `LET`.

A função `raizes` da listagem 4.3, calcula as raízes de uma equação do segundo grau. Neste programa, as duas raízes são empacotadas em uma lista, de forma a produzir um único valor.

Listing 4.3: progs/bhaskara.lsp

```
1 (defun raizes(a b c)
2    (let* (  (delta (- (* b b) (* 4 a c)))
3             (d (sqrt delta))
4             (mb (- b))
5          )
6       (list (/ (+ mb d) 2 a)
7             (/ (- mb d) 2 a))
8    )
9 )
```

```
* (load "progs/bhaskara.lsp")

T
* (raizes 5 3 1)

(#C(-0.3 0.33166248) #C(-0.3 -0.33166248))
```

4.4 Programação funcional

A matemática não contempla a utilização de comandos. Isto significa que, quando utilizamos comandos em um programa, abrimos mão de todos os recursos oferecidos pela lógica e pela matemática para garantir a correção do sistema. Por isso, há um estilo de programação que nunca utiliza comandos como `setf`. Este estilo é denominado programação funcional. Em Lisp, mesmo os adeptos da programação funcional usam comandos de entrada e saída, argumentando que esses comandos não afetam a correção e simplificam o desenvolvimento de sistemas. Há, porém, cientistas de computação que nem mesmo para imprimir e ler dados usam comandos. Vamos analisar como estas pessoas trabalham.

Na linguagem Clean, podemos definir funções de uma forma não muito diferente da utilizada na Lisp. Eis a definição de fatorial em Clean:

```
module fatorial
import StdEnv, ArgEnv

fat 0=1
fat n= n*fat(n-1)

Start= fat (toInt argv.[1])
where
   argv= getCommandLine
```

Admitamos que o programa acima esteja guardado no arquivo `fatorial.icl`. O compilador Clean gera um executável que calcula e apresenta o fatorial de um número:

```
C:\Clean 2.2\exemplos\console>fatorial.exe 5
120

C:\Clean 2.2\exemplos\console>fatorial 6
720
```

Agora vamos ver como efetuar impressões e leituras da console para ver como isso pode ser realizado apenas com expressões funcionais (sem comandos). O programa abaixo lê um nome da console e imprime uma saudação.

```
module hello
import StdEnv

Start mundo= fwrites ("Hello, "+++nome) tela2
where
  (tela, mundo1)= stdio mundo
  tela1= fwrites "Escreva seu nome: " tela
  (nome, tela2)= freadline tela1
```

As funções que manipulam dispositivos de entrada e saída recebem uma alça para o dispositivo e retornam a alça modificada. O fato de retornar a alça modificada faz com que o sistema aja como uma verdadeira função.

```
C:\Clean 2.2\exemplos\console>hello
Escreva seu nome: Paulo
Hello, Paulo
```

No exemplo acima, a função `(tela, mundo1)= stdio mundo` toma o mundo como argumento e volta uma alça para a tela e o mundo modificado. No exemplo, a alça chama-se `tela` e o mundo modificado chama-se `mundo1`. O mundo contém os atributos do mundo real que desejamos manipular: impressoras, robôs, monitor, teclado e arquivos. A função

```
tela1= fwrites "Escreva seu nome: " tela
```

escreve um texto na alça `tela` e volta o alça modificada, ou seja, `tela1`. A diferença entre `tela` e `tela1` é que essa última alça tem algo escrito nela, a saber, o primeiro argumento de `fwrites`.

A função `(nome, tela2)= freadline tela1` lê uma cadeia de caracteres e retorna o par ordenado `(nome, tela2)`, onde `nome` é a cadeia lida e `tela2` é a alça modificada. Finalmente apresenta-se o resultado de `Start`:

```
Start mundo= fwrites ("Hello, "+++nome) tela2
```

que é o resultado da concatenação da *string* `"Hello, "` com `nome` (a operação `+++` efetua a concatenação) e imprime o resultado em `tela2`, produzindo o resultado final da função `Start`.

Relação. Par ordenado é uma dupla de elementos de um conjunto onde a ordem é relevante. Os pares ordenados são representados com os elementos separados por uma vírgula e entre parênteses. Exemplos: $(1, 2)$, $(3.14, 5)$.

Dados dois conjuntos A e B, produto cartesiano $A \times B$ é o conjunto de todos os pares ordenados (a, b), onde $a \in A$ e $b \in B$. Matematicamente, esta definição escreve-se assim: $A \times B = \{(a, b) \backslash\backslash a \in A \wedge b \in B\}$

Listing 4.4: cartesiano.icl — Relações.

```
2 module cartesiano;
3 import StdEnv;

5 Start
6   # cA = [1, 2, 3];
7     cB = [4, 5];
8     cAxB = [(x, y) \\ x ∈ cA, y ∈ cB];
9   = cAxB;
```

Dados dois conjuntos de $A = \{1, 2, 3\}$ e $B = \{4, 5\}$, o programa da listagem 4.4 fornece o produto cartesiano $A \times B$.

Funções. Uma classe de relações especialmente importante é a classe das funções, que é o instrumento matemático usado para mostrar uma relação entre duas quantidades. A primeira definição de função foi atribuída a Leibniz que define função como: *Dado duas variáveis x e y, se cada valor de x determina um único valor de y, então dizemos que y é função de x.* Existem três maneiras de se escrever funções:

1. Tabelas, como nas linhas 4, 5, 6, 7 e 8 da listagem 4.5.
2. Expressões algébricas, como mostrada na linha 9 da listagem 4.5.
3. Gráficos.

Listing 4.5: fatorial.icl — Tabelas e Expressões.

```
1  module fatorial;
2  import StdEnv, ArgEnv;

4  fatorial 0= 1;
5  fatorial 1= 1;
6  fatorial 2= 2;
7  fatorial 3= 6;
8  fatorial 4= 24;
9  fatorial n= n*fatorial(n-1);


12 Start
13  # c= getCommandLine; // c= {"fatorial", "5"}
14  | size c <> 2 = abort "Uso: fatorial <real> \n";
15  = fatorial (toInt c.[1]);
```

4.5 Listas

Uma das mais importantes estruturas de dados é a lista encadeada. De um ponto de vista abstrato, uma lista é uma sequência de elementos em ordem. Na listagem 4.3, a resposta foi dada na forma de uma lista contendo as raízes. Exemplos de listas:

```
'(3 4 5 6 1)                        ; Lista de inteiros
'(3 lápis 5 cadernos)               ; Lista com inteiros e símbolos
(+ 4 5 6)                           ; Lista representando uma soma
'( (3 maçãs) (4 peras) (5 uvas) ) ; Lista de listas
```

Nos exemplos acima, notamos que uma lista pode conter elementos de tipos diferentes. Outro ponto importante é que Lisp considera que toda lista nua é um programa e deve ser executada; o primeiro elemento da lista é visto como uma operação e o resto da lista são os argumentos da operação. Para avisar que a lista representa uma estrutura de dados, é preciso prefixá-la com um apóstrofo, como mostrado acima.

Estruturas de dados possuem partes. No caso das listas, as partes são a cabeça (primeiro elemento) e a cauda (a lista dos elementos que seguem o primeiro). Toda estrutura de dados necessita dos seguintes recursos:

- Construtor monta a estrutura. Por exemplo, `(cons 3 '(a b c))` monta uma lista de cabeça `3` e cauda `(a b c)`, ou seja, `(3 a b c)`.
- Seletores fornecem as partes da estrutura. Listas possuem dois seletores: `(car s)` produz a cabeça de `s` e `(cdr s)` retorna a cauda de `s`. Isso significa que o resultado de `(car '(3 4 5))` é `3`. Analogamente, o resultado de `(cdr '(3 4 5))` é `(4 5)`.
- Predicados para detetar propriedades da estrutura. O predicado `(null s)` produz `T` — *True* — quando `s` é vazia e `NIL` no caso contrário.

Vamos testar algumas das funções que Lisp possui para processar listas.

```
* (defvar xs) ; Criei uma variável xs

XS
* (setf xs '(a b c d))

(A B C D)
* (car xs)  ; Primeiro elemento de xs

A
* (cdr xs)   ; Cauda de xs

(B C D)
* xs      ; Seletores não modificam a lista

(A B C D)
* (cons 34 xs) ; Construo lista de cabeça 34 e cauda xs

(34 A B C D)
* xs   ; Construtores não modificam a lista

(A B C D)
* (push 3 xs)  ;; Modifica xs, empilhando um 3 sobre (A B C D).
```

```
(3 A B C D)
* xs           ;; Note que o valor de xs foi modificado.

(3 A B C D)
* (pop xs)     ;; Modifica xs, desempilhando o valor do topo.

3
* xs           ;; xs voltou ao valor antigo.

(A B C D)
```

Um engano muito comum entre principiantes é pensar que `cdr` modifica a lista. O exemplo acima mostra que isso não acontece. Se você quiser modificar o valor de uma variável, use as funções `push`, `pop` e `setf`.

O operador `mapcar` aplica uma função a cada elemento de uma lista. A função pode ser uma das já existentes no Lisp — como `sqrt`, `sin` ou `cos` — ou pode ser uma expressão lambda. Uma expressão lambda tem a forma $\lambda(x)E(x)$ e representa uma função de variável x.

```
* (setf xs '(4 9 16 49))

(4 9 16 49)
* xs

(4 9 16 49)
* (mapcar #'sqrt xs)

(2.0 3.0 4.0 7.0)
* xs

(4 9 16 49)
* (mapcar (lambda(x) (list x (* x x)))  xs)

((4 16) (9 81) (16 256) (49 2401))
* (list 3 4 5 'a 'b)

(3 4 5 A B)
* (list 3 (* 3 3))

(3 9)
```

A função `list` produz uma lista; `(lambda(x) (list x (* x x)))`, portanto, gera uma lista contendo `x` e o quadrado de `x`.

4.6 Format

O comando `(format T " ~a ~% ~a" 3 (* 3 3))` imprime um texto (se seu primeiro argumento é `T`) ou constrói uma *string* (se seu primeiro argumento é `NIL`). Antes da impressão, `format` preenche todas as diretivas `~a` com os valores que seguem o texto. Abaixo, mostramos como usar esse comando para formatar a informação impressa.

```
* (format T " ~a ~% ~a" 3 (* 3 3)) ;; "~%" quebra de linha.
 3
 9
NIL
* (format NIL "~a ~30t ~a" 1 (atan 1)) ;; "~15t" tabulação.

"1                              0.7853982"
* (format NIL "Custa US$ ~$ " 25.30) ;; "~$" --- centavos.

"Custa US$ 25.30 "
* (format NIL "~f" pi)  ;; "~f" --- Número em ponto flutuante.

"3.141592653589793"
* (format NIL "~10,3f" pi)  ;; Campo: 10; Precisão: 3

"     3.142"
* (format nil "~10<~a~;~a~;~a~>" 3 4 5)  ;; Ajusta texto

"3   4    5"
* (format nil "~12:< ~a~>" 3458)  ;; Ajuste à direita

"        3458"
* (format nil "~12@< ~a~>" 3458)  ;; Ajuste à esquerda

" 3458       "
* (format NIL "~a~~ (til); ~a~5~ (tils)" 1 5) ;; Imprime til

"1~ (til); 5~~~~~ (tils)"
```

```
* (format nil "~12:< ~a~>" 3458)  ;; Ajuste à direita

"        3458"
* (format nil "~12@< ~a~>" 3458)  ;; Ajuste à esquerda

" 3458       "
* (format NIL "Base 16: ~X;  Base 8: ~O" 23 23) ;; Base 16 e 8

"Base 16: 17;  Base 8: 27"
* (format NIL "Base 10: ~6D;  Base 2: ~6B" 23 23) ;; Decimal e binário

"Base 10:     23;  Base 2:  10111"
* (format T "~4D (~R)" 23 23)  ;;Inteiro, em inglês
  23 (twenty-three)
NIL
* (format T "~4Dth (~:R)" 23 23)  ;;Ordinal
  23th (twenty-third)
NIL
* (format T "~4D (~@R)" 23 23)  ;;Numeração romana
  23 (XXIII)
NIL
```

4.7 Emacs

Emacs é escrito em Elisp, uma linguagem que hoje em dia é utilizada quase que exclusivamente para configurar editores. Entretanto, isto significa que, quem sabe Lisp, consegue reprogramar o Emacs. Vamos começar nosso estudo do Emacs descobrindo como obter informações sobre as teclas do editor.

MODIFICADORES

As teclas de controle, shift e alt, devem ser mantidas abaixadas antes de apertar a tecla que a segue. Por isso, elas são denominadas modificadores. ESC não é um modificador, pois é preciso soltá-la antes de apertar a próxima tecla. Quando você aperta ESC, é levado ao minibuffer (janela de uma linha só no rodapé da página). Note que `\M-x` e `\M x` são equivalentes, mas uma é obtida com o modificador **Alt**, enquanto a outra é produzida com a tecla

ESC. Exemplos:

- `\C-h` Mantenha a tecla de controle abaixada e aperte h.
- `\M-x` Mantenha a tecla alt abaixada e aperte x.
- `\M x` Aperte a tecla ESC e solte-a. Depois aperte x.

DESCRIÇÃO DE ALGUMAS TECLAS

Agora vamos ver como utilizar as teclas prefixadas descritas acima. Nos exemplos abaixo, você aprenderá a obter ajuda e a abrir e fechar janelas.

- `\C-h ?` Descreve as teclas utilizadas para obter ajuda.
- `\C-h b` Descreve os bindings de todas as teclas.
- `\C-h h` Saudações em diversas línguas.
- `\C-x 1` Apaga a outra janela.
- `\C-x 0` Apaga esta janela.
- `\C-x 2` Abre duas janelas.

Agora marque um texto qualquer, mantendo o botão do mouse abaixado e arrastando o sobre o texto, que ficará amarelo. Feito isso, leve o cursor para um ponto de inserção e aperte `"\C-v"`. Em vez de scroll-up, a região que foi marcada será inserida no texto. Para restaurar scroll-up, vá para o comando de restauração que escrevemos acima, coloque o cursor na frente dele, e aperte `"\C-x \C-e"` novamente.

Vamos agora ver como modificar Emacs. Em primeiro lugar, vejamos o que faz duas teclas: `"\C-v"` e `"C-y"` Para isso, tecle `"\C-h b"`. Isto produzirá uma listagem dos bindings de todas as teclas. Examinando a lista, descobrimos que `"\C-v"` produz um scroll para cima. `"\C-y"` produz um yank, ou seja, traz o conteúdo do clipboard para o ponto de inserção. Agora que já sabemos os bindings que conhecemos, vamos fechar a janela de bindings. Aperte `"\C-x 1"`. Vamos guardar o significado original de `"\C-v"` para poder restaurar o estado do editor:

```
(global-set-key "\C-v" 'scroll-up)
```

Vamos colocar o yank em `"\C-v"`. Ponha o cursor na frente da expressão-S abaixo e aperte `"\C-x \C-e"`.

```
(global-set-key "\C-v" 'yank)  ;; <\C-x \C-e>
```

Agora marque um texto qualquer, mantendo o botão do mouse abaixado e arrastando o sobre o texto, que ficará amarelo. Feito isso, leve o cursor para um ponto de inserção e aperte `"\C-v"`. Em vez de scroll-up, a região que foi marcada será inserida no texto. Para restaurar scroll-up, vá para o comando de restauração que escrevemos acima, coloque o cursor na frente dele, e aperte `"\C-x \C-e"` novamente.

Definindo Funções

Para rolar o texto uma linha para cima, usamos `"\C-u 1 \C-v"`, o que é muito trabalhoso para tão pouco resultado. Da mesma forma, para rolar uma linha para baixo, precisamos de teclar `"\C-u 1 \M-v"`, que também é inconveniente. Inicialmente, vamos renomear as funções scroll-up e scroll-down:

```
(defalias 'scroll-ahead 'scroll-up)      ;<\C-x \C-e>
(defalias 'scroll-behind 'scroll-down)   ;<\C-x \C-e>

(defun scroll-one-line-ahead ()
    "Scroll ahead one line"
   (interactive)
   (scroll-ahead 1))   ;<\C-x \C-e>

(defun scroll-one-line-behind ()
   "Scroll behind one line"
   (interactive)
   (scroll-behind 1))   ;<\C-x \C-e>
```

Agora vamos fazer o binding dessas funções nas teclas `"\C-v"` e `"\M-v"`, como mostrado abaixo.

```
(global-set-key "\C-v"  'scroll-one-line-ahead)   ;<\C-x \C-e>
(global-set-key "\M-v"  'scroll-one-line-behind)  ;<\C-x \C-e>
```

Agora experimente apertar `"\C-v"` diversas vezes (mantenha a tecla `"\C"` abaixada e aperte `"v"` diversas vezes. Experimente também apertar `"\M-v"` diversas vezes (mantenha alt abaixada e aperte `"v"`).

INSERINDO UMA TABELA

Agora vamos escrever um programa para inserir uma tabela do LaTeX em um texto. Antes de mais nada, contudo, precisamos especificar a finalidade da tabela: Descrever as teclas do LaTeX. Isto significa que a tabela terá duas colunas, uma para as teclas e outra para descrever o que elas fazem. Podemos reservar `4cm` para a tecla e `8cm` para a descrição. Compile o programa abaixo, que especifica uma tabela como a desejada, testando-o logo em seguida. Se o teste não apontar nenhum erro, associe o programa a uma tecla.

```
(defun insert-latex-table ()
  "Insere uma tabela do LaTeX"
  (interactive)
  (insert "\n\\begin{tabular}{p{4cm}p{8cm}}\n")
  (insert "\\end{tabular}"))  ;<\C-x \C-e>

(global-set-key "\C-t"   'insert-latex-table) ;<\C-x \C-e>
```

Abaixo, você vai encontrar uma tabela com as principais teclas do Emacs.

`\C-h ?`	Descreve as teclas de ajuda.
`\C-h ?`	Descreve as teclas de ajuda.
`\C-h b`	Descreve os bindings de todas as teclas.
`\C-h h`	Saudações em diversas línguas.
`\C-x 1`	Apaga a outra janela.
`\C-x 0`	Apaga esta janela.
`\C-x 2`	split-window-vertically
`\C-x 3`	split-window-horizontally
`C-y`	Insere conteúdo do clipboard no ponto
`M-y`	Gira o kill-ring
`C-c C-d`	Salva documento

`C-@`	set-mark-command
`C-a`	move-beginning-of-line
`C-b`	backward-char
`C-d`	delete-char
`C-e`	move-end-of-line
`C-f`	forward-char
`C-g`	keyboard-quit
`C-h`	help-command
`C-k`	kill-line
`C-n`	next-line
`C-p`	previous-line
`C-o`	open-line
`C-r`	isearch-backward
`C-s`	isearch-forward
`C-w`	kill-region
`C-_`	undo
`C-SPC`	set-mark-command
`C-/`	undo
`C-x C-c`	Termina sessão do Emacs.
`C-x C-f`	Lê arquivo.
`C-x C-s`	Salva arquivo.
`C-x C-b`	Lista buffers.
`C-x s`	Salva buffers.

4.8 Entrada e Saída

Vamos imaginar que você tenha um arquivo `"C:/lisp-data/test.dat"` contendo uma lista de números:

```
; Arquivo de dados: C:/lisp-data/test.dat

(10 20 34.5 60 80 90)
```

O programa da listagem 4.6 é capaz de ler o arquivo, e construir uma tabela contendo o quadrado dos números.

Listing 4.6: progs/readwrite.lisp

```
(defun square-table(fName)
  (with-open-file (s (merge-pathnames
                       "C:/lisp-data/" fName))
    (mapcar (lambda(x) (list x (* x x)))
            (read s))))
```

Quando você inicia a `LispIDE`, o editor de programas coloca-o no diretório onde a Lisp foi instalada. Pode acontecer, contudo, que você precise trabalhar em outro diretório. No exemplo abaixo, mostramos como mudar a pasta de trabalho.

```
* (setf *default-pathname-defaults* #P"C:/SBCL/progs/")

#P"C:\\SBCL\\progs\\"
* (load "readwrite.lisp")

T
* (square-table "test.dat")

((10 100) (20 400) (34.5 1190.25) (60 3600)
 (80 6400) (90 8100))
*
```

No próximo exemplo, vamos ler uma lista de ângulos e gravar uma tabela de senos no arquivo `out.txt`. A listagem 4.7 apresenta várias coisas interessantes.

Listing 4.7: progs/sintab.lisp

```
(defun sin-table(f-in)
  (with-open-file (in f-in)
   (with-open-file (W#P"C:/lisp-data/out.txt"
                       :direction :output
                       :if-exists :supersede)
     (loop for x in (read in) do
       (format W "sin(~a) ~30t ~a ~%" x (sin x))) )))
```

Na seção 4.6 (página 112), aprendemos que o primeiro argumento de `format` pode ser `T`, para imprimir texto, ou `NIL`, para construir *strings*. O primeiro argumento de `format` pode também ser um fluxo de informação – *stream* – para um arquivo. Neste caso, o texto formatado é inserido no arquivo.

Na listagem 4.7, `with-open-file` criou um fluxo de informação que deve ser encaminhado para o arquivo `#P"C:/lisp-data/out.txt"`; este fluxo foi identificado com a variável `W`. A direção do fluxo é de escrita, o que foi indicado com os parâmetros `:direction :output`. Se o arquivo já existe, o programa escreve os novos dados por cima dos velhos; esse comportamento é especificado pelos parâmetros `:if-exists :supersede`.

Para repetir uma sequência de comandos, podemos usar a forma `loop`. Eis um programa para calcular uma tabela de senos:

```
1 (defun tabela (dg)
2   (format T "␣angulo␣␣␣␣␣␣␣␣seno")
3   (terpri)
4   (loop
5     for i from 0.0 to 360.0 by dg
6     for s = (sin (* i (/ pi 180)))
7       do (format T "~8,0f␣␣␣␣␣~7,3f~%" i s)))
```

Na linha 5, fazemos `i` variar de 0 a 360, com incremento `dg`. Na linha 6, calculamos o valor do seno para cada valor do ângulo `i`. Na linha 7, imprimimos o resultado.

```
* (tabela 90)
 angulo        seno
      0.       0.000
     90.       1.000
    180.      0.0000
    270.      -1.000
    360.     -0.0000
NIL
*
```

No livro *O Código Da Vinci*, as sequência dos números de Fibonacci foi usada pela Princesa Sophie como senha de sua conta numerada em um banco

suíço. Matematicamente, os números de Fibonacci são definidos da seguinte maneira:

$$(\phi\ i) \ = \ \begin{cases} 1 \text{ quando } i = 0 & \text{;Linha 2} \\ 1 \text{ quando } i = 1 & \text{;Linha 3} \\ (\phi\ i-1) + (\phi\ i-2) \text{ quando } n > 1 & \text{;Linha 4} \end{cases}$$

```
1 (defun fibo(n)
2    (loop for f_i-2 = 1 then f_i-1
3          for f_i-1 = 1 then f_i
4          for f_i = (+ f_i-1 f_i-2) as i from 2 below n
5          finally (return f_i)))
```

Vamos usar o `loop` para definir a função de Fibonacci. Na linha 4, note que `i` vai de 2 até `n`. Na linha 2, para `i=2`, temos que `f_i-2=f_0=1`. Na linha 3, também para `i=2`, temos que `f_i-1=f_1=1`. Finalmente, a linha 4 mostra que, para i variando de `2` a `n`, `f_i= f_i-1 + f_i_2`.

Capítulo 5

Scheme: O outro Lisp

Common Lisp tem por princípio fornecer uma vasta escolha de bibliotecas, macros e estilos de programação. Esta postura vai de encontro à filosofia da Matemática, que é economia de recursos. Além do mais, Common Lisp é pouco elegante, por armazenar identificadores de dados e de funções em espaços diferentes.

```
* (defparameter *xs* '(a b c d))

*XS*
* (defparameter fn (lambda(s) (car (cdr s))))

FN
* (fn *xs*)

; in: LAMBDA NIL
;     (FN *XS*)
;
; caught STYLE-WARNING:
;   undefined function: FN
UNDEFINED-FUNCTION: The function FN is undefined.

* (funcall fn *xs*)

B
```

Pelo cálculo lambda, não deveria haver diferença entre `fn` e `*xs*`. Common Lisp, entretanto, armazena funções e dados em espaços de nome diferentes. Então, para chamar uma função armazenada em uma variável, é preciso prefixá-la com `funcall`, conforme mostrado no exemplo acima. Outra situação incômoda ocorre quando precisamos passar uma função como parâmetro:

```
* (reduce #'+ '(2 3 4 5 6))

20
```

A função `reduce` tem dois argumentos; o primeiro argumento é a função `+`, que deve ser inserida entre os elementos da lista, que vai no segundo argumento. Entretanto, conforme podemos ver, Common Lisp exige que a função `+` receba um prefixo `#'+` para ser passada como argumento. A conclusão disso tudo é que Common Lisp trata funções e, por exemplo, números de forma diferente, o que não é muito elegante. Os adeptos da linguagem Scheme procuram reconciliar a sintaxe flexível e poderosa da Common Lisp com um rigor matemático. Isso significa que a Scheme tem a mesma sintaxe da Common Lisp, com todas as vantagens de regularidade e flexibilidade que tal sintaxe oferece. Por outro lado, Scheme visa a concisão e ortogonalidade da matemática, além da uniformidade semântica.

Funções são ortogonais se nenhuma delas pode ser representada como uma combinação das outras e o conjunto das funções ortogonais é suficiente para representar por combinação os outros elementos de um espaço de funções. A ortogonalidade expressa o princípio de economia e suficiência. Scheme é ortogonal: não se fornece ferramentas que possam ser obtidas pela combinação das já existentes. Por outro lado, nada falta para realizar qualquer computação.

Vamos voltar ao exemplo da função `reduce`. Common Lisp fornece essa função pronta para usar, o que é mostrado no exemplo abaixo, que calcula a média e o desvio padrão de uma lista de números.

```
(defun media(s)
   (/ (reduce #'+ s) (float(length s))))
```

```
(defun sd(s)
   (let ((m (media s)))
      (media (mapcar (lambda(x) (*(- x m) (- x m))) s))))
```

A seguir, veremos que Scheme, embora mais elegante e rigorosa, não é tão prática quanto a Common Lisp.

5.1 Programando em Scheme

Sendo ortogonal, Scheme não tem a função `reduce` que, entretanto, pode ser definida facilmente.

```
(define (reduce fn s)
  (match-case s
    ( (?x) x)
    ( (?x . ?xs) (fn x (reduce fn xs)))
   )
)

(define (media s)
   (/ (reduce + s) (length s)))

(define (sd s)
   (let* ( (m (media s))
           (fn (lambda(x) (* (- x m) (- x m))) ))
      (media (map fn s))))
```

A vantagem da versão Scheme é que não precisamos de nenhum prefixo para passar uma função como argumento de `reduce`:

```
1:=> (reduce + '(3 4 5 6))
18
1:=>
```

A vantagem da Common Lisp é que a função `reduce` já estava pronta. Acontece, porém, que os programadores Scheme acabam por criar vastas bibliotecas que reproduzem todos os recursos da Common Lisp, muitas vezes com

vantagem. Por exemplo, a função `match` que utilizamos para definir `reduce` está em uma biblioteca, que supre a falta da função `destructuring-bind`, existente na Common Lisp. Acontece que `match` é muito mais poderosa do que o recurso que emulou.

Tal como em Lisp, os programas em Scheme são escritos na forma de listas. Como a linguagem manipula listas bem, e como os programas são listas, ela consegue manipular seus próprios programas. Assim sendo:

- Programas podem aprender coisas, tornando-se mais poderosos.
- Você pode adaptar a linguagem acrescentando comandos especializados no problema que quer resolver. Isto pode tornar os programas mais curtos, mais rápidos ou mais fáceis de entender.
- Como todos os comandos da Scheme têm a mesma estrutura, isto é, todos são listas, é mais fácil aprender a linguagem.
- Programas podem descobrir coisas e incorporar a descoberta.

Apesar da extrema simplicidade da Scheme, o que mais assusta na linguagem é exatamente esta simplicidade. As pessoas não estão acostumadas com coisas simples e, por isso, acreditam que coisas simples são difíceis.

Em Scheme, estruturas de dados e comandos são listas, e listas são representados por elementos entre parênteses. Não há exceção a esta regra. Para diferenciar um comando de uma estrutura de dados, prefixa-se a estrutura com um apóstrofo. Exemplos de comandos e de dados:

`'(rosa cravo orquídea)`	Estrutura de dados, porque está prefixada por um apóstrofo.
`(+ 3 4 5 6)` => 18	Comando para somar. Scheme executa o comando, e produz um valor. No caso, o valor é 18.

Como você pode notar, para executar a operação `3+4+5+6`, usamos o comando `(+ 3 4 5 6)`.

Mais exemplos de comandos:

`(* 3 4 5)` => 60	Comando para multiplicar.
`(/ 60 3 4)` => 5	Comando para dividir: $60/3/4 = 5$
`(- 60 3 4)` => 53	Comando para subtrair: $60 - 3 - 4 = 53$
`(+ (* 2 3 4) 30 (* 7 8 9))` => 558	$2 \times 3 \times 4 + 30 + 7 \times 8 \times 9$
`(* 4 (atan 1))` => 3.1415926535898	4 vezes arco cuja tangente é 1 vale π. Fórmula prática para encontrar π.
`(define pi (* 4 (atan 1)))`	Definição de π.
`(* pi 10 10)` => 314.15926535898	Uso de π para encontrar a área de uma círculo cujo raio é 10.
`(define (area r) (* pi r r))`	Definição do comando area, que acha a área de um círculo de raio r.

Vamos aprender a escrever um programa para calcular o fatorial de um número. Eis como quero usar este programa:

```
C:\scheme-eg>fac 5
120
```

Matematicamente, o fatorial é definido como:

$$n! = \begin{cases} 1 \text{ quando } n = 0 \\ n \times (n-1)! \text{ quando } n > 0 \end{cases}$$

Há várias maneiras de expressar estas condições em Scheme. A maneira mais direta é usar um `(if c e1 e2)`. Caso a condição `c` seja verdadeira, a resposta é a expressão `e1`; se a condição for falsa, a resposta é `e2`.

```
;Compile: bigloo fac.scm -o fac.exe                    1

(module factorial (main start))                       3

(define (factorial n)                                 5
  (if (< n 2) 1                                       6
      (* n (factorial (- n 1)) )                      7
   );end-if                                           8
);end-define                                          9

(define (arg1 x)                                      11
  (string->number (car (cdr x))))                     12

(define (start c)                                     14
  (print (factorial (arg1 c) ))                       15
);end-define                                          16
```

O programa acima começa com uma linha de comentário, mostrando como devemos efetuar a compilação. A parte da linha que começa com ponto e vírgula é considerada comentário. No caso, a diretiva de compilação é um comentário. Também as expressões `;end-if`, `;end-define`, etc. são comentários.

As linhas de 5 a 9 definem fatorial. A linha 3 declara o módulo e informa que a função principal (main function) chama-se `start`. O programa vai começar com `(start c)`. A lista `c` contém a linha de comando. Por isso, se eu digitar

```
fac 5
```

terei `c=("fac" "5")`. A função `(car xs)` retorna o primeiro elemento da lista `xs`. A função `(cdr xs)` retorna o que sobra da lista se retiramos o primeiro elemento. Aliás, já aprendemos isso tudo quando estudamos Common Lisp. Veremos que a semelhança entre as duas linguagens não termina aqui. Exemplos de uso das funções `car` e `cdr`:

```
(define xs '("fac" "5" "6" "7" "8"))
(car xs) => "fac"
(cdr xs) => ("5" "6" "7" "8")
(car (cdr xs)) => "5"
```

Na linha 12, `string->number` transforma a string `"5"` no número 5. Então, na linha 15, se digitarmos `fac 5`, a função `(arg1 c)` produz o número 5; o programa, então, calcula e imprime o fatorial deste número.

O problema com a forma `(if c e1 e2)` é que permite apenas uma condição. No caso de duas ou mais condições, usamos `cond`.

```
;Compilar: bigloo -static-bigloo trig.scm -o trig.exe                    1

(module trigonometria (main main))                                       3

(define (call f x)                                                       5
   (cond                                                                 6
        ( (equal? f "sin"); condição                                     7
          (display "(sin ")                                              8
          (display x)                                                    9
          (print ")= " (sin (string->number x)))                         10
        );fim da primeira condição                                       11
        ( (equal? f "cos");                                              12
          (display "(cos ")                                              13
          (print x ")= " (cos (string->number x)))                       14
        ); fim da condição 2                                             15
        (else (print "Só sei calcular:")                                 16
              (print "  sin e cos.")                                     17
              (print "Sinto muito.")                                     18
        );fim do else                                                    19
    );fim do cond                                                        20
);fim do define                                                          21

(define (main c)                                                         23
   (call (car (cdr c)) (car (cdr (cdr c))) ))                            24
```

O programa acima calcula uma ou outra função trigonométrica, conforme a linha de comando:

```
C:\lisp-eg>trig sin 3
(sin 3)= 0.14112000805987

C:\lisp-eg>trig cos 3
(cos 3)= -0.98999249660045
```

As linhas 7, 8, 9, 10 e 11 tratam da primeira condição (cálculo do seno). As linhas 12, 13, 14 e 15 cuidam do cosseno. Finalmente, as linhas 16, 17, 18 e 19 acertam o caso de nenhuma das condições anteriores funcionar.

O programa da página 127 ficaria mais claro se pudéssemos armazenar cálculos parciais em variáveis. Novamente, Scheme é idêntica à LISP neste pont: a introdução de variáveis locais é conseguida com auxílio da forma `let`:

```
(let [ (f (função-que-calcula-f))
       (x (função-que-calcula-x))
       (pi valor-de-pi)
     ]                                  ;;Variáveis locais

         ;;Comandos dentro do let
        (comando_1 ....)
        (comando_2 ....)
        (comando_3 ....)
);fim do let
```

Eis como fica o programa com `let`:

```
;bigloo -static-bigloo calcule.scm -o calcule.exe                    1

(module trigonometria (main main))                                   3

(define (main c)                                                     5
  (let ( (f (cadr c))                                                6
         (x (string→number (caddr c)) )                              7
       );fim da lista de variáveis                                   8

    (cond     ( (equal? f "sin"); condição                           10
                (print "(sin " x ")=" (sin x))     )                11
              ( (equal? f "cos");                                    12
                (print "(cos " x ")=" (cos x))     )                13
              (else (print "Não sei calcular " c)  )                 14
    );fim do cond                                                    15
  );fim do let                                                       16
);fim do define                                                      17
```

No programa da página 128 temos várias novidades. A primeira é que podemos utilizar a função `(cadr c)` para obter o segundo elemento de uma lista; de fato, `(cadr c)= (car (cdr c))`. De maneira análoga, podemos obter o terceiro elemento de uma lista com `(caddr c)`.

```
;bigloo -static-bigloo calcule.scm -o calcule.exe                    1

(module trigonometria (main main))                                   3

(define pi (* 4 (atan 1)))                                           5

(define (em-caso-de-erro msg)                                        7
   (print "Para␣usar,␣digite:")                                      8
   (print "␣␣␣calcule␣sin␣angulo")                                   9
   (print "ou")                                                       10
   (print "␣␣␣calcule␣cos␣angulo")                                   11
);fim do define                                                      12

(define (main c)                                                     14
 (with-handler em-caso-de-erro                                       15
  (let* ( (f (cadr c))                                               16
          (r (string→number (caddr c)) )                             17
          (x (/ (* r pi) 180) )                                      18
        );fim da lista de variáveis                                  19

     (cond     ( (equal? f "sin"); condição                          21
                 (print "(sin␣" x ")=␣" (sin x))     )               22
               ( (equal? f "cos");                                   23
                 (print "(cos␣" x ")=␣" (cos x))     )               24
               (else (print "Não␣sei␣calcular␣" c)  )                25
     );fim do cond                                                   26
  );fim do let                                                       27
 );fim do handler                                                    28
);fim do define                                                      29
```

No programa da página 128, não definimos o que fazer se o usuário cometer um erro. Exemplos de erros:

```
calcule sin zeca
calcule cos
```

No primeiro caso, `"zeca"` não é um número. No segundo caso, o usuário esqueceu-se de fornecer o argumento de **cos**. Em outras linguagens, é necessário examinar cuidadosamente cada possibilidade de erro e tomar as providências cabíveis. Em Scheme, pode-se usar a forma

```
(with-handler
  nome-de-uma-função-que-cuida-do-erro
  (comando-1)
  (comando-2)
  ...
); fim do handler
```

A função que cuida do erro tem um argumento, que é a mensagem de erro. Você deve definir esta função. Na página 129, o erro é tratado pela função `em-caso-de-erro`, definida nas linhas 7, 8, 9, 10, 11 e 12. Na linha 16, observe que usamos `let*`, em vez de `let`; isto possibilita o uso do valor de uma variável no cálculo das variáveis que a seguem. Na página 129, usamos o valor de `r`, calculado na linha 17, para obter o valor de `x`, que é o ângulo em radianos, e cujo cálculo aparece na linha 18.

A forma `cond` é poderosa e simples, mas é preciso fazer muitos exercícios para dominá-la. Ela tem um número indefinido de argumentos. Cada argumento tem a forma

```
(cond (  (> x y) (comando-1) (comando-2)...etc. )
      (  (< x y) (comando3) (comando-4)...etc. )
      (  (equal? x y) (comando-5) (comando-6)...etc.)
      (  else (comando-7) (comando-8)...etc.))
```

Muitos programadores preferem usar parênteses quadrados para agrupar cada condição com os comandos que devem ser executados se ela for verdadeira. Neste caso, a forma `cond` fica assim:

```
(cond [  (> x y) (comando-1) (comando-2)...etc.]
      [  (< x y) (comando3) (comando-4)...etc.]
      [  (equal? x y) (comando-5) (comando-6)...etc.]
      [  else (comando-7) (comando-8)...etc.)]
```

Muitos profissionais consideram Scheme muito fácil, e não fazem exercícios de treinamento. É incrível observar de quantas maneiras estas pessoas erram o `cond`. Vamos escrever a função fatorial utilizando `cond`.

```
(module factorial (main start))                                    1

(define (factorial n)                                              3
  (cond [ (< n 2) 1 ]                                              4
        [else  (* n (factorial (- n 1)) ) ]                        5
   );end-cond                                                      6
);end-define                                                       7

(define (start c)                                                  9
  (let [ (n (string→integer (cadr c))) ]                           10
    (print n "!=" (factorial n))                                   11
  );end-let                                                        12
);end-define                                                       13
```

A maneira mais comum de errar é não agrupar cada condição com suas ações. Em resumo, os profissionais esquecem-se de usar parênteses para associar condição com comandos. Eis como ficaria a definição de fatorial:

```
(define (factorial n)
  (cond  (< n 2) 1
        else  (* n (factorial (- n 1)) )
   );end-cond
);end-define
```

Outro erro muito comum é esquecer-se de que todo comando Scheme é uma lista cujo primeiro elemento é uma função. Neste caso, o profissional escreve a condição em forma infixa:

```
(define (factorial n)
  (cond [ (n<2) 1 ]
        [ else  (* n (factorial (- n 1)) ) ]
   );end-cond
);end-define
```

Os mesmos erros que cometem no `cond`, muita gente comete também no `let`. Para evitar este tipo de erro, é preciso fazer muitos exercícios. Um bom exercício para treinar o `cond` é escrever um programa que resolva a equação do segundo grau. A solução está dada a seguir.

```
;Compile: bigloo eq.scm -o eq.exe

(module baskara (main main))

(define (eq2 sa sb sc)
   (let* [ (a (string->number sa) )
           (b (string->number sb) )
           (c (string->number sc) )
           (delta (- (* b b) (* 4 a c)))  ]

      (cond [ (< delta 0)
              (print "Delta negativo")
              (print "Não existem raízes reais.")
            ];fim da primeira condição
            [ (= delta 0)
              (print "Temos apenas uma raiz:")
              (print (/ (- b) 2 a))
            ];fim da segunda condição
            [ else
                (print "Temos duas raízes:")
                (print (/(+ (- b)
                            (sqrt delta)) 2 a) )
                (print (/ (- (- b)
                             (sqrt delta)) 2 a) )
            ]
        );fim do cond
    );fim do let
);fim do define


(define (main c)
  (with-handler  display
     (eq2 (cadr c) (caddr c) (cadddr c) ) ) )
```

5.2 Repetições

Para repetir uma sequência de comandos, podemos usar uma versão rotulada do `let`. Eis um programa para calcular uma tabela de senos:

```
(module senos (main main) )                                          1

(define 2pi (* 8 (atan 1)))                                          3

(define (tabela dg dr)                                               5
  (let loop (  (i 0) (r 0) )                                         6
     (print "sin " i "= " (sin r))                                   7
     (when (< i 360)                                                 8
         (loop (+ i dg) (+ r dr)  )                                  9
     )                                                               10
  )                                                                  11
)                                                                    12

(define (main c)                                                     14
   (with-handler display                                             15
     (let* [ (n (string->number (cadr c)))                           16
             (dg (/ 360 n))                                          17
             (dr (/ 2pi n))  ]                                       18
      (tabela dg dr )                                                19
     );end-let                                                       20
   );end-handler                                                     21
);end-define                                                         22
```

Na linha 6, as variáveis `i` e `r` são inicializadas com 0. Na linha 8, enquanto $i < 360$, os programa volta para `loop`, incrementando `i` de `dg` e `r` de `dr`. Note que o nome do `let` não precisa ser `loop`. Poderia ser `de-novo`, `outra-vez`, `again`, etc.

```
(define (tabela dg dr)
  (let de-novo (  (i 0) (r 0) )
     (print "sin " i "= " (sin r))
     (when (< i 360)
         (de-novo (+ i dg) (+ r dr)  )
) )  )
```

Agora, vamos usar o let rotulado para definir a função de Fibonacci. Considerando que `f0=1` e `f1=1`, o `let` é inicializado como mostra a linha 4. Na linha 6, temos que `fi+1= (+ fi-1 fi)`. Quando `i=n`, a repetição termina e a resposta está em `fi` (ver linha 5).

```
(module fibonacci (main main))                                    1

(define (fib n)                                                   3
  (let next ( (i 1) (fi-1 1) (fi 1))                              4
     (if (≥ i n) fi                                               5
        (next (+ i 1) fi (+ fi-1 fi) ) )) )                       6

(define (main c)                                                  8
  (with-handler display                                           9
     (let [ (n (string→number (cadr c))) ]                        10
        (print "fib␣" n "=␣" (fib n))                             11
     )                                                            12
  )                                                               13
)                                                                 14
```

5.3 Números grandes

Muitas vezes, precisamos efetuar cálculos com números enormes. O programa abaixo mostra como modificar o cálculo do fatorial para tratar de números grandes, como o fatorial de 100.

```
(module bigfat (main main))                                       1

(define (fat n)                                                   3
   (if (<bx n #z2) #z1                                            4
      (*bx n (fat (-bx n #z1))) ))                                5

(define (main c)                                                  7
  (with-handler display                                           8
  (let [ (n (string→bignum (cadr c))) ]                           9
     (print (bignum→string (fat n)) )) ))                         10
```

5.4 Programas iterativos

Iteração significa repetição, em latim. Um programa é iterativo quando a chamada que provoca a repetição não é argumento de nenhuma função. Você deve tentar usar programas iterativos sempre que possível, pois são mais eficientes. Agora examinemos uma série de exercícios publicados na internete.

L1. Encontrar o último cdr de uma lista. Isto significa que você deve encontrar uma lista contendo o último elemento de outra lista. Por exemplo, no caso da lista `(3 4 5 6)`, o último cdr é `(6)`.

```
(module last (main main))

(define (my-last c)
  (let loop ( (xs c) )
     (cond ( (not (pair? xs)) '())
           ( (null? xs) xs)
           ( (null? (cdr xs)) xs)
           (else (loop (cdr xs)) ) )))

(define (main c)
  (print "Digite a lista")
  (print (my-last (read))) )
```

L2. Encontrar o penúltimo cdr de uma lista.

```
(module penult (main main))

(define (penultimo c)
  (let loop ( (xs c) )
     (cond ( (not (pair? xs)) '())
           ( (null? xs) xs)
           ( (null? (cdr xs)) xs)
           (else (loop (cdr xs)) ) )))

(define (main c)
  (print "Lista: ") (print (penultimo (read))) )
```

L3. Encontrar o enésimo elemento de uma lista.

```
(module nth (main main))


(define (nth xs k)
   (let loop ( (s xs)(i k) )
      (cond ( (null? s) s)
            ( (<= i 1) (car s))
            (else (loop (cdr s) (- i 1))  )
      )
   )
)

(define (main c)
  (print "Digite a lista:")
  (let ( (xs (read) ) )
     (print "Digite n: ")

     (print (nth xs (read)) )
  )
)
```

A propósito, a função pré-definida **list-ref** é equivalente a **nth**.

L4. Encontrar o número de elementos de uma lista.

```
(module length (main main))

(define (len xs)
   (let loop ( (s xs)(i 0) )
      (cond ( (not (pair? s)) i)
            (else (loop (cdr s) (+ i 1))  ))  ))

(define (main c)
  (print "Digite a lista:")
  (print (len (read)) )
)
```

L5. Inverter uma lista.

```
(module reverse (main main))

(define (rev xs)
  (let loop ( (s xs) (acc '()) )
     (cond ( (not (pair? s)) acc)
           (else (loop (cdr s) (cons (car s) acc) ) )
     )
  )
)

(define (main c)
  (print "Lista: ")
  (print (rev (read))) )
```

L6. Descobrir se uma lista é um palíndrome, ou seja, como explicado anteriormente, se é uma frase que pode ser lida da esquerda para a direita ou da direita para a esquerda.

```
(module palin (main main))

(define (rev xs)
  (let loop ( (s xs) (acc '()) )
     (cond ( (not (pair? s)) acc)
           (else (loop (cdr s) (cons (car s) acc)) ))  ))

(define (palin xs)
  (equal? xs (rev xs)))

(define (main c)
  (print "Lista: ")
  (print (palin (read))) )
```

5.5 Programas recursivos

Quando você precisa de construir listas, mantendo a ordem em que os elementos são visitados pelo programa, a melhor solução é a recursividade. Isto significa que seu programa vai fazer chamadas a si mesmo.

L7. Escreva um programa que achate listas.

```
D:\scheme-docs\notsoshort\exemplos>flatten
Lista: (a (b (c d) e))
(a b c d e)
```

Eis aqui o programa:

```
(module flatten (main main))

(define (flatten s)
  (cond ( (not (pair? s)) s)
        ( (pair? (car s))
          (append (flatten (car s)) (flatten (cdr s))))
        (else (cons (car s) (flatten (cdr s)) ))  ))

(define (main c)
   (display "Lista: ")  (print  (flatten (read))) )
```

L8. Eliminar duplicatas consecutivas de uma lista de elementos. A ordem dos elementos deve ser mantida.

```
(module dup (main main))

(define (eldup xs)
   (cond ( (not (pair? xs)) xs)
         ( (not (pair? (cdr xs))) xs)
         ( (equal? (car xs) (cadr xs))  (eldup (cdr xs)) )
         (else (cons (car xs) (eldup (cdr xs)))  )))

(define (main c)
  (print "Lista: ") (print (eldup (read))) )
```

L9. Empacotar elementos contíguos repetidos de uma lista em sublista.

```
(module agrupar (main main))

(define (ag xs)
  (let loop ( (s xs) (x (car xs)) (acc '()))
     (cond ( (null? s) (cons acc s))
           ( (equal? (car s) x) (loop (cdr s) x (cons x acc)) )
           ( (cons acc (loop (cdr s) (car s) (list (car s)) ) ) ) )
     )
  )
)

(define (main c)
  (print "Lista: ") (print (ag (read))) )
```

L10. Um dos sistemas mais simples de compactação de dados é o *código de comprimento*. Se a lista tem a forma `(a a a a b b b c c d d d e)`, ela é transformada em `((4 a)(3 b)(2 c)(3 d) (1 e))`. Use o programa anterior para escrever um codificador de comprimento corrido.

```
(define (ag xs)
  (let loop ( (s xs) (x (car xs)) (acc '()))
     (cond ( (null? s) (cons acc s))
           ( (equal? (car s) x) (loop (cdr s) x (cons x acc)) )
           ( (cons acc (loop (cdr s) (car s) (list (car s)) ) ) ) )
     )
  )
)

(define (cod xs)
  (let loop ( (s (ag xs)) )
   (cond ( (null? s) s)
         ( (cons (list (length (car s)) (caar s))
                 (loop (cdr s)) ) )
   )
  )
)
```

L11. Modifique o resultado do problema **L10** de tal maneira que um elemento sem duplicatas é simplesmente copiado na lista resultante. Apenas elementos com duplicatas são colocados na forma (N E).

```
(module codificar (main main))

(define (ag xs)
  (let loop ( (s xs) (x (car xs)) (acc '()))
     (cond ( (null? s) (cons acc s))
           ( (equal? (car s) x) (loop (cdr s) x (cons x acc)) )
           ( (cons acc (loop (cdr s) (car s) (list (car s)) ) ) )
     )
  )
)

(define (cod xs)
  (let loop ( (s (ag xs)) )
   (cond ( (null? s) s)
         ( (equal? (length (car s)) 1)
           (cons (caar s) (loop (cdr s)) ))
         ( (cons (list (length (car s)) (caar s))
                 (loop (cdr s)) ) ) ) ) )

(define (main c)
  (print "Lista: ") (print (cod (read))) )
```

L12. Implemente o método de compressão com codificador de comprimento diretamente, i.e., não crie as sublistas.

```
(define (cns n x)
  (if (= n 1) x (list n x)))

(define (cod xs)
  (let loop ( (s xs) (x (car xs))  (acc 0))
     (cond ( (null? s) (list (cns acc x)))
           ( (equal? (car s) x) (loop (cdr s) x (+ acc 1)) )
           ( (cons (cns acc x) (loop (cdr s) (car s) 1 ) ) )
) )  )
```

L13. Dado uma lista comprimida com o codificador de comprimento, construa a lista descompactada.

```
(module codificar (main main))

(define (cns n x)
  (if (= n 1) x
      (list n x)))

(define (cod xs)
  (let loop ( (s xs) (x (car xs))  (acc 0))
     (cond ( (null? s) (list (cns acc x)))
           ( (equal? (car s) x) (loop (cdr s) x (+ acc 1)) )
           ( (cons (cns acc x) (loop (cdr s) (car s) 1 ) ) )
     )
  )
)

(define (ncons n x xs)
   (let loop ( (i n)  (acc xs) )
      (if (< i 1) acc
         (loop (- i 1) (cons x acc)) )))


(define (decod s)
   (cond ( (null? s) s)
         ( (not (pair? (car s)))
           (cons (car s) (decod (cdr s)) ))
         (else (ncons (caar s) (cadar s)
                   (decod (cdr s)) ))
   )
)

(define (main c)
  (print "Lista: ")
  (let ( (n (read)) )
     (print (cod n))
     (print (decod (cod n)))
  )
)
```

L14. Escreva um programa que duplique os elementos de uma lista. E.G.:

```
(dup-elementos '(a b c d e)) => (a a b b c c d d e e)
```

```
(define (dup-elementos xs)
   (cond ( (null? xs) '())
         (else (cons (car xs)
                     (cons (car xs)
                        (dup-elementos (cdr xs)) ))
         )
    )
)
```

L15. Escreva um programa que insere um elemento na posição n de uma lista dada. Por exemplo, `(insn 'a 3 '(1 2 3 4 5)) => (1 2 a 3 4 5)`.

```
(define (insn elem n xs)
   (cond  ( (= n 1) (cons elem xs))
          ( (null? xs) '() )
          (else (cons (car xs)
                      (insn elem (- n 1) (cdr xs)))
          )
    )
)
```

L16. Escreva um programa que construa a lista dos inteiros em uma dado intervalo. `(iotab 3 9) => (3 4 5 6 7 8 9)`

```
(define (iotab a b)
  (if (> a b) '()
    (cons a (iotab (+ a 1) b)
    )
  )
)
```

Números primos são muito importantes em criptografia. Vamos, então, escrever alguns programas sobre números primos.

L17. Escreva um programa que determina se um dado inteiro é primo.

```
(define (has_factor? N L)
     (cond ( (zero? (remainder N L)) )
           (else (and (< (* L L) N)
                      (has_factor? N (+ L 2))) )) )

(define (is_prime x)
  (cond ( (equal? x 2) )
        ( (equal? x 3) )
        (else (and (integer? x) (> x 3)
                   (not (zero? (remainder x 2)))
                   (not (has_factor? x 3))  ))  ))
```

L18. Determine o maior divisor comum de dois inteiros positivos.

```
(define (my-gcd X Y)
   (cond ( (zero? Y) X)
         ( (> Y 0) (my-gcd Y (remainder X Y)) )) )
```

L19. Determine se dois inteiros positivos são co-primos. Dois inteiros são co-primos se o maior divisor comum deles é 1.

```
(define (coprime X Y)
  (equal? 1 (my-gcd X Y)))
```

L20. A conjectura de Goldbach diz que todo número par positivo maior que 2 é a soma de dois números primos. Escreva uma função que, dado um número par, encontre os dois primos da conjectura.

```
(define (range start end)
  (if (> start end)
      '()
      (cons start
            (range (+ start 1) end))))
```

```
(define (filter pred li)
  (if (null? li)
      '()
      (if (pred (car li))
          (cons (car li)
                (filter pred (cdr li)))
          (filter pred (cdr li)))))


(define (primes max)
  (let loop ((sieve-list (range 2 max)))
    (if (> (car sieve-list) (sqrt max))
        sieve-list
        (cons (car sieve-list)
              (loop
               (filter
                (lambda (x)
                  (not
                   (= (remainder x (car sieve-list)) 0)))
                (cdr sieve-list)))))))

(define (goldbach-pairs num)
  (define (all-pairs li)
    (let top-loop ((top-list li) (acc '()))
      (if (null? top-list)  acc
          (let sub-loop ((sub-list top-list) (sub-acc acc))
            (cond ((or (null? sub-list)
                       (> (+ (car top-list)
                             (car sub-list)) num))
                   (top-loop (cdr top-list) sub-acc))
                  ((= (+ (car top-list) (car sub-list)) num)
                   (sub-loop (cdr sub-list)
                             (cons (cons (car top-list)
                                         (car sub-list))
                                   sub-acc)))
                  (else
                   (sub-loop (cdr sub-list) sub-acc))))))))

  (if (not (= (remainder num 2) 0))   '()
      (all-pairs (primes num))))
```

5.6 Match

As linguagens funcionais modernas, como Clean, ML, Erland e Haskell possuem casamento de padrão. LISP não pode ser chamada de moderna, visto que é a linguagem mais antiga que ainda está sendo utilizada. Entretanto, LISP é tão flexível que adaptou-se às mudanças do tempo. Considere, por exemplo, a definição abaixo:

```
(define (app xs ys)
  (match-case xs
    ( (?x . ?s) (cons x (app s ys)))
    ( () ys)))
```

Como você pode ver, existem padrões para selecionar a cabeça e a cauda de uma lista. O padrão `(?x . ?s)` unifica-se com qualquer lista contendo mais de um elemento; quando isto acontece, a variável `x` assume o valor da cabeça da lista, enquanto `s` assume o valor da cauda (o que sobra da lista com a remoção da cabeça). Consideremos um caso concreto. Quando `(?x . ?s)` unifica-se com `(5 2 6 3)`, `x` recebe o valor `5` e `s` recebe `(2 6 3)`, que é a cauda da lista. Façamos um teste:

Listing 5.1: Exemplo de padrão

```
(module duplica (main main))                                    1

(define (app xs ys)                                             3
  (match-case xs                                                4
    ( (?x . ?s) (cons x (app s ys)))                            5
    ( () ys)))                                                  6

(define (main argv)                                             8
  (print "Digite uma lista:")                                   9
  (let ( (xs (read)) )                                          10
     (print (app xs xs))                                        11
  )                                                             12
)                                                               13
```

Na listagem 5.1, tenha cuidado para inserir espaços em volta do ponto de `(x . y)`. Todos os símbolos léxicos do LISP estão rodeados de espaço.

Como já vimos antes, para avisar LISP de que não desejamos avaliar uma expressão, usamos um apóstrofo como prefixo; por exemplo: `'(era uma vez)`. Muitas vezes, porém, queremos avaliar parte da expressão; neste caso, usamos o quase apóstrofo; exemplo:

```
`(a b ,(3 4 5) c) => (a b 60 c)
`(a b ,@(append '(3 4 5) '(6 7 8)) c) => (a b (3 4 5 6 7 8) c)
```

Podemos usar o quase apóstrofo para definir app; a vírgula força a avaliação da expressão que a segue.

```
(define (app xs ys)
  (match-case xs
    ( (?x . ?s) `(,x . ,(app s ys)))
    ( () ys)))
```

As linguagens funcionais modernas, como Clean, Haskell e ML, adotam um estilo de programação fortemente apoiado na unificação. Embora LISP não encoraje a unificação, sua linguagem de padrões é mais flexível e poderosa do que a de Haskell e Clean. Entre outras coisas, há duas maneiras de combinar padrões:

`(and pat1 pat2 pat3...)`	A combinação `and` tem sucesso se todos os padrões unificam-se com a estrutura de dados.
`(or pat1 pat2 pat3...)`	A combinação `or` tem sucesso se um dos padrões unifica-se com a estrutura de dados.

Outras linguagens funcionais unificam-se pela igualdade. LISP permite o uso de qualquer predicado para assegurar a unificação. Por exemplo, se você quer verificar se o primeiro elemento de uma lista é inteiro, e unificá-lo com uma variável em caso verdadeiro, pode usar o seguinte padrão:

```
((and (? integer?) ?x) . ?xs)
```

Vamos definir uma função para filtrar os elementos não inteiros de uma lista.

```
(define (ints s)
  (match-case s
     ( ((and (? integer?) ?x) . ?xs) `(,x . ,(ints xs)))
     ( (?x . ?xs) (ints xs))
     ( ( ) '( )) ))
```

Um padrão um tanto quanto inútil é `(not pat)`. Vamos utilizá-lo para limpar uma lista de números.

```
(define (numbers s)
   (match-case s
      ( ( (and  (not (? number?)) ?x) . ?xs)
          (numbers xs) )
      ( (?x . ?xs) (cons x (numbers xs)) )
      ( () '() )
    )
)
```

5.7 Padrões com elipses

Os padrões podem ter elipses, que são representadas por três pontinhos. Apresentamos abaixo um programa que lê os coeficientes de uma equação do segundo grau.

```
(module inteiros)                                          1

(define (nums? xs)                                         3
    (match-case xs                                         4
       ( ( (? number?) ...) xs)                            5
       (?any #f)))                                         6

(define (notzero? a) (not (= a 0)))                        8
```

```
(define (complex x y)                                                    10
  (list x y 'i) )                                                        11

(define (eq2 abc)                                                        13
  (match-case abc                                                        14
    ( (0 ?b ?c) "Não é equação do segundo grau")                         15
    ( (?a ?b ?c)                                                         16
      (let* ( (delta (- (* b b) (* 4 a c)))                              17
              (d (sqrt (abs delta)) )                                    18
              (-b/2a (/ (- b) 2 a) )                                     19
              (d/2a (/ d 2 a))                                           20
              (-d/2a (- d/2a))                                           21
              (2a (/ 2 a)) )                                             22
        (if (< delta 0)                                                  23
            (list (complex -b/2a d/2a)                                   24
                  (complex -b/2a -d/2a))                                 25
            (list (+ -b/2a d/2a)                                         26
                  (- -b/2a d/2a) ))  ))                                  27
    ( ?- "Dados incorretos.") ))                                         28

(print "Entre com a, b, c entre parênteses")                             30

(print (eq2 (nums? (read))))                                             32
```

5.8 Entrada e saída

Scheme tem um sistema de entrada e saída que é ao mesmo tempo conveniente e produtivo. Entrada e saída são realizadas por meio de portais. Portais são dispositivos que podem ser associados com arquivos, consoles e *strings*.

Lendo dados de um portal. A melhor maneira de entender o funcionamento dos processos de entrada e saída do Scheme é testando-o em um

intérprete. Até agora, temos usado o Bigloo, que é excelente quando escrevemos programas compilados. No caso de programas interpretados, existem sistemas mais rápidos (ver [PLT]). No teste abaixo, abrimos uma *string* para entrada e saída, e usamos `read` com a finalidade de ler duas estruturas da *string*.

```
Linguagem: Module; memory limit: 128 megabytes.

> (define st "327 (Lili Marlene)") ;; define uma string
> st
"327 (Lili Marlene)"
> (define pin (open-input-string st)) ;; abrir um input port
> (read  pin)
327
> (read pin)
(Lili Marlene)
> (read pin)
#<eof>
>
```

Escrevendo em um portal. No exemplo acima, abrimos uma *string* para ler seu conteúdo. Podemos também escrever em uma *string*; aliás, um dos métodos mais fáceis de montar *string*s é escrevendo nelas.

```
Linguagem: Module; memory limit: 128 megabytes.

> (define ws (open-output-string))
> (display "Think!" ws)
> (write "Le Penseur" ws)
> (newline ws)
> (get-output-string ws)
"Think!\"Le Penseur\"\n"
> (close-output-port ws)
>
```

Entrada e saída padrão. Em geral, se lemos alguma coisa da entrada e saída padrão, a função leitora não recebe argumento representando o portal.

A mesma coisa acontece com a escrita, ou seja, a função de saída não precisa de um argumento para representar o portal padrão. Vamos ver um exemplo.

```
> (let ( (x (read)))
    (display "Square: ")
    (display (* x x))
    (newline)
    (display "sin(x)= ")
    (display (sin x))
    (newline))
1.57
Square: 2.4649
sin(x)= 0.9999996829318346
>
```

Lendo e escrevendo em arquivo. Em Scheme, ler coisas de arquivos ou escrever em arquivo é tão fácil como fazê-lo em *string*.

```
> (define pout (open-output-file "d:/lixo/scratch.txt"))
> (write "Hello, World!" pout)
> (newline pout)
> (display (/ 3.1416 2) pout)
> (close-output-port pout)
> (define ip (open-input-file "d:/lixo/scratch.txt"))
> (read ip)
"Hello, World!"
> (read ip)
1.5708
> (read ip)
#<eof>
> (close-input-port ip)
>
```

Fechamento automático de portal. As funções `call-with-input-file` e `call-with-output-file` são muito convenientes, pois elas cuidam da abertura e do fechamento de um portal, quando terminamos de usá-lo. O procedimento `call-with-input-file` tem como parâmetros um nome de arquivo (*string*) e uma expressão lambda com um único argumento. A expressão lambda é aplicada a um portal de entrada que dá acesso a um arquivo.

Quanto ao procedimento `call-with-output-file`, ele também tem dois argumentos; o primeiro entretanto é um arquivo de escrita; o segundo é uma expressão lambda que tem um portal por argumento.

```
> (call-with-output-file "scratch.txt"
    (lambda(out)
       (write "Hello, world" out) (newline out)
       (display 327 out)
       (newline out)
     )
   )

> (call-with-input-file "scratch.txt"
     (lambda(in)
        (display (read in)) (newline)
        (display (read in)) (newline)
        (display (read in)) (newline)
      )
   )

Hello, world
327
#<eof>
```

Função surda (thunk). Uma função surda (*thunk* em inglês) é uma expressão lambda sem argumentos. Funções surdas são utilizadas principalmente para atrasar a avaliação de uma expressão. Por exemplo, a função surda abaixo

```
(lambda() (display "Hello, world!"))
```

congela o procedimento `display`, que será ativado apenas depois da avaliação da função surda.

Scheme tem formas que tomam um nome de arquivo e uma função surda como parâmetros e realizam todas as operações de entrada e saída no arquivo. Em outras palavras, se acionarmos um procedimento de escrita na saída padrão, o resultado irá para o arquivo. Naturalmente, para cada procedimento de escrita ou de leitura em arquivo, existe um equivalente que

trabalha com *strings*; isso cria um método conveniente de construir *strings* e de converter uma representação em *string* para números ou listas.

```
> (with-output-to-file "scratch.txt"
    (lambda() (display "Hello\n")
        (display (* 3 4 5)) ))

> (with-input-from-file "scratch.txt"
    (lambda() (display (read))
      (newline)
      (display (read)) ))
Hello
60
> (define str (with-output-to-string
     (lambda() (write "Hello!") (display " ")
          (write "Hi, World!"))
     )
   )

> (with-input-from-string str (lambda() (read)))
"Hello!"
>
```

Observe a diferença entre `write` e `display`. O procedimento display imprime uma estrutura de dados em forma que facilita a visualização; em particular, não são impressas as aspas de uma *string*. Por outro lado, `write` imprime em forma conveniente para posterior leitura; no caso de *strings*, `write` imprime as aspas.

```
> (display "Hello, world!")
Hello, world!
> (write "Hello, world!")
"Hello, world!"
>
```

Já vimos exemplos das seguintes funções de entrada e saída: `read`, `display`, `write` e `newline`. O procedimento `read-line` permite a leitura de uma linha terminada por `#\newline`.

Em linguagens mais primitivas do que Lisp, como é o caso de Java e C, não é possível ler estruturas de dados complexas, como listas e vetores. Em Lisp, procedimentos de leitura, tais como `read`, permitem ler qualquer estrutura de dados, tanto de arquivos quanto de *strings*. Entretanto, para possibilitar a emulação de programas em C e em Java, Scheme oferece funções primitivas de entrada e saída. Uma dessas funções é `read-char`, que lê um caracter por vez. Finalmente, Scheme tem uma função para reconhecer a leitura do fim de arquivo, que é `eof-object?`.

```
> (define str (open-input-string "Hello, World!"))
> str
#<input-port:string>
> (do ( (x (read-char str) (read-char str)))
     ((eof-object? x) (display "Fim do arquivo"))
    (write x))
#\e#\l#\l#\o#\,#\space#\W#\o#\r#\l#\d#\!Fim do arquivo
>
```

5.9 Gramáticas

As gramáticas permitem que se faça a análise léxica e sintática de linguagens. Scheme possui poderosas bibliotecas que implementam gramáticas. Vamos explicar esse conceito com um exemplo. Suponha que seja necessário ler uma linha de latim botânico (uma língua utilizada pelos botânicos para descrever plantas). Naturalmente, você poderia trapacear um pouco e convencer o botânico a escrever suas descrições em forma de listas, de modo que possam ser lidas pelo procedimento `read`. Entretanto, não utilizaremos desse estratagema. Em vez disso, escreveremos uma gramática que construirá uma lista e reconhecerá os sinais de pontuação tais como são utilizados em latim, português ou inglês. Vamos elaborar nesse ponto. A sintaxe do Scheme exige que todos os símbolos léxicos (palavras e sinais de pontuação) estejam separados por espaço. Latim e português não permitem espaços entre o sinal de pontuação e a palavra que o precede. Por isso, precisamos de um analisador léxico que aja de acordo com as regras do latim.

Analisador léxico é um procedimento que separa uma *string* ou o conteúdo de um arquivo em seus componentes léxicos, ou seja, palavras e sinais de pontuação. O analisador léxico varia de língua para língua. Todas as linguagens de programação oferecem ferramentas que facilitam a construção de analisadores léxicos. Por exemplo, linguagens como C e Java têm o YACC. Scheme oferece construtores de analisadores léxicos muito mais poderosos e fáceis de usar do que o YACC.

```
;; Compile: bigloo parser.scm -o pars

(module countword (main start-here))


(define *g*
  (regular-grammar ()
   ( (+ (or #\tab #\space)) (ignore) )
   ( (+ (out #\newline #\space #\tab ",.;?!"))
        (cons 'word (the-string)))
   (#\newline 'nl)
   (#\, 'VG)
   (#\. 'PT)
   ( (in ";?!")
     (cons 'pmark (the-string))) ))


(define *gram*
   (lalr-grammar
      ( word pmark VG PT nl)
      ( words (() '())
      (  (word words) (cons word words))
      (  (pmark words) (cons pmark words))
      (  (PT words) (cons "." words))
      (  (VG words) (cons "," words)) )))

(define (start-here argsv)
    (let* ( (s (read-line))
       (pin (open-input-string s)))
       (write (read/lalrp *gram* *g* pin))))
```

Um dos dois componentes de uma gramática é o analisador léxico ou *tokenizer*. No exemplo, o *tokenizer* é armazenado na variável global `*g*`.

```
(define *g*
  (regular-grammar ()
   ( (+ (or #\tab #\space)) (ignore) )
   ( (+ (out #\newline #\space #\tab ",.;?!"))
        (cons 'word (the-string)))
   (#\newline 'nl)
   (#\, 'VG)
   (#\. 'PT)
   ( (in ";?!")
     (cons 'pmark (the-string))) ))
```

O *tokenizer* é uma gramática regular com um conjunto finito de regras. Vamos considerar a primeira regra do latim botânico.

```
( (+ (or #\tab #\space)) (ignore) )
```

Ela diz que se o leitor `(read/lalrp *gram* *g* pin)` encontrar uma ou mais ocorrências de `#\tab` ou `#\space`, ele deve ignorá-las. O sinal `+` significa que se o leitor encontrar pelo menos uma e possivelmente mais ocorrências do padrão, deve disparar a regra. Se o sinal `+` indica pelo menos uma ocorrência, o asterisco significa zero ou mais ocorrências. Vamos examinar agora a próxima regra.

```
( (+ (out #\newline #\space #\tab ",.;?!"))
     (cons 'word (the-string)))
```

Ela diz que o *tokenizer* deve construir um *token* (símbolo léxico) com o conteúdo de `(the-string)`, caso encontre um ou mais caracteres que não sejam `#\newline`, `#\space`, `#\tab`, ou `",.;?!"`. O *token* produzido por essa regra é classificado como uma `'word` pela expressão `(cons 'word (the-string))`. Vamos supor que o *token* seja a palavra FULGIDA. Nesse caso, o *token* classificado será `(word . "fulgida")`. Note que há um pontinho separando o `car` do `cdr` dessa lista. Quando a lista termina em alguma coisa diferente da

lista vazia `'()`, devemos utilizar um ponto para separar o último elemento dos outros.

As regras

```
(#\newline 'nl)
(#\, 'VG)
(#\. 'PT)
```

dizem que se o leitor encontrar os caracteres de nova linha, vírgula ou ponto final, deve produzir os *tokens* `nl`, `VG` ou `PT` respectivamente. A última regra

```
( (in ";?!")
   (cons 'pmark (the-string))) ))
```

diz que se o leitor encontrar um sinal de pontuação, ele deve produzir algo como `(pmark . "?")`.

O *tokenizer* separa um *token* do outro, e classifica as *strings* que reconhece em classes arbitrárias. Usualmente, o *tokenizer* segue uma gramática que Chomsky denominou *regular* (veja página 7).

O que aprendemos aqui sobre o *tokenizer* é suficiente para a maioria das aplicações. O próximo passo será realizar a análise sintática de uma lista de *tokens*. Como foi dito antes e será repetido aqui, se tudo que desejamos fazer é ler uma lista de palavras, não precisamos da *artilharia pesada* oferecida pelas gramáticas formais. A função `read` é suficiente para manusear listas da Lisp. Entretanto, estamos lidando aqui com regras estranhas à Lisp.

A fim de analisar uma lista de *tokens*, precisamos de uma gramática `Lalr`. As regras das gramáticas `Lalr` têm o seguinte formato:

```
(rule-label
    ((pattern1) (rewriting-expression1))
    ((pattern2) (rewriting-expression2))...)
```

A gramática que constrói listas tem uma única regra, cujo rótulo é `words` (ver na página 154). A primeira cláusula desta regra diz que, se um *parser* encontra o padrão `( )`, ele deve reescrevê-lo como uma lista vazia `'( )`.

A segunda cláusula diz que o *parser* reescreve o padrão `(word words)` como `(cons word words)`. Devemos passar tanto o *tokenizer* quanto o *parser* para a função `read/lalrp`, a fim de ler um texto na gramática dada.

Abaixo temos um exemplo que implementa uma calculadora que aceita entradas na notação convencional, utilizada em matemática elementar.

```
(module expr (main start-here))
(define *g* (regular-grammar ()
   ( (+ (or #\tab #\space)) (ignore) )
   (#\newline 'nl)
   ( (+ digit) (cons 'const (string->number (the-string))))
   (#\+ 'plus)
   (#\- 'minus)
   (#\* 'mult)
   (#\/ 'div)
   (#\( 'lpar)
   (#\) 'rpar) ))

(define *gram* (lalr-grammar
(nl plus mult minus div const lpar rpar)
(lines
   (())
   ((lines expression nl) (print "--> " expression) )
   ( (lines nl) ) )
(expression
   ( (expression plus term) (+ expression term))
   ( (expression minus term) (- expression term))
   ( (term) term))
(term
   ( (term mult factor) (* term factor))
   ( (term div factor) (/ term factor))
   ( (factor) factor))
(factor
   ( (lpar expression rpar) expression)
   ( (const) const))) )

(define (start-here argsv)
   (let* ((s (read-line) )
          (pin (open-input-string (string-append s "\n"))))
      (write (read/lalrp *gram* *g* pin) )
      (reset-eof pin) ))
```

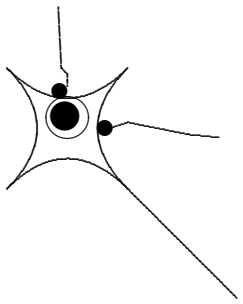
5.10 Backtracking

O termo *backtracking* refere-se a programas capazes de voltar atrás, quando entra pelo caminho escolhido leva a situações que falseiam requisitos, pré-condições, pós-condições ou invariantes. A estrutura `call/cc` permite capturar o estado da computação em qualquer ponto da execução, o que possibilita o *backtracking*. No exemplo abaixo, mostramos como utilizar *backtracking* para encontrar x, y e z tal que $x^2 = y^2 + z^2$.

```
(define fail
   (lambda () (error "no solution")))

(define enumerate
   (lambda (a b cont)
       (if (> a b)  (fail)
          (let ((save fail))
             (set! fail
                  (lambda ()
                      (set! fail save)
                      (enumerate (+ a 1) b cont)))
             (cont a)))))

(define in-range
   (lambda (a b)
      (call/cc
         (lambda (cont)
            (enumerate a b cont))  )))

(define (xyz a b)
   (let ( (x (in-range a b))
          (y (in-range a b))
          (z (in-range a b)))
      (if  (= (* x x)  (+ (* y y) (* z z)))
           (list x y z)
           (fail))))
```

Capítulo 6

Redes neurais



O cientista espanhol Don Santiago Ramón y Cajal estudou a fisiologia do sistema nervoso e concluiu que ele era mantido funcionando por células que chamou de *neurônios*. Hoje em dia, cientistas acreditam que um dado neurônio recebe sinais de outros neurônios através de conexões denominadas synapses. A propósito, *synapse* é a palavra grega para *conexão*. Tipicamente, um neurônio coleta sinais de outros neurônios através de um conjunto de estruturas denominadas dendritos, que em grego significa arborescência. Se a intensidade dos sinais de entrada ultrapassam um certo limiar, o neurônio dispara e envia um estímulo elétrico através de um cabo conhecido como axônio. Na extremidade, o axônio ramifica-se em dendritos, cujas synapses vão ativar outros neurônios. Para resumir, na extremidade de cada dendrito, uma synapse converte atividade neuronal em estímulos elétricos, que inibem ou excitam outros neurônios.

Em 1943, [Warren McCulloch] e Walter Pitts propuseram o primeiro modelo matemático para os neurônios. O neurônio deles era um dispositivo binário, com um limiar fixo. Este dispositivo recebia múltiplas entradas de synapses excitatórias, que dava a cada fonte de entrada um peso, que era um inteiro positivo. As entradas inibitórias tinham um poder de veto

sobre qualquer entrada excitatória. Em cada passo do processamento, os neurônios eram sincronicamente atualizados pela somatória ponderada das entradas excitatórias; a saída era colocada em 1 se e somente se a soma fosse maior do que o limiar. O próximo grande passo na teoria das redes neurais foi o perceptron, introduzido por Frank Rosenblatt da Universidade de Cornell em um artigo publicado em 1958.

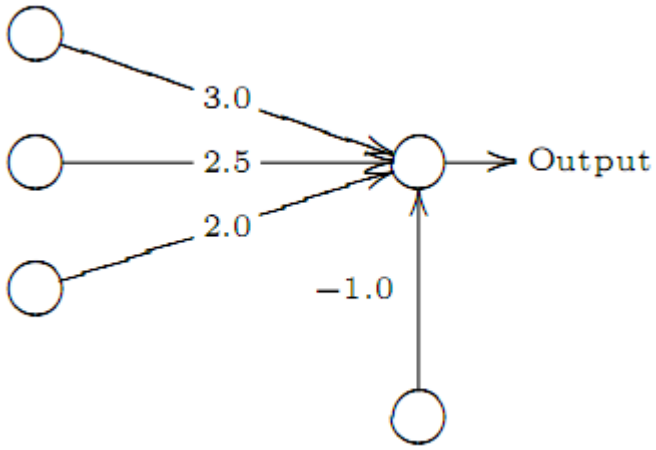
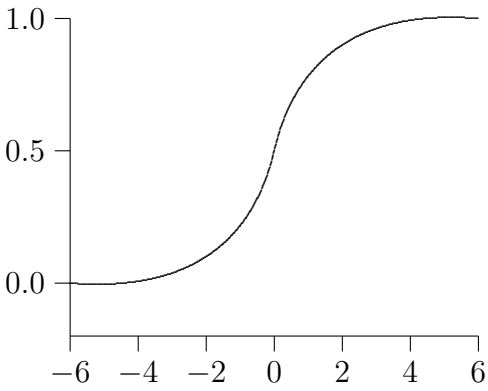


Figura 6.1: Um perceptron de três entradas



O perceptron soma os sinais de entrada multiplicados por um peso (um peso diferente para cada sinal de entrada) e, se o sinal for maior do que um limiar, ele produz uma saída diferente de zero. Este esquema foi melhorado ao alimentar uma função sigmoide com a soma ponderada, produzindo o resultado do processamento neuronal. Grafando a sigmoide, ela toma a forma da letra grega ς, que é conhecida como sigma final (daí o nome sigmoide). A sigmoide é definida como:

$$\varsigma(x) = \frac{1}{1 + e^{-x}}$$

Em Common Lisp, esta definição é escrita assim:

```
(defun sigma(x)
   (/ 1 (+ 1.0 (exp (- x)))))
```

A figura 6.1 mostra um perceptron com três entradas. Sejam x_1, x_2, e x_3 estas entradas. Neste caso, a saída será o resultado da expressão abaixo.

$$\varsigma(3x_1 + 2.5x_2 + 2.0x_3 - 1)$$

Uma característica interessante do perceptron é sua capacidade de aprendizado. Podemos treinar o perceptron para reconhecer padrões de entrada. De fato, dados exemplos em número suficiente, há um algoritmo que modifica os pesos do perceptron de modo que ele possa discriminar entre dois ou mais padrões. Por exemplo, um perceptron pode aprender a predizer quando um **and-gate** produzirá 1 e quando produzirá 0.

O algoritmo de aprendizado do perceptron deve ter um conjunto de exemplos de treinamento. Vamos admitir que o perceptron esteja aprendendo o funcionamento de um **xor-gate**. Eis aqui os exemplos:

```
(defparameter *exs*
   '#( (0 1 1) (1 0 1) (1 1 0) (0 0 0)) )
```

O perceptron tem uma rotina que utiliza a fórmula $\varsigma(\sum w_i x_i)$ para predizer o valor de saída, dado um conjunto de valores de entrada. Em Common Lisp, o procedimento `newneuron` que cria um neurônio com capacidade de predizer a saída correta é codificado conforme mostrado na página 162. O procedimento `porta` combina dois neurônios de modo a simular portas lógicas.

Os pesos são armazenados no vetor passado como argumento do criador de neurônios, ou seja, `newneuron`. Os pesos iniciais são arbitrários. Os verdadeiros pesos devem ser encontrados por meio do algoritmo de aprendizagem. O criador de neurônios `newneuron` tem dois argumentos. O primeiro é o vetor onde serão armazenados os pesos; o segundo argumento (`ws`) é uma lista dos índices do vetor que serão de fato utilizados para os pesos do neurônio que está sendo criado. Isso é necessário porque o mesmo vetor é compartilhado por vários neurônios.

```
#| Dado um vetor para armazenar pesos, e uma lista ws
   de índices, newneuron cria um neurônio. E.g.
   (let ( (v #(0 0 0 0)) )
      (newneuron v '(0 1 2)))
   produz um neurônio de duas entradas que usa v[0], v[1]
   e v[2] para armazenar seus pesos. |#

(defun sigma(x)
   (/ 1 (+ 1.0 (exp (- x)))))

(defun newneuron(v ws)
  (lambda(xs)
     (sigma
          (loop for i in ws
                for x in (cons 1.0 xs)
                sum (* (aref v i) x)))))

(defun porta(vt)
   (let ( (n1 (newneuron vt '(4 5 6)))
           (ns (newneuron vt '(0 1 2 3))) )
      (lambda(i)
          (if (null i) vt
             (funcall ns (list (car i)
                               (funcall n1
                                   (list (car i) (cadr i)))
                                   (cadr i)  )))  )))

(defparameter xor
  (porta (make-array 7 :initial-element 0.0  )))
```

Precisamos de um procedimento para obter os exemplos sem correr risco de erros devido a índices fora de escopo.

```
(defparameter *examples* #( (0 1 1) ) )

(defun egratia(eg)
   (aref *examples*
     (min eg (length *examples*) ) ))

(defparameter DX 0.01)
```

```
(defparameter LC 0.1)

(defparameter *nuweights*  (make-array 90)  )

(defun assertWgt(vt I R)
   (setf  (aref vt I) R)   R)
```

O algoritmo de aprendizagem é baseado no cálculo e correção do erro cometido pela função de predição. O erro de um dado conjunto de elementos é calculado pela seguinte função:

```
(defun setWeights(vt Qs)
  (loop for i from 0 to (- (length vt) 1) do
      (setf (aref vt i) (aref Qs i)) ) )

(defun errSum(prt Exs)
   (loop for e in Exs sum
       (let* ( (eg (egratia e))
               (vc (funcall prt (cdr eg )))
               (v (car eg)) )
            (* (- vc v) (- vc v)) )))
```

Os pesos serão atualizados através do método do gradiente descendente, de tal maneira a reduzir o erro a um valor mínimo.

```
(defun updateWeights (prt vt err0 ns Exs )
    (loop for i from 0 to ns
          for v across vt do
      (assertWgt vt i (+ v DX))
      (let ((nv (+ v (/ (* LC (- err0 (errSum prt Exs)))
                        DX))
            ))
         (assertWgt vt i v)
         (setf (aref *nuweights* i) nv)
      )
    )
     (setWeights vt *nuweights*))
```

O programa termina com o algoritmo de treinamento. O procedimento `train` é simples: Testa-se o erro e, se ele estiver acima do limite estabelecido, atualizam-se os pesos e repete-se o treinamento.

```
(defun train(p exs)
   (setf *examples* exs )
   (setf *nuweights* (make-array 90))
   (setWeights  (funcall p ()) #(0 1 0 0 2 0 0 0 0))
   (loop with vt = (funcall p ())
         with es = '(0 1 2 3)
        until (< (errSum p es) 0.001) do
        (updateWeights p vt (errSum p es)
                    (- (length vt) 1) es)))

(defparameter *exems* #( (0 1 1) (1 0 1) (1 1 0) (0 0 0)))
```

O método do gradiente descendente é um algoritmo de otimização que encontra um mínimo local de uma função tomando passos proporcionais ao gradiente negativo da função no ponto corrente. Em cada passo do método do gradiente descendente o peso é atualizado de acordo com a seguinte fórmula:

$$\omega_{n+1} = \omega_n - \gamma \nabla \mathbf{error}(\omega)$$

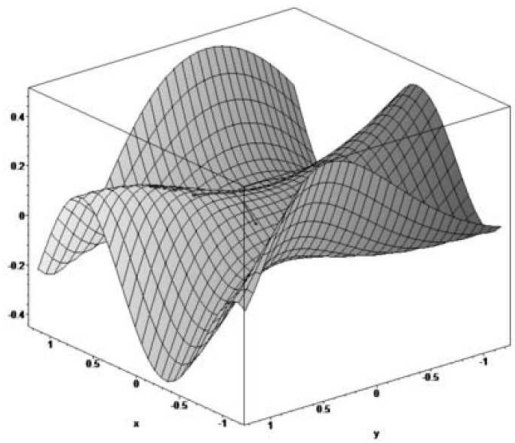


Figura 6.2: Gradiente de uma função

Nesta fórmula, se $\gamma > 0$ é um número suficientemente pequeno, $\mathbf{error}(\omega_{n+1}) < \mathbf{error}(\omega_n)$. Se começamos com ω_0, a sequência $\omega_0, \omega_1, \omega_2 \ldots$ convergirá para um mínimo. O mínimo será alcançado quando o gradiente se torna zero (ou próximo de zero, na prática).

Imagine um terreno com montanhas e vales. A posição no

terreno é dada pelas coordenadas ω_1 e ω_2; as alturas das montanhas e profundidades dos vales representam os erros. O algoritmo de treinamento move a rede de uma posição inicial para um vale, ou seja, para uma posição em que o erro atinge um mínimo.

A situação ideal ocorre quando a rede atinge um mínimo global. Entretanto, métodos de gradiente quase sempre acabam presos em mínimos locais. Isto acontece porque, quando a rede atinge um mínimo, não importa se local ou global, o gradiente aproxima-se de zero. De fato, o algoritmo sabe que atingiu um mínimo porque o gradiente chega em zero. O problema é que este critério não informa se o mínimo é global ou local. O dicionário diz que o *gradiente* fornece a inclinação de um local. No topo de uma montanha ou no fundo de um vale o gradiente é zero. Se usarmos um nível de pedreiro no topo de uma montanha, não detetaremos nenhuma inclinação.

Um neurônio como o mostrado na figura 6.3 pode ser descrito pela seguinte equação:

$$\omega_1 x_1 + \omega_2 x_2 + \omega_0 = 0$$

Esta é a equação de uma linha reta. Se usarmos mais entradas, teremos retas em um espaço multidimensionais, mas que continuarão sendo retas. Vamos grafar uma função lógica chamada *or*-gate. Ela tem duas entradas, x_1 e x_2.

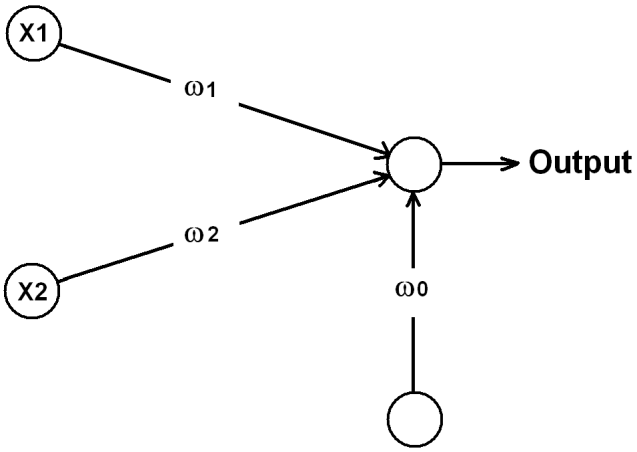
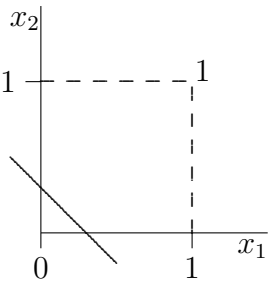


Figura 6.3: Um perceptron de duas entradas

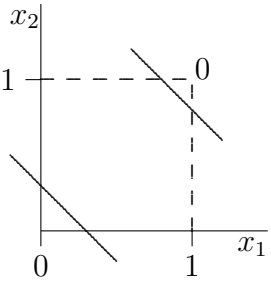


A função *or*-gate produz 1, se ou $x_1 = 1$ ou $x_2 = 1$. Ela produz 1, se tanto $x_1 = 1$ quanto $x_2 = 1$. O gráfico da figura mostra o comportamento da *or*-gate. O que a rede neural da figura 6.3 faz é desenhar uma linha reta separando os pontos onde a *or*-gate é 1 dos pontos onde é 0. Então, tudo que um perceptron pode fazer é aprender a desenhar linhas retas em um espaço de estado a fim de separar pontos com uma certa propriedade. O problema é que tais retas nem sempre existem. A *xor*-gate é muito mais útil do que a *or*-gate. Afinal, as entradas da *xor*-gate podem representar proposições tais como:

$x_1 =$ **the machine is on**; $x_2 =$ **the machine is off**
$x_1 =$ **the door is open**; $x_2 =$ **the door is closed**

Já que máquinas não podem estar ligadas ou desligadas ao mesmo tempo, $x_1 = 1$ **e** $x_2 = 1$ não pode ser verdadeira; $x_1 = 0$ **e** $x_2 = 0$ também não pode ser verdadeira. Para descrever portas lógicas, usamos tabelas de verdade que forneçam a saída para todos os valores das variáveis de entrada. Abaixo, temos uma tabela de verdade para a *xor*-gate.

x_1	x_2	output
1	1	0
1	0	1
0	1	1
0	0	0



Pode-se ver da figura que uma linha reta por si só não pode discriminar entre os valores de uma *xor*-gate. Entretanto, duas linhas retas podem separar valores 1 de valores 0, como podemos ver na figura. Da mesma forma, um simples perceptron não pode aprender a calcular as saídas de uma *xor*-gate, mas uma rede com mais de um perceptron, como a mostrada na figura 6.4, pode realizar esta tarefa.

Um ponto importante é que as redes neurais não são únicas. Isso quer dizer que poderíamos utilizar uma configuração de neurônios diferente da mostrada na figura 6.4 para representar o `xor`. De fato, no programa em Common Lisp da página 162 foram utilizados apenas dois neurônios para implementar o `xor`.

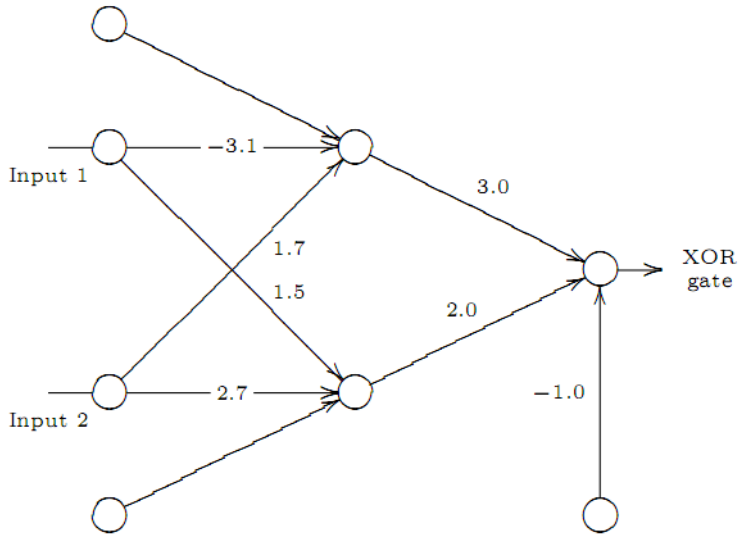


Figura 6.4: Rede neural que pode aprender como calcular uma xor-gate

6.1 Linguagem de domínio específico

O programa da página 162 começa criando uma linguagem de domínio específico para redes neurais. Em uma linguagem com raízes menos profundas na matemática e na lógica, a tarefa de criar uma linguagem de domínio específico seria bastante penosa. Em Lisp, contudo, ela se torna trivial. Em um livro de grande influência, [Paul Graham] diz:

> *Because Lisp gives you the freedom to define your own operators, you can mold it into just the language you need. If you are writing a text-editor, you can turn Lisp into a language for writing text-editors. If you are writing a CAD program, you can*

> *turn Lisp into a language for writing CAD programs. And if you are not sure yet what kind of program you are writing, it is a safe bet to write it in Lisp. Whatever kind of program yours turns out to be, Lisp will, during the writing of it, have evolved into a language for writing that kind of program.*
>
> *If you are not sure yet what kind of program you are writing? To some ears that sentence has an odd ring to it. It is in jarring contrast with a certain model of doing things wherein you (1) carefully plan what you are going to do, and then (2) do it. According to this model, if Lisp encourages you to start writing your program before you have decided how it should work, it merely encourages sloppy thinking.*
>
> *Well, it just ain't so. The plan-and-implement method may have been a good way of building dams or launching invasions, but experience has not shown it to be as good a way of writing programs. Why? Perhaps it is because computers are so exacting. Perhaps there is more variation between programs than there is between dams or invasions. Or perhaps the old methods don't work because old concepts of redundancy have no analogue in software development: If a dam contains 30% too much concrete, that's a margin for error, but if a program does 30% too much work, that is an error.*

Paul Graham estava falando em Common Lisp. Evidentemente as mesmas conclusões são válidas para outras formas de Lisp, como é o caso da Scheme. Estando mais próxima da matemática, a Scheme adapta-se melhor às manipulações lógicas necessárias para construir linguagens de domínio específico do que a Common Lisp. De fato, os programas em Scheme são mais limpos do que em Common Lisp. Scheme não exige prefixar com `funcall` aplicações de funções criadas pelo próprio programa. Veremos mais sobre isso no próximo capítulo.

6.2 Fisiologia dos neurônios

Uma das características definidoras de vida é a existência de uma fronteira separando o ser vivo de seu ambiente. No caso da célula, uma membrana dupla de lipídios circunda e preserva o protoplasma, separando-o do meio extracelular. A propósito, lipídio é a palavra grega para gordura.

O texto que segue foi escrito pensando em células de animais superiores, ou seja, animais com sistema nervoso. Talvez algumas das afirmações do texto sejam válidas também para outros seres vivos, como as plantas, mas estamos interessados apenas em animais superiores.

Alguns sais, presentes no corpo dos animais superiores, dissolvem-se nos fluidos intracelular e extracelular e dissociam-se em íons positivos e negativos. Cloreto de sódio, por exemplo, dissocia-se em Na^+ (íon de sódio) e Cl^- (íon cloreto). Outros íons positivos presentes no interior das células de animais e no espaço extracelular são o potássio (K^+) e o cálcio (Ca^{2+}). As membranas das células exibem diferentes graus de permeabilidade para cada um desses íons. A permeabilidade é determinada pelo tamanho e pelo número de poros denominados canais iônicos, que são macromoléculas com formas e cargas que permitem a passagem seletiva de alguns íons, enquanto bloqueiam outros. Resumindo, os canais iônicos são seletivamente permeáveis a íons de sódio, potássio e cálcio.

A permeabilidade específica da membrana leva a diferentes distribuições de íons no interior e no exterior da célula; em consequência disto, o interior dos neurônios é negativamente carregado com relação ao fluido extracelular. Para entender isso, suponhamos que, no interior da célula, haja um número igual de íons positivos e de íons negativos. Inicialmente não há íons no espaço extracelular. Átomos e moléculas têm a tendência de se distribuir de maneira homogênea por um processo chamado difusão. Se a parede celular for permeável apenas a íons de potássio positivos, estes sofrerão difusão para fora da célula através dos canais seletivos a íons de potássio. O processo de difusão ocorrerá até que a repulsão eletrostática dos íons positivos no exterior da célula compense o potencial de difusão. No final do processo, o interior da célula ficará carregado negativamente. A diferença de potencial é dada

pela equação de Nernst:

$$E = k(\ln(c_o) - \ln(c_i))$$

onde c_i é a concentração de potássio no interior da célula, c_o é a concentração no líquido extracelular e k é uma constante de proporcionalidade. Para íons de potássio, o potencial de equilíbrio é $-80mV$. Com uma mistura de íons do tipo que ocorre na célula, o potencial de equilíbrio é algo em volta de $-70mV$.

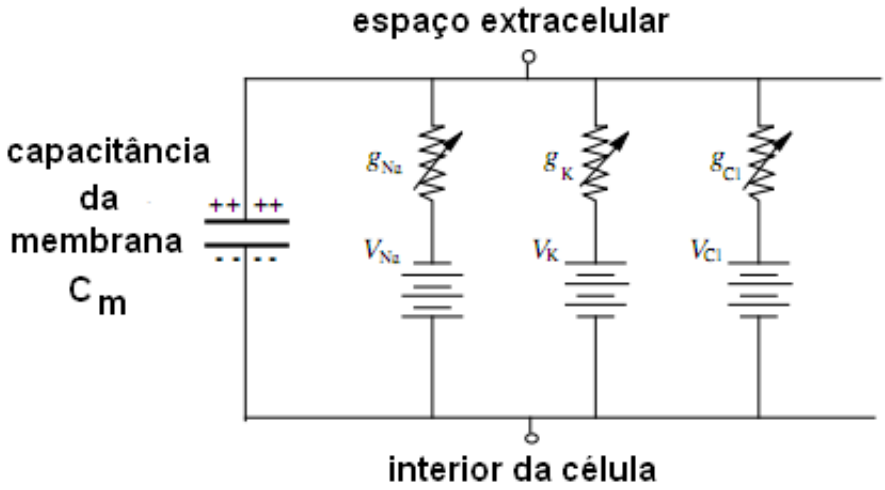


Figura 6.5: Modelo de Hodgkin-Huxley

[Alan Hodgkin] e Andrew Huxley propuseram o modelo de neurônio da figura 6.5. A capacitância é produzida pela dupla camada de fosfolipídeos. A permeabilidade específica para cada íon é modelada como condutâncias. Na figura, estão representadas as permeabilidades do potássio, do sódio e do cloro. Um sinal pode ser produzido modificando a polaridade da célula através de mudanças nas condutâncias g_{Na} e g_K. Suponhamos que no exterior da célula haja íons de sódio e que no interior, de potássio. Aumentando g_{Na} e tornando a mobilidade dos íons de sódio maior do que a do potássio, a polaridade da célula muda de $-70mV$ para $58mV$ (porque os íons de sódio entram na célula). Depois, quando a condutância g_K torna-se maior do que

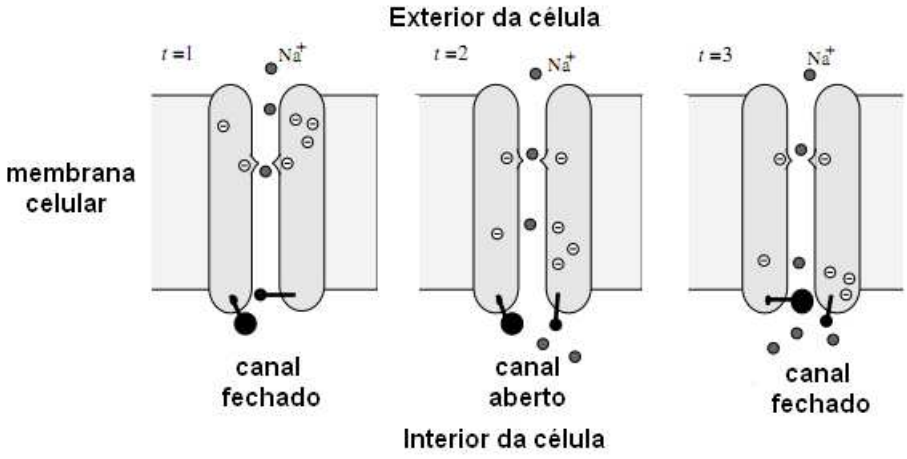


Figura 6.6: Canais de sódio e potássio

g_{Na}, o potencial volta ao valor inicial, o interior da célula torna-se negativo novamente, com sobretensão menor do que $-70mV$. Para gerar um sinal, é necessário um mecanismo controlado de despolarização e polarização.

A condutância e a resistência da membrana celular em relação a diferentes classes de íons depende da sua permeabilidade, que pode ser controlada abrindo ou fechando os canais iônicos. Existe uma classe de canais iônicos que podem ser controlados eletricamente. Estes canais reagem à despolarização da membrana celular. Quando isto acontece, ou seja, quando o potencial no interior em relação ao exterior atinge um limiar, os canais de sódio abrem-se automaticamente e os íons positivos de sódio fluem do exterior ao interior da célula, tornando o interior positivo. Isto, por sua vez, abre os canais de potássio; os íons de potássio, então, fluem para o exterior, restaurando a polarização original.

A figura 6.6 mostra um canal eletricamente controlado que deixa os íons de sódio passar. O funcionamento é baseado em uma pequena abertura no meio do canal, que, no instante $t = 1$ está negativamente carregado e fechado. Se o interior da célula torna-se positivo, cargas negativas deslocam-se no canal, aproximando-se da porta e abrindo-a por repulsão elétrica ($t = 2$). Íons de sódio, então, fluem através do canal para dentro da célula. Depois de

um curto período de tempo, a segunda porta é fechada e o canal é bloqueado ($t = 3$).

Como em qualquer sistema elétrico, os neurônios sofrem perda de carga, que deve ser continuamente balanceada. Isto é conseguido pela bomba de sódio e potássio, que transporta excesso de sódio para fora da célula e, ao mesmo tempo, potássio para o interior. A fonte de energia da bomba é o ATP. Quando o ATP é hidrolizado em difosfato de adenosina (ADP), libera-se energia que possibilita reações que, de outra forma, seriam energeticamente desfavoráveis. O ATP doa fosfato ao aspartato 369, causando mudança na conformação do enzima. A figura 6.7 mostra como isto acontece.

Sinais são produzidos e conduzidos na membrana celular. Os sinais são ondas de despolarização viajando através dos axônios. Uma tal onda de despolarização é chamada de potencial de ação. Uma ação potencial é produzida por uma polarização inicial da membrana celular. O potencial cresce de $-70mV$ até $40mV$. Depois de algum tempo, a membrana celular torna-se negativa novamente e sofre sobretensão, indo até $-80mV$. A célula recupera-se gradualmente e a membrana celular retorna ao potencial que tinha no início. O tempo de chaveamento é determinado, como em qualquer configuração capacitor/resistor, pela constante RC. Em neurônios, esta constante é de 2.4ms.

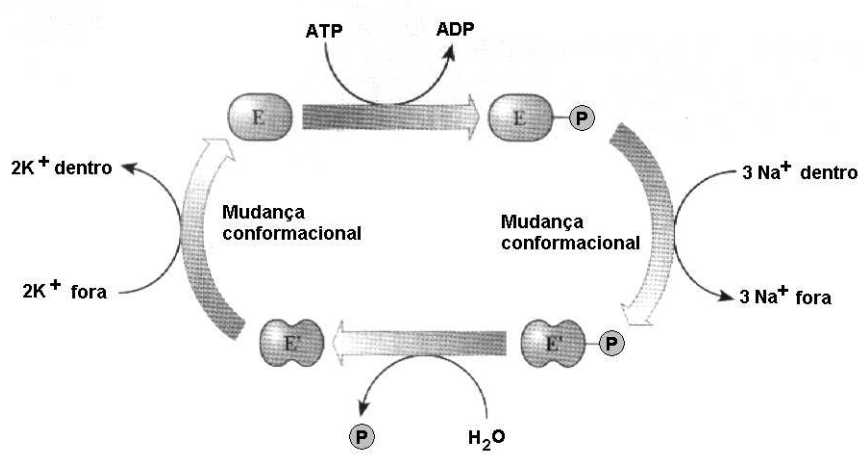


Figura 6.7: Bomba de sódio potássio

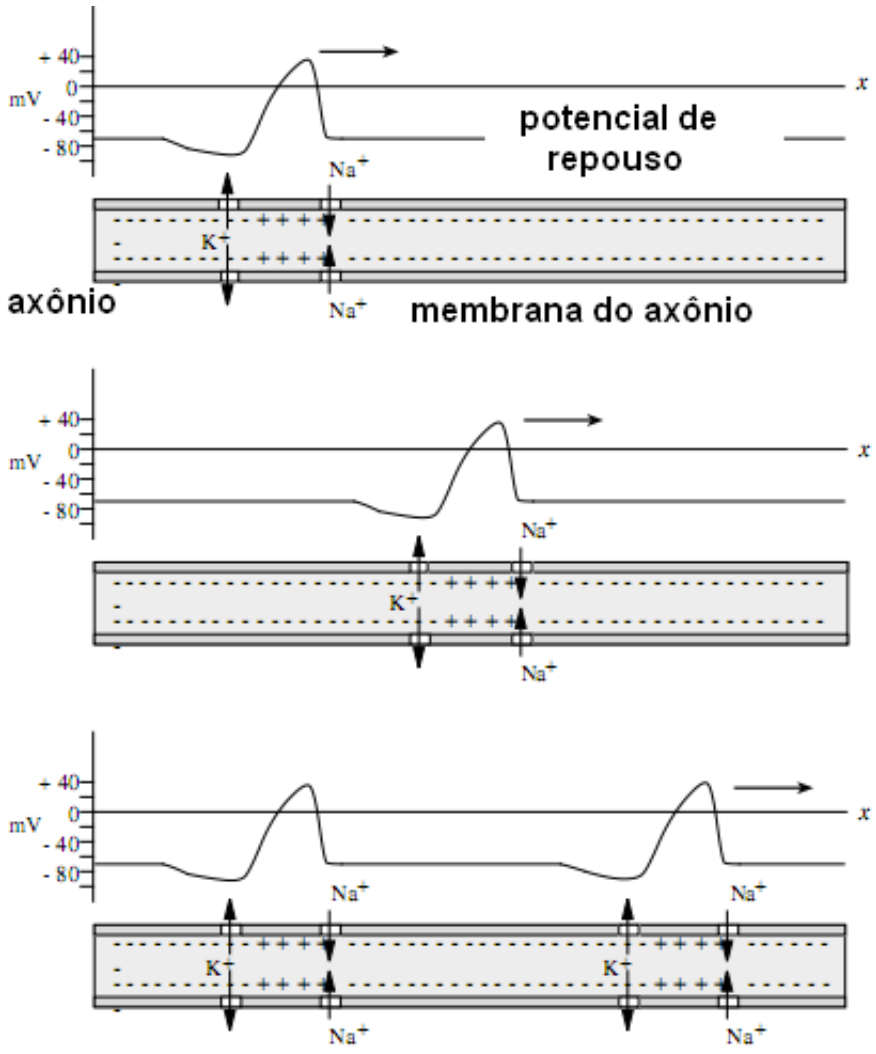


Figura 6.8: Sinal viajando pelo axônio

A figura 6.8 mostra um potencial de ação viajando através de um axônio. Uma perturbação, produzida por um sinal que chega aos dendritos, leva a uma abertura de canais de sódio em certa região da membrana. A membrana é, então, despolarizada e íons Na^+ entram na célula. Depois de um curto intervalo de tempo, um fluxo para fora de íons K^+ compensa a despolarização. Desta forma, a perturbação é transmitida através do axônio. Em todo processo, apenas energia local é consumida, ou seja, energia armazenada na própria membrana. O potencial de ação é, portanto, uma onda de per-

meabilidade ao Na^+ seguida de uma onda de permeabilidade ao K^+. Note que as partículas carregadas movem-se apenas o suficiente para perturbar o próximo canal e fazer a perturbação propagar-se.

Os neurônios transmitem informação usando potenciais de ação. O processamento desta informação envolve a combinação de processos químicos e elétricos, regulados pelas synapses. Embora existam synapses elétricas, a maioria das synapses usa sinais químicos. A figura 6.9 mostra o diagrama de uma synapse típica.

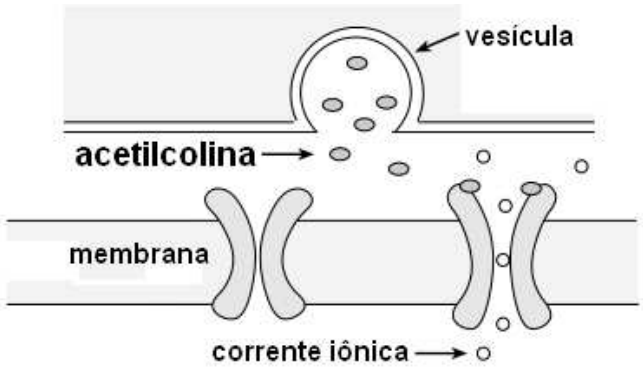


Figura 6.9: Synapse Neuronal

As vesículas contêm transmissores químicos (neurotransmissores). O pequeno espaço entre a synapse e a célula à qual está ligada chama-se hiatus. Quando um impulso elétrico chega na synapse, a vesícula funde-se à membrana. O neurotransmissor flui para o hiatus e parte dele liga-se aos canais iônicos. Se o neurotransmissor é apropriado, os canais iônicos são abertos e mais íons fluem do exterior para o interior da célula. O potencial da célula é alterado desta maneira. Se o potencial no interior da célula aumenta, isto ajuda a preparar um potencial de ação e a synapse causa uma excitação da célula. Se íons negativos são transportados para dentro da célula, a probabilidade de disparar um potencial de ação decresce, o que significa que estamos tratando de um synapse inibitória.

Synapses determinam a direção para a transmissão da informação. Sinais fluem de uma célula para outra de um modo bem definido. Isto é expresso em redes neurais artificiais organizando-se os neurônios em um grafo direcionado.

Em redes neuronais, a informação é armazenada nas synapses. Se existe outra forma de armazenamento, ela ainda é desconhecida. A eficiência da synapse em despolarizar a célula de contato pode aumentar se mais canais iônicos são abertos. Em anos recentes, receptores de NMDA estão sendo estudados porque eles exibem propriedades que ajudam a explicar como as redes neuronais aprendem (ver [Haykin]). NMDA é uma sigla para N-metil D-Aspartato. O NMDA é um aminoácido excitatório agonista do neurotransmissor, também aminoácido, glutamato. Age ativando receptores ionotrópicos conhecidos como receptores glutamatérgicos do tipo NMDA. A ativação de receptores NMDA está envolvida em mecanismos de aquisição de memórias e aprendizado.

Receptores NMDA são canais iônicos permeáveis a diferentes tipos de íons, como íons de sódio, potássio e cálcio. Estes canais são bloqueados por um íon de magnésio de tal maneira que a permeabilidade ao sódio e ao cálcio torna-se baixa. Se esta célula é levada a um certo grau de excitação, o canal iônico perde o íon de magnésio e é desbloqueado. A permeabilidade para $Ca2^{+}$ aumenta imediatamente. Através de um fluxo de íons de cálcio, uma reação em cadeia é disparada, o que provoca uma mudança durável no limiar da célula. A figura 6.10 ilustra este processo.

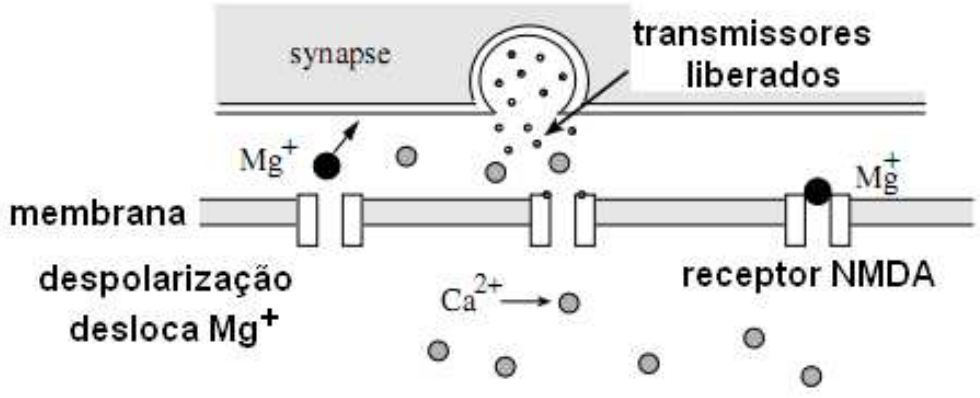
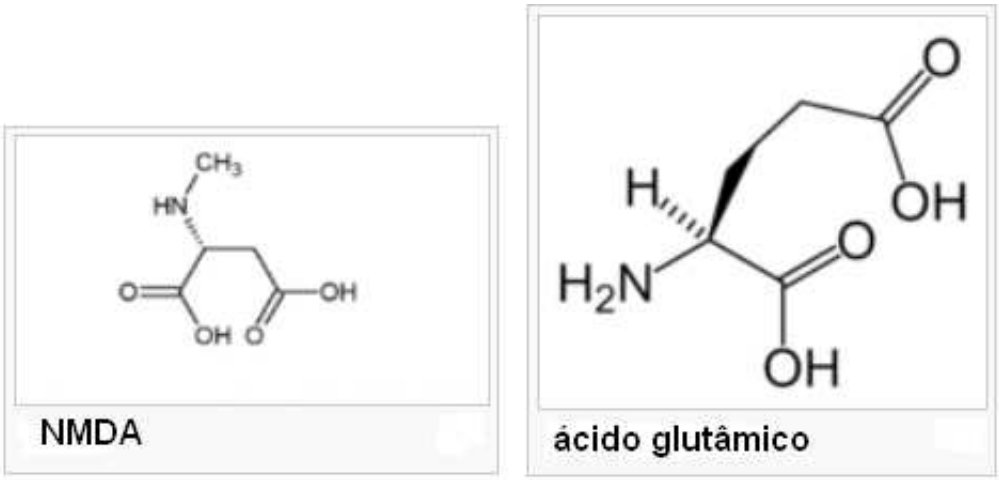


Figura 6.10: Canais iônicos

A ativação dos receptores de glutamato provoca a abertura de um canal iônico não-seletivo para os cátions, o que permite o fluxo de Na^+ e de pequenas quantidade de Ca^{2+} para dentro da célula e de K^+ para fora. Acredita-se que o fluxo de cálcio através destes receptores desempenhe um papel importante na plasticidade sináptica, um mecanismo envolvido em processos de formação de memória e aprendizagem.



O receptor NMDA possui uma série de pontos de ligação para diversas substâncias, e tem sua ação modulada por tais agente ligantes: glicina, D-serina, Magnésio, etc.

Antagonistas do receptor NMDA, tais como: Amantadina, Dextrometorfano, Dextrorfano, Etanol, Ketamina, Metadona, Óxido nitroso, PCP e Tramadol, são utilizados como anestésicos para animais e por vezes, em seres humanos, e muitas vezes são utilizados como drogas recreativas por suas propriedades alucinógenas.

Capítulo 7

Estudo de Caso — Rede neural

Este capítulo trata da obsolescência de *software* e de como evitá-la usando linguagens de domínio específico. Isto significa que abordaremos o problema apenas do lado das linguagens em que programas são escritos, deixando de lado fatores como bibliotecas e sistemas operacionais.

Considerando que linguagens de computador são sistemas formais, podendo ser consideradas ramos da matemática, era de se esperar que fossem praticamente imunes à obsolescência. Afinal de contas, não há muitas teorias matemáticas que se tornem obsoletas. Cursos de matemática, com frequência, adotam livros escritos por Gauss, Laplace, ou até mesmo por Euclides, sem que o material apresentado passe a impressão de que é ultrapassado. Linguagens de computação, contudo, tornam-se obsoletas porque sofrem restrições impostas pelo equipamento de base (*underline hardware*) e pela cultura dos programadores. Além disso, os dois fatores que afetam a obsolescência realimentam-se continuamente. Por exemplo, a tecnologia pode impôr limitações de memória, o que torna impossível a implementação de memória dinâmica. Por isso, uma linguagem de computação pode precisar de reusar o espaço alocado para variáveis. Esta é a principal razão para que atualizações destrutivas sejam tão difundidas nas linguagens de computação prevalentes. Da mesma forma, velocidade e falhas na segurança podem impedir a implementação de otimizações globais ou da recursividade de cauda. Em poucas palavras, o estado da arte em projeto de *hardware*

faz a semântica das linguagens desviar-se dos sistemas matemáticos, gerando sistemas formais que são mais fáceis de implementar e instalar. Quando avanços tecnológicos tornam desnecessárias as modificações introduzidas por limitações técnicas ou de *hardware*, as linguagens ultrapassadas são reprojetadas, tornando-se impossível compilar programas antigos nelas baseados. O grande problema, contudo, é que as linguagens reestruturadas continuam usando vários conceitos com os quais os programadores estão acostumados. Assim, a cultura ultrapassada dos profissionais torna-se um empecilho ao projeto de linguagens fortemente baseadas na matemática e na lógica.

Para fixar idéias, vamos estudar o caso particular de reconhecimento de padrões de sinais biológicos. Para isso, um sinal é representado por coeficientes que permitam, com suficiente precisão, reproduzi-lo por meio de séries ou por outro processo qualquer. Comparando os coeficientes do sinal com os coeficientes de padrões, podemos classificar os sinais. Por exemplo, admitamos que o sinal seja produzido por um eletromiógrafo (sensor que mede o potencial elétrico produzido por atividade muscular). Quando uma pessoa deseja virar-se para uma dada direção (para a esquerda, ou para a direita), a atividade elétrica dos músculos do ombro adequam-se a um certo padrão condizente com a intenção de virar-se para um lado ou para outro. Coeficientes de uma série ou de algum método de interpolação que reproduzam o sinal devem ter características que permitam descobrir a intenção do organismo que emitiu o sinal. Estas características são reconhecidas pela comparação dos coeficientes com padrões pré-estabelecidos. Assim, o sistema de reconhecimento de padrões proposto nesse trabalho possui dois componentes:

1. Obtenção de coeficientes que representem características (atributos) do sinal.

2. Comparação dos atributos do sinal com padrões, de modo a descobrir fatos sobre o organismo que emitiu os tais sinais.

Inicialmente, vamos mostrar um meio de obter os atributos do sinal, ou seja, características que permitam reproduzi-lo.

7.1 Mínimos quadrados

O algoritmo LMS (do inglês, Least-Mean-Square algorithm) é usado em filtros adaptativos para encontrar os coeficientes que permitem obter o valor esperado mínimo do quadrado do sinal de erro, definido como a diferença entre a amostra do sinal e o resultado produzido pela saída do filtro. Os parâmetros do filtro que minimizam o erro determinam as características do sinal. Vamos chamar esses parâmetros de coeficientes do filtro.

Vamos admitir que os coeficientes sejam armazenados em um vetor `a`, e que as amostras do sinal sejam guardadas em um vetor `y`. Nesse caso, o sinal em um dado instante pode ser estimado como:

```
(define (yHat a y n m)
   (- (sum-ec (:range i \min(m, n+1); )
           \a[i]*y[n-i];   )
   )
)
```

A função `sum-ec` é uma extensão sintática, denominada compreensão, e descrita por [Egner]. A definição acima realiza a seguinte soma:

$$- \sum_{0}^{min(m,n+1)} a[i] \times y[n-i]$$

Como podemos ver, a sintaxe da compreensão de Egner é um espelho perfeito da expressão matemática.

O algoritmo LMS atualiza iterativamente os parâmetros do filtro. Isto pode ser realizado com o seguinte programa:

```
(define (newA a y yh mu n)
   (vector-ec (:vector wi (index i) a)
       \ if (i>n) wi;
         else wi-2.0*mu*y[n-i]*(y[n]-yh);  ))
```

A extensão sintática `vector-ec` é parte da biblioteca de compreensão de Egner, exatamente como `sum-ec`. Cada elemento do novo vetor filtro é dado

por $w_i - 2.0 \times \mu \times y[n-i] \times (y[n] - yh)$. Novamente, o algoritmo Scheme espelha a formulação matemática bastante bem. Isso acontece porque Egner criou uma extensão sintática apropriada para esse tipo de problema. Essa extensão sintática é distribuída com as fontes do programa. No futuro, é possível recompilar o programa, mesmo que a biblioteca de compreensão de Egner não esteja disponível na distribuição da Scheme que se pretenda utilizar. A propósito, tanto quanto sabemos, a biblioteca de compreensão de Egner não faz parte de nenhum compilador Scheme. Se o leitor quiser testar os programas do presente documento, deve fazer o *download* da página de Egner ou reescrevê-la seguindo instruções dadas em [Egner]. Ambas as soluções são fáceis de implementar.

O leitor deve estar curioso em saber se podemos escrever bibliotecas de compreensão em linguagens que não pertençam à família da Lisp. De fato, compreensão é uma construção sintática muito popular, e está presente na Clean, na Haskel e em outras linguagens. O ponto importante é que ela pode ser facilmente adicionada à Scheme. Mesmo que a sintaxe de Egner torne-se obsoleta, podemos continuar a utilizá-la para compilar programas antigos, já que é fácil estender Scheme com recursos de compreensão. No caso de outras linguagens, é muito difícil implementar uma extensão como a biblioteca de compreensão. Por isso, o programador precisa esperar por decisões dos implementadores.

Acreditamos que os exemplos anteriores sejam suficientes para ilustrar a principal alegação deste capítulo, ou seja, que podemos vencer a obsolescência pela utilização de um sistema formal com sintaxe minimalista. Por completitude, porém, vamos apresentar o resto do algoritmo LMS.

```
(define (lms y w mu)
   (let loop ( (n 0) (a w) (yhs '()) )
       (if (= n (vector-length y))
          (list a (reverse yhs))
          (let ((yh (yHat a y n (vector-length a))))
             (loop (+ n 1) (newA a y yh mu n)
                           (cons yh yhs)))  )))
```

O algoritmo acima é suficientemente pequeno para ser compreendido por uma

pessoa com algum conhecimento de Scheme. Um bom texto para aprender Scheme é [Dybvig]. Agora, vamos escrever um programa simples para calcular a média de uma lista de números.

```
(define (avg s)
   (/ (sum-ec (:list x s) x) (length s)))
```

O próximo passo é calcular o desvio padrão de uma lista.

```
(define (dev s)
   (let ((m (avg s)))
      (avg (list-ec (:list x s) (* (- x m) (- x m)))
)  )   )
```

O erro entre um sinal e uma lista de estimativas é dado pela seguinte definição:

```
(define (err y yhs)
  (dev (list-ec (:parallel
                   (:vector yi y)
                   (:list yhi yhs))
          \yi - yhi;)
  )
)
```

Com esse método de estimação de erro, os parâmetros usados no algoritmo de reconhecimento de padrões são calculados pelo seguinte programa:

```
\obj loopit(int j, obj y, obj a, obj mu) {
     ax = lms(y,a,mu);
     if (err(y,cadr(ax)) < 0.01) ax;
     else loopit(j+1, y, car(ax), mu); };
```

Na definição dada acima, utilizamos uma sintaxe similar à encontrada em linguagens tradicionais. A sintaxe acima é uma extensão que vem na caixa da distribuição da linguagem Gambit Scheme. É muito fácil adicionar notação infixa às implementações Scheme que não a fornecem.

Nós resolvemos o problema de calcular os parâmetros necessários para o reconhecimento de padrão de sinais biológicos com um filtro de mínimos quadrados. Naturalmente, poderíamos utilizar qualquer outro método de estimativa de sinal: transformada de Fourier/Ferraz-Mello, transformada de Foster, ondeletas (francês, ondelettes), filtro de Kalman, polinômios de Bernstein ou redes neurais. Aqui cumpre dizer que o método das redes neurais, estudado no capítulo anterior, pode ser utilizado para aproximar um sinal.

Algoritmos de LMS são utilizados em filtros adaptativos para encontrar os parâmetros relacionados a um sinal de erro que produz uma média dos quadrados mínima. Trata-se de um método de gradiente descendente estocástico, isto é, o filtro é adaptado de acordo com o erro corrente.

LMS é um método muito antigo, tendo sido usado pela primeira vez em 1960. Um método novo é a rede neural, que estudamos no capítulo anterior. Neste capítulo, vamos apresentar, em Gambit Scheme, o mesmo algoritmo que publicamos em Common Lisp no capítulo anterior.

```
(include "ec.scm")

#| Given a vector to store the weights, and a
   list ws of indexes, newn builds a neuron. E.g.
   (let [ (v '#(0 0 0 0)) ]
      (newn v '(0 1 2)))
   produces a two input neuron that uses v[0],
   v[1] and v[2] to store its weights. |#

\float sig(float x) {1.0/(1.0+exp(-x));}

(define (newn v ws)
   (lambda(xs)
     (sig (sum-ec (:parallel
                       (:list i ws)
                       (:list x (cons 1.0 xs)))
                \v[i]*x; ))  ))
```

```
;; Given a vector vt, (prt vt) creates
;; a neuron network that can learn to act
;; like a logical port.

(define in-1 car)
(define in-2 cadr)

(define (gate vt)
 (let [ (n1 (newn vt '(4 5 6)) )
        (ns (newn vt '(0 1 2 3)))]
     (lambda (i)
       (if (null? i) vt
          (ns (list (in-1 i)
                    (n1 (list (in-1 i) (in-2 i)))
                    (in-2 i)  )))   )))

;; Here is how to create a xor neural network:

(define xor (gate (vector -4 -7 14 -7 -3 8 8)))


(define dx 0.01)
(define lc 0.5)

(define *nuweights*  (make-vector 90)  )
(define *examples* #f)

(define (assertWgt vt I R)
   (vector-set! vt I R)   R)

(define (egratia eg)
   (vector-ref *examples*
     (min eg (- (vector-length *examples*) 1)) ))

(define (setWeights vt Qs)
  (do-ec (:range i (vector-length vt))
      (vector-set! vt i
         (vector-ref Qs i)) ) )
```

```
(define (errSum prt Exs)
  (sum-ec (:list e Exs)
     (:let eg (egratia e))
     (:let vc (prt (cdr eg)  ))
     (:let v (car eg) )
       \(vc-v)*(vc-v); ) )

(define (updateWeights prt vt err0  ns Exs)
  [do-ec (:range i (+ ns 1))
      (:let v (vector-ref vt i))
      (:let v1 (assertWgt vt i  (+ v dx)))
      (:let nerr (errSum prt Exs))
      (:let NV  \v+lc*(err0-nerr)/dx; )
      (begin (assertWgt vt i v)
            (vector-set! *nuweights* i nv)   ) ]
  (setWeights vt *nuweights*) )

(define (train p exs)
   (set! *examples* exs )
   (set! *nuweights* (make-vector 90))
   (setWeights  (p '()) '#(0 1 0 0 2 0 0))
   (do ( (vt (p '()))
         (exs '(0 1 2 3 3 2 1 0)) )
      ( (< (errSum p exs) 0.001) )
      (updateWeights p vt (errSum p exs)
                        (- (vector-length vt) 1) exs)  ) )

(define *exs*
   '#( (0 1 1) (1 0 1) (1 1 0) (0 0 0)) )
```

Capítulo 8

Conclusões e Trabalhos Futuros

8.1 Sensores adaptáveis

Neste trabalho de tese, foi mostrado que sensores adaptáveis a diferentes tipos de sinais oferecem um arcabouço seguro para sistemas de aquisição e processamento da dados. A segurança vem do fato de que não precisamos de testar sensores específicos. Além disso, a instalação de um novo sistema de aquisição de dados torna-se muito mais simples e rápida, uma vez que se pode aproveitar os sensores e o *hardware* que os acompanha. O *software*, graças aos mecanismos matemáticos de proteção contra a obsolescência, tem uma meia vida muito mais longa e, tal como o *hardware*, pode ser rapidamente adaptado a novas circunstâncias.

8.2 Sistemas evolutivos

Sistemas evolutivos permitem determinar parâmetros e funções de correção que minimizam o erro dos sensores. Isso facilita ainda mais a adaptação do *software* a novos contextos e aplicações. Funções de calibração e controle de erros podem ser geradas automaticamente, diminuindo o tempo entre a especificação do sistema e seu emprego em medições.

8.3 Modelos biológicos de computação

Modelos biológicos de computação (redes neurais) garantem a plausibilidade de medições de sinais fisiológicos (eletromiografia, capnografia, etc.) A biologia utiliza-se, no processamento de sinais, de redes neurais, de sistemas hormonais e de evolução por seleção natural. Desses três sistemas de controle, o autor desta tese usou dois, a saber, evolução e redes neurais. Por trabalhar dentro dos mesmos princípios que os sistemas biológicos, os sensores descritos nesse trabalho têm o potencial de integrar-se melhor nos sistemas biológicos que se queira medir. Essa melhor integração é fundamentada nos seguintes ítens:

Relação entre escalas temporais. Sistemas possuem diferentes escalas de tempo e espaço. Por exemplo, um sistema geológico tem escala de milhões de anos. Um sistema estelar tem escala de bilhões de anos. Não é possível acompanhar esses sistemas por medição direta; por essa razão, eles são estudados por meio de fósseis, como restos petrificados de dinossauros ou a radiação de fundo do *Big Bang*. Sistemas históricos e arqueológicos ocorrem em escalas de dezenas a centenas de anos e são estudados por meio de documentos e vestígios (cerâmica, monumentos, templos e obras de construção civil). A dinâmica de certas reações químicas ocorrem em femtosegundos (10^{-15} segundos). Para estudá-las, Ahmed Hassan Zewail projetou pulsos de laser de curta duração sobre partículas que participam da reação. Como átomos e moléculas, segundo os princípios da mecânica quântica (espectro de rotação, vibração, vibração nuclear, mudança de nível eletrônico, etc.), absorvem e irradiam radiação eletromagnética distintamente, modificando o espectro de forma característica, [Zewail] conseguiu estudar o comportamento das partículas em tempo real.

Os sitemas biológicos ocorrem em três escalas de tempo. A evolução por seleção natural, que constrói os organismos vivos, com sistemas cibernéticos de controle e comunicação *hardwired* (estrutura das redes neurais e circuitos hormonais), tem escala de milhões de anos. As redes neurais têm escalas que vão de milisegundos a dias. Os circuitos e ciclos hormonais têm escalas

que vão de segundos a meses. Ao preparar sensores que imitam os processos biológicos, mantemos a relação entre escalas. Por exemplo, a evolução de uma rede neural leva horas; obviamente, seria inviável se levasse milhões de anos; claro que uma rede cuja evolução durou horas não é tão sofisticada quanto um *Tyranosaurus Rex*, cuja evolução custou milhões de anos a Gaia. A resposta de uma rede neural artificial leva de 10^{-6} s a 10^{-3} s, dependendo do computador (velocidade varia de alguns megaflops a vários teraflops) e da complexidade da rede. Admitamos que a evolução da rede leve algumas horas. Então, a relação entre as escalas vários milhões e alguns bilhões. Essa relação é, exatamente, a mesma que nos sistemas biológicos. Essa plausibilidade biológica coloca restrições úteis na metodologia de emprego dos sensores. Assim, o sensor só poderá ser utilizado depois de ter sua eficiência e correção otimizadas pela programação genética.

Comportamento da rede neural. O comportamento de uma rede neural é, aproximadamente, o mesmo que de uma estrutura de neurônios. Assim, os resultados das redes neurais são aceitos com menos resistência e mais segurança pelos usuários e podem ser melhor interpretados. A dificuldade que seres humanos têm de interpretar sinais condicionados por sistemas que funcionam por princípios muito diferentes do encontrado no cérebro é notória. Em um livro sobre o assunto, [Mlodinow] fornece um exemplo esclarecedor:

> *My most memorable encounter with the Reverend Bayes came one Friday afternoon in 1989, when my doctor told me by telephone that the chances were 999 out of 1000 that I'd be dead within a decade...*
>
> *The adventure started when my wife and I applied for life insurance. The application procedure involved a blood test. A week or two later we were turned down. The ever economical insurance company sent the news in two brief letters that were identical, except for a single additional word in the letter to my wife. My letter stated that the company was denying me insurance because of the "results of your blood test." My wife's letter stated that the company was turning her down because of the "result of your*

> *husband's blood test." When the added word* husband's *proved to be the extent of the clues the kindhearted insurance company was willing to provide about our uninsurability, I went to my doctor on a hunch and took an HIV test. It came back positive. Though I was too shocked initially to quiz him about the odds he quoted, I later learned that he had derived my 1-in-1,000 chance of being healthy from the following statistic: the HIV test produced a positive result when the blood was not infected with the AIDS virus in only 1 to 1,000 blood samples. That might sound like the same message he passed on, but it wasn't. My doctor had confused the chances that I would test positive if I was not HIV-positive with the chances that would not be HIV-positive if I tested positive.*
>
> *. . . let us prune the sample space to include only those who tested positive. We end up with 10 people who are false positives and 1 true positive. In other words, only 1 in 11 people who test positive are really infected with HIV.*

[Mlodinow] e outros autores fornecem dezenas de outros exemplos da dificuldade que temos de interpretar redes bayesianas e outras formas de fazer inferência sobre dados. Por oferecem métodos de inferência mais intuitivos e naturais, tornou-se comum a utilização de redes neurais em câmaras digitais, controle de temperatura de chuveiro, etc. A combinação das redes neurais com programação genética, por outro lado, impede que elas seja usadas em situações em que não são adequadas; afinal, redes inadequadas serão recusadas pela função de ajustamento (*fitness function*).

8.4 Linguagens de domínio específico

Linguagens de domínio específico baseadas em sistemas formais matemáticos (Cálculo Lambda) herdam as características desejáveis da matemática: Sistemas expressivos, corretos e que não se tornam obsoletos.

Em geral, redes neurais são implementadas em linguagens de baixo nível. Mostramos que, utilizando linguagens de domínio específico, podemos imple-

mentar redes neurais encapsuladas, prontas para paralelismo (os neurônios são independentes), com neurônios que se comunicam por meio de uma área de memória comum. Além disso, a linguagem de domínio específico permite mudar facilmente a estrutura da rede, sem modificar o programa principal.

8.5 Artigos Submetidos

O autor submeteu quatro artigos, onde discute estas conclusões em detalhes:

1. Pattern Recognition of Biological Signals - Ênfase em sistemas resistentes à obsolescência e cálculo lambda na programação (artigo publicado).

2. Relationship Between Completeness and Iterated Soundness - Aspectos formais de redes neurais e sensores adaptativos.

3. Remote Laboratory and Distance Education - Utilização de sensores adaptativos para projeto de aulas experimentais em educação à distância.

4. Evolutionary Computation of Electric Circuit Parameters - Sistemas evolutivos na determinação da função de circuitos elétricos.

8.6 Trabalhos Futuros

Circuitos eletrônicos (*hardware*) de aquisição e armazenamento de dados foram discutidos neste documento e implementados pelo autor. Esse sistema de *hardware* e *software* é utilizado para alimentar redes neurais, criando um sistema capaz de fazer aquisição de sinais e processá-los.

Num futuro próximo, o autor desse documento pretende utilizar tais equipamentos em quatro projetos.

Aquecimento global. O aquecimento global é um dos principais problemas que a humanidade enfrenta nesse século XXI. Se não resolvido, o aquecimento global pode ter consequências catastróficas, principalmente em países

como o Brasil, onde se espera desertificação e dramáticas quedas na produção agrícola.

Uma das consequências do aquecimento global é a intensificação de um fenômeno chamado ilhas de calor. Tecnicamente, ilha de calor urbana (ICU) é a distribuição espacial e temporal do campo de temperatura sobre a cidade que apresenta um máximo, como se fosse uma ilha quente localizada. O autor construiu sensores capazes de registrar temperatura e pressão parcial de CO_2, assim como o *software* necessário para ler, registrar e processar os sinais gerados pelos sensores. Com isso, pretende-se estudar a influência da concentração de CO_2 na intensidade da ilha de calor. Uma rede de sensores será conectada à internete para realizar os estudos pretendidos.

De acordo com o [Wittgenstein] do Tratactus, qualquer sistema pode ser descrito em termos de mudança de estado. Diz o filósofo:

> Die Welt is alles, was der Fall ist.
> ...
> Was der Fall ist, ist das Bestehen von Sachverhalten.

Para nossos propósitos, podemos considerar estado como um conjunto de atributos com valores. Estamos interessados em atributos cujos valores são quantidades físicas, como temperatura, potencial elétrico ou pressão parcial de um gás.

Mudanças climáticas e saúde pública. Uma das consequências da mudança climática causada pelo aquecimento global refere-se à saúde pública. Alguns médicos brasileiros estão estudando a correlação entre mudanças climáticas e aquecimento global. Os sensores construídos pelo autor poderão ser utilizados nesses estudos, tanto para colher dados meteorológicos de temperatura e pressão parcial de CO_2 em tempo real, quanto na aquisição de sinais fisiológicos de pacientes. Por exemplo, um dos sensores pode medir o aproveitamento de oxigênio em atmosfera mais quente e com maior pressão parcial de CO_2.

Coleta de dados fisiológicos. Inicialmente, esse trabalho tinha por finalidade processar sinais fisiológicos para sistemas de saúde pública e colocá-los em rede. Pensava-se especificamente em eletrocardiografia (ECG), eletroencefalografia (EEG) e eletromiografia (EMG). Evidentemente, o sistema presta-se a essa aplicação para o qual foi projetado desde o início. Entretanto, criada a oportunidade de trabalhar com pesquisadores da UNESP, USP e Ryerson em aquecimento global, decidiu-se ampliar as aplicações do sistema.

Utilização de sinais fisiológicos em sistemas de controle. Em uma aplicação futura, pretende-se utilizar os sensores para captar sinais eletromiográficos, que serão utilizados no controle dos movimentos de uma cadeira de rodas motorizada, já desenvolvida pelo autor.

Figura 8.1: Cadeira de rodas que será controlada através de sinais eletromiográficos e Visor do sistema de controle.

Nessa cadeira, pretende-se controlar velocidade, direção (esquerda ou direita), movimento para a frente ou para trás e parada.

Bibliografia

[Aires] AIRES, M. M. Fisiologia Básica. Rio de Janeiro: Editora Guanabara Koogan S.A., 1985

[Alain Colmerauer] Alain Colmerauer e Philippe Roussel. La naissance de Prolog. In History of Programming Languages, edited by Thomas J. Bergin and Richard G. Gibson, ACM Press/Addison-Wesley, 1996.

[Alan Hodgkin] Hodgkin, A.L. and Huxley, A.F. (1952) Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo. Journal of Physiology, 116: 449-472.

[Alonzo Church] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, Series 2, 33:346-366, 1932.

[Anirban Dash] Anirban Dash. PANINI'S GRAMMAR - A FEW CHARACTERISTICS. LANGUAGE IN INDIA. Volume 5 : 2 February 2005

[Barski] Conrad Barski. Land of LISP (Paperback). NO STARCH PRESS; 1 edition (30 Mar 2010). ISBN-10: 1593272006 ISBN-13: 978-1593272005

[Basha] Basha, T.; Scott, R.N.; Parker, P.A.; Hudgins, B.S. Deterministic Components in the Myoeletric signal. `Med. & Biol. Eng. & Comput.`, 1994.

[Berbari] Berbari, E. J. Principles of Electrocardiography. in: Bronzino, J. D. (Ed.) The Biomedical Engineering Hand-Book. Boca Raton - Florida : CRC Press, 1995.

[Bernstein] S. N. Bernstein. Démonstration du théorème de Weierstrass fondée sur le calcul des probabilités. *Comm. Soc. Math. Kharkov*, 13 (1912) 1-2.

[Bob Glickstein] Bob Gluckstein, Writing GNU Emacs Extensions. O'Reilly & Associates, Inc. ISBN 1-56592-261-1. 1997.

[Burr-Brown] BURR-BROWN, Low Power, 16 Bit, Sampling CMOS Analog-to-Digital Converter, disponível em http://focus.ti.com/lit/ds/symlink/ads7807.pdf, acesso em 20/01/2010.

[Caparelli] Paulo S. Caparelli, Eduardo Costa, Alexsandro S. Soares, Hipólito Barbosa. Pattern Recognition of Biological Signals. International Journal of Computational Intelligence. ISSN: 13044508, EISSN: 13042386. Volume 5, Number 4, Autumn 2009.

[Church-1] A. Church. A note on the Entscheidungsproblem. *Journal of Symbolic Logic*, 1:40-41, 1936.

[Church-2] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58:345-363, 1936.

[Cleanest] G. Foster. The CLEANEST Fourier spectrum. Astronom. J., 109(4):1889-1902, Apr 1995.

[Cooley and Tukey] Cooley, James W., and John W. Tukey, 1965, An algorithm for the machine calculation of complex Fourier series, Math. Comput. 19: 297-301.

[David Henderson] David Henderson. Differential geometry and Non-Euclidean geometry (with Daina Taimina), signed articles in Encyclopedia Britannica, 2005.

[Douglas Hofstadter] Douglas Hofstadter. Godel, Escher, Bach: An Eternal Golden Braid. Vintage. 1989.

[Drew McDermott] Drew McDermott. A critique of pure reason. Computational Intelligence 3(33),pp. 151-160. (1987).

[Dybvig] Kent Dybvig. The Scheme Programming Language. The MIT Press.ISBN-10:0-262-51298-X. ISBN-13:978-0-262-51298-5. September 2009.

[Eduardo Costa] Eduardo Costa. Visual Prolog for Tyros. Pdc. 2008.

[Elena Efimova] . Eduardo Costa e Elena Efimova (tradutora). Visual Prolog Dlya Nachinayuschikh. Pdc. 2008.

[Egner] Sebastian Egner. Eager Comprehensions in Scheme. Sixth Workshop on Scheme and Functional Programming. Tallinn, Estonia. 2005.

[Euclid] Euclid and T.L. Heath (translator). Euclid's Elements. Green Lion Press. (August 20, 2002). ISBN-10: 1888009195. ISBN-13: 978-1888009194.

[Fejer] B. G. Fejer, J. R. Souza, A. S. Santos, and A. E. Costa Pereira. Climatology of F region zonal plasma drifts over Jicamarca. *Journal of Geophysical Research*, Vol. 110, A12310, doi: 10.1029/2005JA011324, 2005.

[Ferraz-Mello] S. Ferraz-Mello. Estimation of periods from unequally spaced observations. Astronom. J., 86:619-624, 1981.

[Gauss] Carolus Fredericus Gaussius. Disquisitiones generales circa superficies curvas. Commentationes Societatis Re-

giae Scientiarum Gottingesis Recentiores. Volume VI, pp. 99-146. 1827.

[Haykin] Simon S. Haykin. Redes Neurais - Principios e Pratica. Editora Bookman, Edição 2, 2001. ISBN 8573077182, 9788573077186.

[Heideman et al.] Michael T Heideman, Don H Johnson, Sidney C Burrus. Gauss and the history of the fast Fourier transform. Archive for History of Exact Sciences, Vol. 34, No. 3. (1985), pp. 265-277.

[Henk Barendregt] Henk Barendregt, The Impact of the Lambda Calculus in Logic and Computer Science. The Bulletin of Symbolic Logic, Volume 3, Number 2, June 1997.

[Ivan Bratko] Prolog Programming for Artificial Intelligence. Terceira Edição. Addison Wesley. 2000.

[John Allen Paulos] John Allen Paulos. Innumeracy, Mathematical Illiteracy and its consequences. Hill and Wang, a division of Farrar, Straus and Giroux.2001.

[John Backus] John Backus. Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. Communications of the ACM. August 1978. Volume 2 i. Number 8.

[Joshua Lederberg] Joshua Lederberg. How Dendral was conceived and born. ACM Symposium of Medical Informatics. National Library of Medicine. November 5, 1987.

[Lacerny] William D Clinger e Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In the Proceedings of the 1994 ACM conference on LISP and Functional Programming, SIGPLAN Lisp Pointers 7(3) (July-September 1994), pages 128-139.

[Lathi] Lathi, B. P. Modern Digital and Analog Communication Systems, Oxford University Press, 3rd ed, 1998.

[Leo Brodie] Leo Brodie. Starting Forth. Prentice Hall. 1981.

[Lienhard] J. H. Lienhard. Interchangeable Parts. April 18, 2009. `http://www.uh.edu/engines/epi1252.htm`.

[Lucília Figueiredo] Elton M. Cardoso e Lucília Figueiredo. Entrada e Saída em Haskell. Departamento de Computação, Universidade Federal de Ouro Preto. 2005.

[Manfredo do Carmo] Manfredo do Carmo. Differential Geometry of Curves and Surfaces. Prentice Hall; First edition (February 11, 1976). ISBN-10: 0132125897. ISBN-13: 978-0132125895.

[Michael Gordon] Michael J. C. Gordon. Programming Language Theory and its Implementation. Prentice Hall. (May 1988). ISBN 0-13-730417-X. ISBN-13: 978-0137304172.

[Mlodinow] Leonard Mlodinow. The Drunkard's Walk. Pantheon. (May 2008). ISBN-10: 0375424040. ISBN-13: 978-0375424045.

[Moses Schonfinkel] Moses Schonfinkel, Uber die Bausteine der mathematischen Logik, Mathematische. Annalen, vol. 92. (1924), pp. 305-316.

[Niedermeyer] Niedermeyer, E. and Lopes da Silva, F. Electroencephalography: Basic Principles, Clinical Applications, and Related Fields. Willians & Wilkins. 3a. Ed. 1993.

[Nyquist] H. Nyquist, Certain topics in telegraph transmission theory, Trans. AIEE, vol. 47, pp. 617-644, Apr. 1928. Reprint as classic paper in: Proc. IEEE, Vol. 90, No. 2, Feb 2002.

[Paul Graham] Paul Graham. On Lisp. Prentice Hall, 1993, 432 pages, paperback. ISBN 0130305529.

[Perkis] T. Perkis, Stack-based genetic programming, in Proc. 1994 IEEE World Congress on Comput. Intell., IEEE Press: New York, 1994, pp. 148-153.

[Perry] T. S. Perry. In search of the future of air traffic control. Spectrum, IEEE Publication Date: Aug 1997. Volume: 34, Issue: 8. On page(s): 18-35. ISSN: 0018-9235.

[Peter Seibel] Peter Seibel. Practical Common Lisp. Apress; 1 edition (April 11, 2005). ISBN-10: 1590592395. ISBN-13: 978-1590592397.

[Peyton Jones] P. W. Trinder, K. Hammond, H. W. Loidl and S. Peyton Jones. Algorithm+Strategy= Parallelism. *Journal of Functional Programming* 1 (1), January, 1993. Cambridge University Press.

[Philippos] Philippos Apolinário Costa. Exercícios de Óptica. Comunicação Privada.

[Philippos-2] Philippos A. Costa. Diagramando a química do vestibular; e-book distribuído em www.discenda.org/organica.pdf. Acessado em 8 de janeiro de 2010.

[Proclus] Proclus and Glenn R. Morrow (translator). A Commentary on the First Book of Euclid's Elements. Princeton University Press. (October 19, 1992). ISBN-10: 0691020906. ISBN-13: 978-0691020907.

[PLT] How to Design Programs, Matthias Felleisen, Robert Bruce Findler, Matthew Flatt and Shriram Krishnamurthi, MIT Press, 2001.

[Quine] Williard van Orman Quine. O Sentido da Nova Lógica. Editora: UFPR. 1996.

[Richard Gabriel] Gabriel, Richard P. (May 1985) (PDF). Performance and evaluation of Lisp systems. MIT Press; Computer Systems Series. ISBN 0-262-07093-6; LCCN: 85-15161.

[Richard Dawkins] Dawkins, Richard. The Selfish Gene. New York, New York: Oxford University Press. p. 15. ISBN 0192860925, 1976.

[Sandborn] Peter Sandborn. The Other Half of the DMSMS Problem — Software Obsolescence. DMSMS Knowledge Sharing Newsletter, Vol. 4, Issue 4, pp. 3 and 11, June 2006.

[Shiro Kawai] Shiro Kawai. Tracking Assets in the Production of Final Fantasy : The Spirits Within. Game Developers Conference 2002, San Jose, March 19-23, 2002.

[Sterling e Shapiro] The Art of Prolog. Leon Sterling e Ehud Shapiro. Segunda Edição. The MIT Press. 1994.

[Texas] TEXAS, Getting the most out of your instrumentation amplifier design, ANALOG Applications Journal, 2005.

[Time-series] G. Foster. Time series analysis by projection. i. statistical properties of Fourier analysis. Astronom. J., 111(1):541-554, Jan 1996.

[Time-series-ii] G. Foster. Time series analysis by projection. ii. tensor methods for time series analysis. Astronom. J., 111(1):555-566, Jan 1996.

[Warren McCulloch] Warren Sturgis McCulloch 1943. A Logical Calculus of the Ideas Immanent in Nervous Activity. With: Walter Pitts. In: Bulletin of Mathematical Biophysics Vol 5, pp. 115-133.

[Wavelets] G. Foster. Wavelets for period analysis of unevenly sampled time series. Astronom. J., 112(4):1709-1729, Oct 1996.

[Wittgenstein] Ludwig Wittgenstein. Tractatus Logico-Philosophicus, (1921). WITTGENSTEIN, Ludwig. Tractatus Logico-Philosophicus. São Paulo: Edusp, 1994.

[Zewail] A. H. Zewail, Advances in Lasers Spectroscopy, SPIE, ISBN-10: 0892521406; ISBN-13: 978-0892521401. 1977.

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo