Capítulo 5

Linguagens de programação para STR

Introdução

- Questões a tratar para STR
  - Estimativa de tempos execução (performance)
    - Problemas de 'deterioração'
  - Robustez
    - Tratamentos de erros, tipagens, etc
  - Recursos de implementação
  - Outros critérios
    - Produtividade
    - Manutenção a médio prazo
    - Plataforma
    - Quantidade código produzida
    - Maturidade compilador
  - Linguagens de projetos legados (ex. ADA, Cobol, etc)
  - A questão das construções "não-analizáveis" (e.g., LOOPs)

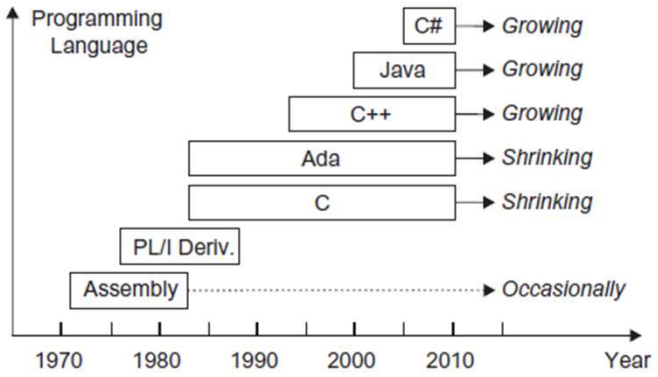
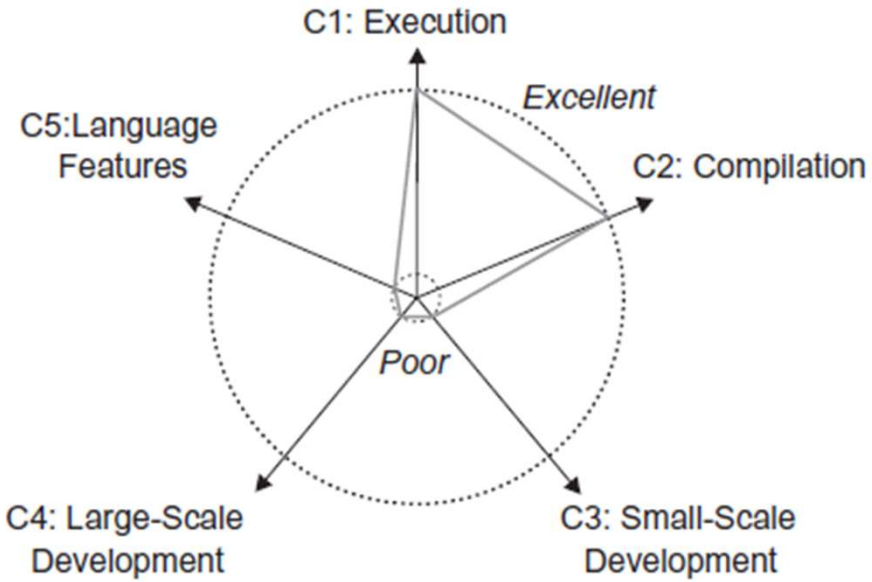


Figure 4.1. Mainstream usage of real-time programming languages over the years (the year limits are approximate).

Introdução

- Qual linguagem escolher?
  - Critério de Cardelli



C1. Economy of Execution. How fast does a program run?

C2. Economy of Compilation. How long does it take to go from multiple source files to an executable file?

C3. Economy of Small-Scale Development. How hard must an individual programmer work?

C4. Economy of Large-Scale Development. How hard must a team of programmers work?

C5. Economy of Language Features. How hard is it to learn or use a programming language?

Introdução

- Questão de padronização de elementos de programação para o time
  - Objetivo: programas mais fáceis de ler a analisar

> - Header format
> - Frequency, length, and style of comments
> - Naming of classes, data, files, methods, procedures, variables, and so forth
> - Formatting of program source code, including use of white space and indentation
> - Size limitations on code units, including maximum and minimum number of code lines, and number of methods used
> - Rules about the choice of language construct to be used; for example, when to use `case` statements instead of nested `if-then-else` statements

  - Padrões de programação

Tipos de linguagens para STR

- A) Assembly
  - Altamente previsível/preditível
  - Muito usada para device drivers
  - Recursos críticos ou específicos de CPU
  - Tempo mínimo de execução
  - Depuração de baixo nível (logic analyzer)
- B) Procedimental
  - Boa modularidade
  - Tipagem forte
    - trucagem
  - Abstração de tipos
  - Passagem de parâmetros versátil
    - "Por valor" x "por referência"
  - Boa alocação de memória
  - Exception handling (overflow, divisão zero, etc)

```
int x,y;
float a,b;
y=x*a+b;
```

Tipos de linguagens para STR

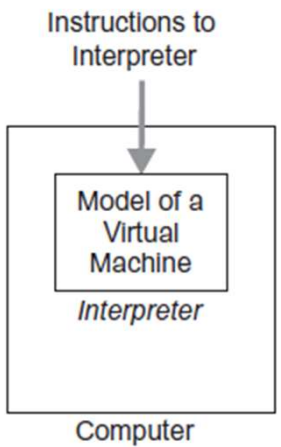
- C) Orientada a objetos
  - Melhor produtividade
  - Melhor re-uso
  - Suporte a abstração e polimorfismo
    - Exemplo: classe pixel (posição, cor, brilho, atividade, etc)
  - Avanços em messaging
  - Encapsulamento de dados
  - ‘Garbage collection’ atuantes
  - Facilita sincroniza de objetos e threads
    - Sincronia de blocos de código por mutex
    - Propriedade única de objetos por threads (método ‘dispose’)

Tipos de linguagens para STR

- Baseado em [1], são consideradas as linguagens mais ‘comuns’ para STR:
  - C
    - Elementos específicos para desempenho como ‘register’, static, ‘volatile’, etc
    - Trabalha forte com chamadas por referência (ponteiros)
    - Boa flexibilidade
    - Facilidade de levantamento de fluxo (estimar temporização)
  - C++
    - É uma versão ‘hibrida’ de orientação a objeto
    - Implementação de diretivas de pré-processamento (#define)
    - Tipagem fraca
    - Assim como C, C++ a maioria dos bugs estão no uso de ponteiros (gasta-se muito tempo depuração)
      - Exemplo: ausência tipo de dados string
    - Não tem garbage collection

Tipos de linguagens para STR

- C#
  - A grosso modo: mistura de C++ com Java
  - Operado em tempo real (run time .NETFramework)
  - Suporte a “unsafe code” (referneciar especificamente regiões de memória)
  - Tem um garbage collection que permite trabalho com DMA
  - Amplo suporte a thread e sincronização
  - Boa performance com ponto flutuante
  - Usado só para STR tipo ‘soft’ e ‘firm’.
- JAVA
  - Operado em tempo real (run time Java virtual machine)
    - Tem opção de “bare metal” também
  - Tudo em java é implementado como uma classe
  - Passagem de arrays e objetos só por referência



Otimizações de código

- Tipos mais comuns a observar:
  - Uso de identidades aritméticas

```
y = x*0.5; //melhor
Y= x/2.0; //pior
```

  - Uso de funções intrínsecas

```
x=6+a*b;
y=a*b+z;
```



```
t=a*b;
x=6+t;
y=t+z;
```

  - Duplicações de constantes

```
Y= 2.0*x*4.0; //pior
Y= 8.0*x; //melhor
```

Otimizações de código

- Tipos mais comuns a observar:
  - Remoção de loop invariante

```
x=100;
while (x>0)
 x=x-y+z;
```

→

```
x=100;
t=y+z;
while (x>0)
 x=x-t;
```

  - Eliminação de indução de loop

```
for (i=1;i<=10;i++)
 a[i+1]=1;
```

→

```
for (j=2;j<=11;j++)
 a[j]=1;
```

Otimizações de código

- Tipos mais comuns a observar:
  - Remoção de código morto ou de depuração

```
#ifdef DEBUG
 {
 ...
 }
#endif
```

  - Verificação booleana antecipada

```
if ((x>0) && (y>0))
 z=1;
```

→

```
if (x>0)
 if (y>0)
  z=1;
```

  - Desenrolamento de loop

```
for (i=1;i<=6;i++)
 a[i]=a[i]*8;
```

→

```
a[1]=a[1]*8;
a[2]=a[2]*8;
a[3]=a[3]*8;
a[4]=a[4]*8;
a[5]=a[5]*8;
a[6]=a[6]*8;
```

Referências

- [1] Real-time systems design and analysis – tools for the practitioner