Resolução dos Exercícios do Capítulo I

1. Identifique problemas de legibilidade e redigibilidade nas LPs que conhece. Verifique se existem casos nos quais essas propriedades são conflitantes.

Problemas de Legibilidade:

O comando *goto* de C permite a criação de programas de difícil entendimento. Em C, o símbolo ´*´ possui mais de um significado, o que pode ocasionar dificuldade de entendimento em expressões. LISP requer o uso extensivo de parênteses como marcadores de inicio e fim em programas. Em módulos com muito aninhamento, o programador pode se perder no emaranhado de parênteses. Uma mesma variável em BASIC pode ter diferentes tipos ao longo de um programa. Isso dificulta a legibilidade porque pode ser necessário acompanhar o código para identificar o tipo atual de uma variável.

Problemas de Redigibilidade:

FORTRAN não possui registros para armazenar tipos diferentes de dados. Isso faz com que se tenha a necessidade de criar vários vetores, cada um para armazenar elementos de um determinado tipo. A manipulação desse conjunto de vetores torna-se uma tarefa complexa, diminuindo a redigibilidade do programa.

COBOL é uma linguagem com o propósito de fazer os programas se parecerem com relatórios escritos em inglês. A necessidade do programa ser parecido com um texto com palavras em linguagem natural acaba por aumentar a necessidade de escrita e reduzir a redigibilidade.

Características Conflitantes:

O seguinte comando em C prioriza redigibilidade, mas reduz a legibilidade.

```
c = (a < b) ? 3 : 5;
```

Isso pode ser contrastado com o trecho equivalente abaixo, que prioriza a legibilidade em detrimento da redigibilidade.

```
if (a < b)
   c = 3;
else
   c = 5;
```

O comando for abaixo também é um exemplo no qual a boa redigibilidade conflita com a redigibilidade. Com uma só linha é possível copiar um vetor de elementos sobre outro. Por outro lado, o significado dessa linha não fica imediatamente explícito no código.

```
for ( ; *q = *p; q++, p++);
```

2. Identifique problemas de confiabilidade e eficiência nas LPs que conhece. Verifique se existem casos nos quais essas propriedades são conflitantes.

Problemas de Confiabilidade:

Uso de ponteiros em C permitem acessar áreas de memória não alocadas.

Ausência de tipo fixo das variáveis de BASIC só permite a identificação de erros de tipo em tempo de execução.

Ausência de tipo booleano em C pode provocar erros como no exemplo abaixo.

```
int i=1;
while( i = 10 ){ // o programador pulou um = do==, gerando loop infinito
        i++;
}
```

Problemas de Eficiência:

JAVA e PASCAL fazem verificações de acesso a posições válidas em vetores em tempo de execução. Isso reduz a eficiência.

Verificação se os argumentos de uma operação são de tipo válido ocorre em tempo de execução em BASIC. Isso reduz a eficiência.

Características Conflitantes:

O problema da verificação de acesso em JAVA a posições de vetores é um exemplo de sacrifício da eficiência para obter maior confiabilidade. Por outro lado, a postura de C de não fazer esse tipo de verificação prioriza eficiência em detrimento da confiabilidade.

3. Identifique problemas de falta de ortogonalidade nas LPs que conhece. Esses problemas comprometem a facilidade de aprendizado da LP?

   PASCAL não permite que registros e vetores sejam retornados como resultado de funções. A postura ortogonal seria permitir que esses tipos fossem retornados tal como todos os outros tipos da linguagem. Esse problema acaba gerando dificuldades de aprendizado, uma vez que o programador deve ter de memorizar essas restrições. Muitas vezes, ele só aprenderá isso no momento que tentar criar uma função retornando um valor desses tipos.

4. Reusabilidade e modificabilidade muitas vezes contribuem para a melhoria uma da outra. Dê exemplos de situações nas quais isso ocorre.

   Ao se criar uma constante para designar uma aproximação do número π a reusabilidade e modificabilidade são alcançadas simultaneamente. A reusabilidade é obtida porque o identificador da constante poderá ser usado em diversos pontos do mesmo programa ou até de outros programas para designar a aproximação do número π. A modificabilidade também é atingida porque em caso de necessidade de melhoria na precisão da aproximação basta se alterar o código na definição da constante. Nenhuma outra modificação se fará necessária.

   Argumento semelhante pode ser empregado para a definição de uma função qualquer. Ela poderá ser reutilizada em vários pontos de código. Além disso, qualquer modificação em sua implementação precisa ser feita apenas na sua definiçao, sem que isso provoque modificações nos códigos usuários dessa função.

5. Identifique situações nas quais a busca por eficiência computacional compromete a portabilidade de LPs e vice-versa.

A opção de deixar a definição do intervalo de valores dos tipos primitivos de C para o implementador do compilador permite que esses intervalos sejam ajustados de tal modo que as operações sobre os tipos primitivos tenham correspondentes diretas no hardware da máquina. Essa opção permite a criação de código mais eficiente. Por outro lado, a portabilidade é sacrificada pois a definição de intervalos de valores específicos para as plataformas pode ocasionar programas que não funcionam corretamente em duas plataformas distintas.

Outra situação ocorre quando se dá flexibilidade ao compilador para otimizar código em linguagens que possibilitam a construção de expressões com efeitos colaterais, como é o caso de C. Esse tipo de propriedade pode gerar expressões cujo resultado pode variar de compilador para compilador. Assim, a busca por eficiência computacional nesse caso compromete a portabilidade.

6. Uma LP sempre pode ser implementada usando tanto o método de compilação quanto o de interpretação? Em caso positivo, discuta se existem LPs que se ajustam melhor a um método de implementação do que a outro. Em caso negativo, apresente um exemplo de uma LP na qual só se pode utilizar um método de implementação e justifique.

   Sim. Compilação e interpretação são modos distintos de fazer a tradução, mas ambas têm um mesmo poder expressivo. A diferença principal está em propriedades desses mecanismos tais como eficiência e flexibilidade.

   Existem LPs que se ajustam a um melhor método do que outro. LPs que requerem a declaração de tipos das variáveis, como C, são projetadas já visando a compilação, embora nada impeça de se construir interpretadores de C. Por sua vez, linguagens com ampla tipagem dinâmica, como BASIC, e que permitem a construção dinâmica de expressões ou comandos, como LISP, são mais ajustadas à interpretação, uma vez que é necessário avaliar e interpretar as expressões e comandos dinamicamente. Nesses casos, o método compilado requer que seja gerado código capaz de interpretar esses comandos e expressões.

7. Faça uma análise léxica, sintática e semântica das seguintes linhas de código C e descreva quais as conclusões obtidas:

   ```
   int a, i;
   int b = 2, c =3;
   a = (b + c) * 2;
   i = 1 && 2 + 3 | 4;
   ```

Análise léxica:
int :: tipo de dados
a :: identificador
, :: símbolo
i :: identificador
; :: símbolo
b :: identificador
= :: símbolo

2 :: número
c :: identificador
3 :: número
(:: símbolo
+ :: símbolo
) :: símbolo
* :: símbolo
1 :: número
&& :: símbolo
3 :: número
| :: símbolo
4 :: número

Análise sintática:

Considerando a seguinte gramática de C:

<declaração> ::=
<tipo> <var.> <pv> | <tipo> <ident> <igual> <valor> <pv>
<var.> ::= <ident> | <ident> <virg.> <variaveis>
<igual> ::= '='
<pv> ::= ';'
<ident> ::= <letra> | <letra><ident>
<seqletras> ::= a|b|c|d|e|...|x|y|z
<atrib> ::= <var> <igual> <expressao> <pv>
<expressão> ::=
<valor> | <valor> <operador> <expressão> | (<expressao>)
<valor> ::= <número> | <sinal><número>
<número> ::= <semsinal> | <semsinal>.<semsinal>
<semsinal> ::= <digito> | <digito><semsinal>
<digito> ::= 0|1|2|3|4|5|6|7|8|9
<sinal> ::= +|-
<operador> ::= +|-|/|*|&&|'|'

int a, i;
int b=2, c=3;
Sintaxe válida para definições

a = (b + c) * 2;
i = 1 && 2 + 3 | 4;
Sintaxe válida de expressões

Análise semântica:

int a, i;
Declara e reserva espaço em memória para duas variáveis inteiras a e i;

int b=2, c=3;
Declara, reserva espaço e inicializa duas variáveis inteiras b e c;

a = (b+c)*2;
Atribui à variável a o valor da conta (b+c)*2.

i=1 && 2 +3 | 4;
Atribui à variável i o valor da expressão (1 && 2 + 3 | 4)

Conclusões:
Programa de sintaxe válida que calcula o valor 10 para a e o valor 1 para i.

8. Enumere e explique os principais fatores que influenciaram a evolução das LPs imperativas.

    a) Arquitetura de Computadores: A forma como computadores são implementados influencia o uso do conceito de variáveis como representação para células de memória e no uso de comandos como instruções.
    b) Produtividade dos programação: Na medida que os sistemas de software cresceram significativamente, a produtividade dos programadores ser tornou o grande gargalo para a disseminação dos computadores. A necessidade de melhorar a produtividade dos programadores levou a incorporação dos conceitos de programação estruturada, abstração de dados e orientação a objetos nas LPs imperativas.
    c) Eficiência de uso: Para que os recursos computacionais pudessem ser melhor utilizados por um ou mais usuários foram incorporados os conceitos de programação concorrente em LPs.

9. Induzir a legibilidade, confiabilidade e reuso de programas são algumas das propriedades desejáveis em Linguagens de Programação. Mostre, através de exemplos (um para cada propriedade) retirados de linguagens de programação conhecidas, como elas podem cumprir estes papéis e justifique os seus exemplos.

Legibilidade

A necessidade de colocação de marcadores de fim de bloco em ADA evita que se construam programas nos quais fica difícil saber a qual *if* pertence o *else* quando se tem dois comandos *if* aninhados.

```
if x < 10 then
  if y < 20 then
    y := x + y;
  end if;
else
  x := x + y;
end if;
```

Confiabilidade

A verificação estática dos tipos dos parâmetros impedem a utilização inapropriada de funções.

```
int f (struct data i);
```

f(3); // erro de compilação

Reuso

Subprogramas facilitam a reusabilidade. Por exemplo, a função *printf* de C é um subprograma altamente reutilizado.