Projeto de funções

Programação Funcional

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Vamos voltar ao problema da Márcia.

Depois que você fez o programa para o Alan, a Márcia, amiga em comum de vocês, soube que você está oferecendo serviços desse tipo e também quer a sua ajuda. O problema da Márcia é que ela sempre tem que fazer a conta manualmente para saber se deve abastecer o carro com álcool ou gasolina. A conta que ela faz é verificar se o preço do álcool é até 70% do preço da gasolina, se sim, ela abastece o carro com álcool, senão ela abastece o carro com gasolina. Você pode ajudar a Márcia também?

Como proceder para projetar este programa?

Projeto de funções

Vamos usar um processo de projeto de funções

- Análise
- Definição dos tipos de dados
- Especificação
- Implementação
- Verificação
- Revisão

Esse processo é inspirado no livro How to Design Programs.

Objetivos

Cada etapa tem um objetivo e depende das etapas anteriores

- Análise: identificar o problema a ser resolvido
- Definição dos tipos de dados: definir como as informações serão representadas
- Especificação: especificar com precisão o que a função deve fazer
- Implementação: implementar a função de acordo com a especificação
- Verificação: verificar se a implementação está de acordo com a especificação
- Revisão: identificar e fazer melhorias na especificação e implementação

Note que às vezes precisamos alterar a ordem das etapas, por exemplo, às vezes estamos na implementação e encontramos uma nova condição e devemos voltar e alterar a especificação.

Mas devemos evitar fazer a implementação diretamente!

Mas esse processo serve para projetar funções, como projetamos programas?

Um programa é composto de várias funções, então temos que decompor o programa em funções e aplicar o processo para projetar cada função.

Vamos treinar com problemas simples, de uma função, depois vamos utilizar o processo em problemas mais elaborados.

Depois que você fez o programa para o Alan, a Márcia, amiga em comum de vocês, soube que você está oferecendo serviços desse tipo e também quer a sua ajuda. O problema da Márcia é que ela sempre tem que fazer a conta manualmente para saber se deve abastecer o carro com álcool ou gasolina. A conta que ela faz é verificar se o preço do álcool é até 70% do preço da gasolina, se sim, ela abastece o carro com álcool, senão ela abastece o carro com gasolina. Você pode ajudar a Márcia também?

Análise

- Quais informações são relevantes e quais podem ser descartadas?
- Existe alguma omissão?
- Existe alguma ambiguidade?
- Quais conhecimentos do domínio do problema são necessários?
- O que deve ser feito?

Resultado

Determinar o combustível que será utilizado. Se o preço do álcool for até 70% do preço da gasolina, então deve-se usar o álcool, senão a gasolina.

Análise

Determinar o combustível que será utilizado. Se o preço do álcool for até 70% do preço da gasolina, então deve-se usar o álcool, senão a gasolina.

Definição dos tipos de dados

- Quais são as informações envolvidas no problema?
- Como as informações serão representadas?

Resultado

Informações: preço do litro do combustível e o tipo do combustível. Representações:

```
/// O preço do litro do combustível, deve ser
/// um número positivo.
type Preco =
  Float

/// O tipo do combustível, deve ser "Álcool" ou "Gasolina
type Combustivel =
  String
```

Análise

Determinar o combustível que será utilizado.

Se o preço do álcool for até 70% do preço da gasolina, então deve-se usar o álcool, senão a gasolina.

Tipos

```
/// O preço do litro do combustível,
/// deve ser um número positivo.
type Preco =
  Float

/// O tipo do combustível,
/// deve ser "Álcool" ou "Gasolina".
type Combustivel =
  String
```

Especificação

- Assinatura da função
- Propósito (o que a função faz)
- Exemplos de entrada e saída

Resultado

```
/// Encontra o combustível que deve
/// ser utilizado no abastecimento.
/// Produz "Álcool" se *preco_alcool*
/// for até 70% do *preco_gasolina*,
/// produz "Gasolina" caso contrário.
fn seleciona_combustivel(
  preco_alcool: Preco,
  preco_gasolina: Preco,
) -> Combustivel {
  todo
}
```

Apesar das anotações de tipos serem opcionais, de agora em diante, vamos **sempre** colocar os tipos das entradas e saída das funções.

Exemplos

- Álcool 3.00, Gasolina 4.00, produz “Gasolina” ($3.00 < 0.7 \times 4.00$ é falso)
- Álcool 2.90, Gasolina 4.20, produz “Álcool” ($2.90 < 0.7 \times 4.20$ é verdadeiro)
- Álcool 3.50, Gasolina 5.00, não está claro na especificação o que fazer quando o preço do álcool é exatamente 70% ($3.50 = 0.7 \times 5.00$)!

Precisamos tomar uma decisão e modificar o propósito para ficar mais preciso. Vamos assumir que exatamente 70% também implica no uso do álcool (quais são as outras possibilidades?). O propósito modificado fica

```
/// Encontra o combustível que deve ser utilizado no abastecimento. Produz
/// "Álcool" se *preco_alcool* for menor ou igual a 70% do *preco_gasolina*,
/// produz "Gasolina" caso contrário.
fn seleciona_combustivel(preco_alcool: Preco, preco_gasolina: Preco) -> Combustivel {
  todo
}
```

Propósito

No propósito da função descrevemos **o que** a função faz, e não **como** ela faz (que é a implementação - às vezes precisamos dizer como ela faz, mas isso é raro).

No propósito também informamos as garantias da saída e as restrições sobre os parâmetros.

Diferença de “o que” e “como”

Número par

- O que: verifica se um número é par
- Como: faz o resto da divisão do número por 2 e compara com 0; ou; faz a divisão inteira do número e multiplica por 2 e compara com o número

Ordenação

- O que: ordena os elementos de uma lista em ordem não decrescente
- Como: ordenação por seleção, por inserção, por intercalação, etc

Para saber se a especificação está adequada, faça a seguinte pergunta:

Um outro desenvolvedor, que não tem acesso ao problema original e nem a análise, tem as informações necessárias na especificação para fazer uma implementação e verificação inicial?

Se a resposta for sim, então a especificação está adequada; senão, ela está incompleta.

```
/// Encontra o combustível que deve
/// ser utilizado no abastecimento.
/// Produz "Álcool" se *preco_alcool*
/// for menor ou igual 70% do
/// *preco_gasolina*, produz "Gasolina"
/// caso contrário.
fn seleciona_combustivel(
  preco_alcool: Preco,
  preco_gasolina: Preco,
) -> Combustivel
```

3.00, 4.00, “Gasolina” ($3.00 \leq 0.7 \times 4.00$ é falso)

2.90, 4.20, “Álcool” ($2.90 \leq 0.7 \times 4.20$ é verdade)

3.50, 5.00, “Álcool” ($3.50 \leq 0.7 \times 5.00$ é verdade)

Implementação

- Veremos diversas estratégias de implementação ao longo da disciplina.
- Uma delas é a **direta**. Se a forma de calcular a resposta é sempre a mesma (não depende de uma condição), então escrevemos a expressão da resposta diretamente.
- Outra é a **análise de casos**. Identificamos as formas de resposta e a condição para cada forma, então escrevemos um caso para cada forma.

Temos duas formas de resposta, `"Álcool"` e `"Gasolina"`, portanto, precisamos de uma condição para distinguir quando utilizar cada resposta. No caso, a resposta é `"Álcool"` se `preco_alcool` é menor ou igual a 70% do preço de `preco_gasolina`; e `"Gasolina"` caso contrário.

```
/// Encontra o combustível que deve ser utilizado no abastecimento. Produz
/// "Álcool" se *preco_alcool* for menor ou igual a 70% do *preco_gasolina*,
/// produz "Gasolina" caso contrário.
fn seleciona_combustivel(preco_alcool: Preco, preco_gasolina: Preco) -> Combustivel {
  case preco_alcool <=. 0.7 *. preco_gasolina {
    True -> "Álcool"
    False -> "Gasolina"
  }
}
```

```
fn seleciona_combustivel(
  preco_alcool: Preco,
  preco_gasolina: Preco,
) -> Combustivel {
  case preco_alcool <=.
       0.7 *. preco_gasolina {
    True -> "Álcool"
    False -> "Gasolina"
  }
}
```

3.00, 4.00, então “Gasolina”.

2.90, 4.20, então “Álcool”.

3.50, 5.00, então “Álcool”.

Verificação

- A implementação está de acordo com a especificação?

Resultado

Vamos utilizar os exemplos que criamos na especificação para verificar se a resposta é a esperada.

```
> seleciona_combustivel(3.0, 4.0)
"Gasolina"
```

```
> seleciona_combustivel(2.9, 4.2)
"Álcool"
```

```
> seleciona_combustivel(3.5, 5.0)
"Álcool"
```

Preparem-se, agora vem uma sequência de muitas perguntas!

De forma geral, o fato de uma função produzir a resposta correta para alguns exemplos, implica que a função está correta? Não!

Então porque “perder tempo” fazendo os exemplos? O primeiro objetivo dos exemplos é ajudar o projetista a entender como a saída pode ser obtida a partir das entradas. O segundo é ilustrar o seu funcionamento para que a especificação fique mais clara. Depois esses exemplos podem ser usados como uma forma inicial de verificação, que mesmo não mostrando que a função funciona corretamente, aumenta a confiança do desenvolvedor que o código está correto.

Já que os exemplos são uma verificação inicial, então temos que ampliar a verificação? Sim! De que forma? Testes de propriedades, fuzzing, etc. Para esta disciplina, vamos utilizar apenas os exemplos para fazer a verificação.

Nós fizemos os exemplos em linguagem natural e no momento de verificar os exemplos nós "traduzimos" para o Gleam e fizemos as chamadas da funções de forma manual no repl.

Podemos melhorar esse processo? Sim!

Vamos escrever os exemplos diretamente em forma de código de maneira que eles possam ser executados automaticamente quando necessário.

```gleam
import sgleam/check

fn seleciona_combustivel(preco_alcool: Preco, preco_gasolina: Preco) -> Combustivel {
  case preco_alcool <=. 0.7 *. preco_gasolina {
    True -> "Álcool"
    False -> "Gasolina"
  }
}

pub fn seleciona_combustivel_examples() {
  check.eq(seleciona_combustivel(3.0, 4.0), "Gasolina")
  check.eq(seleciona_combustivel(2.9, 4.2), "Álcool")
  check.eq(seleciona_combustivel(3.5, 5.0), "Álcool")
}
```

Para executarmos os testes usamos o comando

No Windows

```
.\sgleam -t arquivo.gleam
```

No Linux ou Mac

```
./sgleam -t arquivo.gleam
```

A saída será algo como

```
Running tests...
3 tests, 3 success(es), 0 failure(s) and 0 errors.
```

Por que um exemplo pode falhar?

- O exemplo está errado
- A implementação está errada
- O exemplo e a implementação estão errados

```
/// O preço do litro do combustível,
/// deve ser um número positivo.
type Preco = Float
/// O tipo do combustível,
/// deve ser "Álcool" ou "Gasolina".
type Combustivel = String
fn seleciona_combustivel(
  preco_alcool: Preco,
  preco_gasolina: Preco,
) -> Combustivel {
  case preco_alcool <=.
       0.7 *. preco_gasolina {
    True -> "Álcool"
    False -> "Gasolina"
  }
}
```

Revisão

- Podemos melhorar a especificação e o código?
- Podemos fazer simplificações eliminando casos especiais (generalizando)?
- Podemos criar abstrações (definição de constantes e funções)?
- Podemos renomear os objetos?

Resultado

Os tipos de dados permitem representar informações inválidas.

Veremos depois como lidar com essa questão.

Referências

Básicas

- Vídeos BSL
- Vídeos How to Design Functions