Sucesso e Falha. Um Paradigma Poderoso para Controle de Fluxo

Introdução

Para os exemplos será utilizada a linguagem Icon, criada por Ralph Griswold em 1977 como uma modernização de SNOBOL (de 1962 também criada por ele).

Sucesso e Falha

Mas o que significa **sucesso e falha**? Bem, na maioria das linguagens de programação, o controle de fluxo é baseado em estruturas condicionais como *if-else*, *laços* e *exceções*. Uma expressão retorna um valor lógico (falso ou verdadeiro) e o fluxo será gerenciado por uma estrutura de decisão.

O paradigma de ***sucesso e falha*** é o conceito fundamental na linguagem de programação Icon, que introduz uma abordagem única ao controle de fluxo e à avaliação de expressões. Em vez de uma valor lógico, uma expressão pode ter sucesso ou falhar.

Características principais:

1. **Geração de valores**: Quando uma expressão tem sucesso, ela pode gerar não apenas um, mas potencialmente múltiplos valores sequencialmente.
2. **Controle de fluxo baseado em sucesso/falha**: O fluxo de controle é determinado pelo sucesso ou falha das expressões, não apenas por valores booleanos. Por exemplo, um loop continua enquanto uma expressão tiver sucesso. Toda uma expressão pode ser abortada em caso de falha. Quase um commit/rollback.
3. **Backtracking automático**: Quando uma expressão falha, Icon pode automaticamente retroceder e tentar outras alternativas.
4. **Operadores especiais**: Icon possui diversos operadores específicos para manipular sucesso e falha, como o operador de alternativa (|) que tenta a expressão à direita se a expressão à esquerda falhar (considere com um ***ou*** em outras linguagens).

Este paradigma torna Icon particularmente eficiente para processamento de texto, análise de padrões e outras tarefas que envolvem busca e exploração de alternativas, pois ele incorpora nativamente conceitos de não-determinismo e backtracking que outras linguagens precisam implementar explicitamente.

Ela também poderia ser classificada como "Goal-Directed Language" pois, assim como SNOBOL4, existe todo um conjunto de instruções direcionados ao objetivo de análise e processamento de textos. Mas eu acho que sucesso e falha diferencia da maioria das linguagens conhecidas.

Sucesso e falha na prática

> A sintaxe de Icon é uma mistura de C e Pascal. O mais importante a saber é que a atribuição utiliza o símbolo `:=` enquanto que o `=` é utilizado para testar igualdade.

Vejamos como as coisas funcionam para que o conceito fique um pouco mais claro. Supondo o seguinte programa em Icon (sim, compliquei um pouco mas explico depois):

```
procedure main()
    v := 45
    write("v = ",v)
    v := 12 > 44 < 98
    write("v = ",v)
    v := 48 > 17 > 11
    write("v = ",v)
    v := 16 < ((c := 5) | 12 | 37 | 44)
    write("v = ",v)
    write("c = ",c)
    write(4 < find("a","abracadabra"))
end
```

O resultado da execução do programa será:

```
v = 45
v = 45
v = 11
v = 37
c = 5
6
```

- O início não tem maiores problemas. Foi atribuído o valor 45 para a variável `v`.
- No segundo caso, será atribuído a `v` o valor da expressão `12 > 44 < 98`. Como `12 > 44` **falha** já que 12 não é maior do que 44, então *toda a expressão falha*, inclusive a atribuição. Como resultado, o valor de v permanece inalterado.
- No terceiro caso, a expressão é `48 > 17 > 11`. Como 48 é maior que 17, temos um **sucesso**. Então é retornado o último valor que é 17. Continuamos com a avaliação da

expressão. Como 17 é maior que 11, um novo sucesso e retorna o último valor que é 11. Finalmente o valor **11** é atribuído a variável.

- O quarto caso é um pouco mais complexo. As expressões entre parênteses são avaliadas primeiro (como qualquer outra linguagem). Então, primeiro a variável c recebe o valor 5. Como é sucesso, retorna 5. A expressão pode ser reescrita como `16 < (5 | 12 | 37 | 44)`. Traduzindo para outras linguagens, temos uma expressão tipo `16 < 5 or 16 < 12 or 16 < 37 or 16 < 44`. O **or** fará com que, caso a expressão falhe, verifique o próximo valor fornecido e assim sucessivamente até obter algum sucesso ou todas falhares e retorna uma falha. As duas primeiras comparações falham. A terceira, 16 < 37 retorna sucesso e o valor é 37. Não é mais necessário verificar as próximas expressões. Então a variável recebe o valor **37**.

- o valor de c foi atribuído na expressão anterior.

- Finalmente, será procurado por um "a" na string onde sua posição seja maior ou igual a 4. Se não for encontrado, nada acontece. Se for, a posição é retornada.

Acho que ficou bem claro que, em Icon, uma comparação **não** retorna *falso* ou *verdadeiro*. Apenas **sucesso** e um ou mais **valores** ou **falha**. Também é possível inserir outros comandos e eles irão retornar sucesso ou falha. Se algum falha, toda a expressão falha.

Outras expressões e operadores

Pretendo colocar exemplos em Icon e Python, ***apenas*** para mostrar as diferenças/semelhanças. Pelo meu conhecimento restrito de Python, se algum exemplo estiver muito fora da realidade, peço desculpas e aceito correções. Outras linguagens poderão ser mais ou menos parecidas.

if then else

Mais ou menos como em outras linguagens. Se a expressão após o **if** retornar sucesso, então o bloco após o **then** será executado. Se resultar em um falha, o bloco após o **else** será executado. Se eu quiser saber se uma string contém uma substring e, em caso afirmativo, mostrar a posição onde ela se encontra, em Icon seria assim:

```
if x := find("na", "banana") then
    write("Encontrado na posição: ", x)
else
    write("Não encontrado")
```

Se a substring não for encontrada, toda a expressão falha e o fluxo passa para o `else`. Caso seja encontrada, retorna a posição (o que significa sucesso) e o resto da expressão é avaliada. Se a posição retornada puder ser atribuída a variável x, também retorna sucesso e a expressão após o `then` é avaliada.

Uma possível equivalência em Python

```
x = "banana".find("na")

if x != -1:
    print("Encontrado na posição:", x)
else:
    print("Não encontrado")
```

Bem, Python 3.8 (PEP 572) foi incluído o operador **:=** (walrus)

Apesar da sintaxe ficar mais parecida, ainda assim é necessário efetuar a comparação com o retorno da busca.

```
texto = "banana"
substring = "na"

if (x := texto.find(substring)) != -1:
    print("Encontrado na posição:", x)
else:
    print("Não encontrado")
```

every

A finalidade do **every** é exaurir um gerador, isto é, solicitar os valores gerados até que ele falhe por não ter mais nenhum para oferecer. Pode ser considerado como um `for i in` de Python. Se temos o seguinte trecho em Python para imprimir os valores `1, 2, 3, 4, 5`:

```
for i in range(1, 6):
    print(i, end=" ")
```

O equivalente em Icon seria

```
every i := 1 to 5 do
    writes(i," ")
```

Bem parecido. Como uma expressão irá retornar um valor se for bem sucedida, atribuir o valor a uma variável apenas para imprimir seria mais perda de tempo. Em Icon, uma versão mais idiomática é:

```
every writes(1 to 5," ")
```

É possível ter loops aninhados da seguinte forma (logo você irá saber o significado de **&**):

```
    every i :=1 to 3 & j := 1 to 3 & i ~= j do
        write(i," : ",j)
```

o que equivaleria em Python a

```
for i in range(1, 4):
    for j in range(1, 4):
        if i != j:
            print(f"{i} : {j}")
```

e o resultado será

```
1 : 2
1 : 3
2 : 1
2 : 3
3 : 1
3 : 2
```

exp1 & exp2

Executa **exp2** apenas se **exp1** retorna sucesso.

```
3 > 2 & write("A expressão é verdadeira.")
```

em python ficaria:

```
if 3 > 2:
    print("A expressão é verdadeira.")
```

Um exemplo um pouco mais complexo (backtracking). Talvez você já tenha visto algo semelhante.

```
if (i := 1 to 10) & (j := 1 to i) & (i + j = 11) then
    write(i, " + ", j, " = 11")
```

resulta em

```
6 + 5 = 11
```

Em Python seria mais ou menos assim:

```python
encontrou = False
for i in range(1, 11):
    if encontrou:
        break
    for j in range(1, i + 1):
        if i + j == 11:
            print(f"{i} + {j} = 11")
            encontrou = True
            break
```

Talvez fosse melhor colocar em uma função para utilizar apenas **return** quando `i + j` for igual a `11`.

resume

Utilizada para a criação de geradores personalizados. Basicamente retorna o valor e suspende a execução da função até que um novo valor seja solicitado. Por exemplo:

```
procedure primos(n)
    i := 2
    while i <= n do {
        j := 2
        while j < i do {
            if i % j == 0 then
                break
            j := j + 1
        }
        j == i & suspend i
        i := i + 1
    }
end

procedure main()
    every writes(primos(10)," ")
    write()
    every writes(primos(100)," ")
end
```

Irá imprimir

```
2 3 5 7
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Em Python seria utilizado yield. O código fica bem parecido:

```
def primos(n):
    i = 2
    while i <= n:
        j = 2
        while j < i:
            if i % j == 0:
                break
            j += 1
        else:
            yield i
        i += 1

print(*primos(10), end=" ")
print()
print(*primos(100), end=" ")
```

! (gera elementos)

Gera todos os elementos de um argumento. Novamente semelhante ao `for i in` existente em outras linguagens com algumas peculiaridades.

- `!n` irá gerar os inteiros 1,2,3,4,....,n
- `!3.14` é o equivalente a converter o número de ponto flutuante para string e retornar seus elementos; no caso `"3", ".", "1", "4"`
- `!"casa"` gera `"c", "a", "s", "a"`
- existem outros argumentos como listas, sets, arquivos, etc.

n1 to n2 [by n3]

Parecido com *range* de Python. Gera uma sequência numérica iniciando em **n1** e incrementada por 1 até **n2** (inclusive). O incremento pode ser alterado por **by**. Neste caso , o valor final pode não ser necessariamente **n2**.

- `1 to 6` : gera `1, 2, 3, 4, 5, 6` (veja que pode ser substituído por `!5`)
- `1 to 6 by 2` : resulta `1, 3, 5`
- `6 to 1 by -1` : gera `6,5,4,3,2,1`
- `5 to 1` : falha e não gera nada

seq

Gera uma sequência infinita nos mesmos moldes de n1 to n2 by n3.

- `seq()` : gera uma sequência 1,2,3,4,.....
- `seq(i)` : gera uma sequencia i,i+1,i+2,i+3,....
- `seq(i,j)` : gera uma sequência i,i+j,i+2j, ...

obs.: j não pode ser zero e não trabalha com inteiros arbitrários.

\n (limita geração)

Irá limitar a geração da sequência em **n** valores. O exemplo abaixo procura pelas vogais na string informada. A primeira retorna a posição das 7 vogais encontradas. A segunda, o retorno é limitado a três.

```
procedure main()
    every writes(upto('aeiou',"minha casa, meu lar") || " ")
    write()
    every writes(upto('aeiou',"minha casa, meu lar")\3 || " ")
end
```

e o resultado será:

```
2 5 8 10 14 15 18
2 5 8
```

\var (testa não nulo)

Caso a variável possua algum valor, retorna o valor. Caso contrário a expressão falha.

/var (testa se nulo)

Se variável existir e tiver algum valor, a expressão irá falhar. Caso a variável não exista, retorna uma referência para que a expressão seja avaliada.

```
procedure main()
    x := \y | -11
    write("x = ",x)
    /x := 50
    /y := 13
    write("x = ",x)
    write("y = ",y)
    x := \y | -11
    write("x = ",x)
end
```

o resultado será

```
x = -11
x = -11
y = 13
x = 13
```

- `x := \y | -11` : como `y` era `&null` então x recebeu 11
- `/x := 50` : como já existia valor em x, a expressão falhou

- `/y := 13` : como y não existia, foi atribuí o valor a y
- `x := \y | -11` : como a primeira expressão mas, como agora y possui um valor, é assinalado para x

|exp (repete alternadamente)

Gera uma sequência infinita com a expressão informada. `|2` irá gerar `2,2,2,2,...` ao passo que `|(1 to 3)` irá gerar `1,2,3,1,2,3,1,2,3,....` e `| (0 | 1)` gera `0, 1, 0, 1, 0, 1, ...` (ok, agora você já sabe a necessidade do **\n**).

Se a sequência falha, então não será necessário o limitador **\n**. Como por exemplo, para ler todo o conteúdo de um arquivo, `|read()` a leitura será efetuada até chegar ao final do arquivo quando então retornará uma falha e interrompe a sequência.

Bônus

Alguns pequenos programas que eu achei interessantes para demonstrar outras características da linguagem.

Processamento de texto

É uma área onde a linguagem se sai muito bem. Como a origem é SNOBOL ("StriNg Oriented and symBOlic Language") é o mínimo que se espera. Por exemplo, dadas duas palavras quaisquer, queremos saber onde elas se intersectam (mesma letra nas duas palavras). Tipo palavras cruzadas ou outro passatempo do gênero. Um programa como este resolveria o problema:

```
procedure main()
    cross("casa","carros")
end

procedure cross(word1,word2)
    local i,j
    every (i := upto(word2, word1)) & (j := upto(word1[i], word2)) do {
        every write(right(word2[1 to j - 1], i))
        write(word1)
        every write(right(word2[j + 1 to *word2], i))
        write()
    }
    return
end
```

Executando o programa acima, a saída será (coloquei lado a lado para efeitos de visualização; originalmente será uma solução abaixo da outra)

```
casa          c             c             c
a            casa           a          casa
r             r             r             r
r             r             r             r
o             o             o             o
s             s          casa             s
```

Co-expressions

Apesar de trabalhar com threads, existe o conceito de co-expressions. Basicamente são geradores executados em paralelo. Cada gerador irá gerar apenas um valor por vez

Se eu quiser criar uma tabela ASCII com os 256 caracteres, sendo uma coluna com o decimal, outra octal, hexadecimal e o caractere correspondente, uso o programa abaixo.

```
procedure main()
    write("DEC\tOCT\tHEX\tCHAR")
    dec := create(0 to 255)
    hex_dig := "0123456789ABCDEF"
    hex := create(!hex_dig || !hex_dig)
    oct := create((0 to 3) || (0 to 7) || (0 to 7))
    char := create image(!&cset)
    while write(@dec, "\t", @oct, "\t", @hex, "\t", @char)
end
```

como resultado

```
DEC    OCT    HEX    CHAR
0    000    00    "\x00"
1    001    01    "\x01"
2    002    02    "\x02"
3    003    03    "\x03"
4    004    04    "\x04"
5    005    05    "\x05"
6    006    06    "\x06"
7    007    07    "\x07"
8    010    08    "\b"
9    011    09    "\t"
10    012    0A    "\n"
11    013    0B    "\v"
12    014    0C    "\f"
13    015    0D    "\r"
...
31    037    1F    "\x1f"
32    040    20    " "
33    041    21    "!"
34    042    22    "\""
35    043    23    "#"
36    044    24    "$"
37    045    25    "%"
38    046    26    "&"
39    047    27    "'"
40    050    28    "("
41    051    29    ")"
42    052    2A    "*"
43    053    2B    "+"
44    054    2C    ","
45    055    2D    "-"
46    056    2E    "."
47    057    2F    "/"
48    060    30    "0"
49    061    31    "1"
...
56    070    38    "8"
57    071    39    "9"
58    072    3A    ":"
59    073    3B    ";"
60    074    3C    "<"
61    075    3D    "="
62    076    3E    ">"
63    077    3F    "?"
64    100    40    "@"
65    101    41    "A"
66    102    42    "B"
67    103    43    "C"
...
87    127    57    "W"
```

```
88    130    58    "X"
89    131    59    "Y"
90    132    5A    "Z"
91    133    5B    "["
92    134    5C    "\\"
93    135    5D    "]"
94    136    5E    "^"
95    137    5F    "_"
96    140    60    "`"
97    141    61    "a"
98    142    62    "b"
99    143    63    "c"
100    144    64    "d"
...
122    172    7A    "z"
123    173    7B    "{"
124    174    7C    "|"
125    175    7D    "}"
126    176    7E    "~"
127    177    7F    "\d"
128    200    80    "\x80"
...
253    375    FD    "\xfd"
254    376    FE    "\xfe"
255    377    FF    "\xff"
```

Gráficos

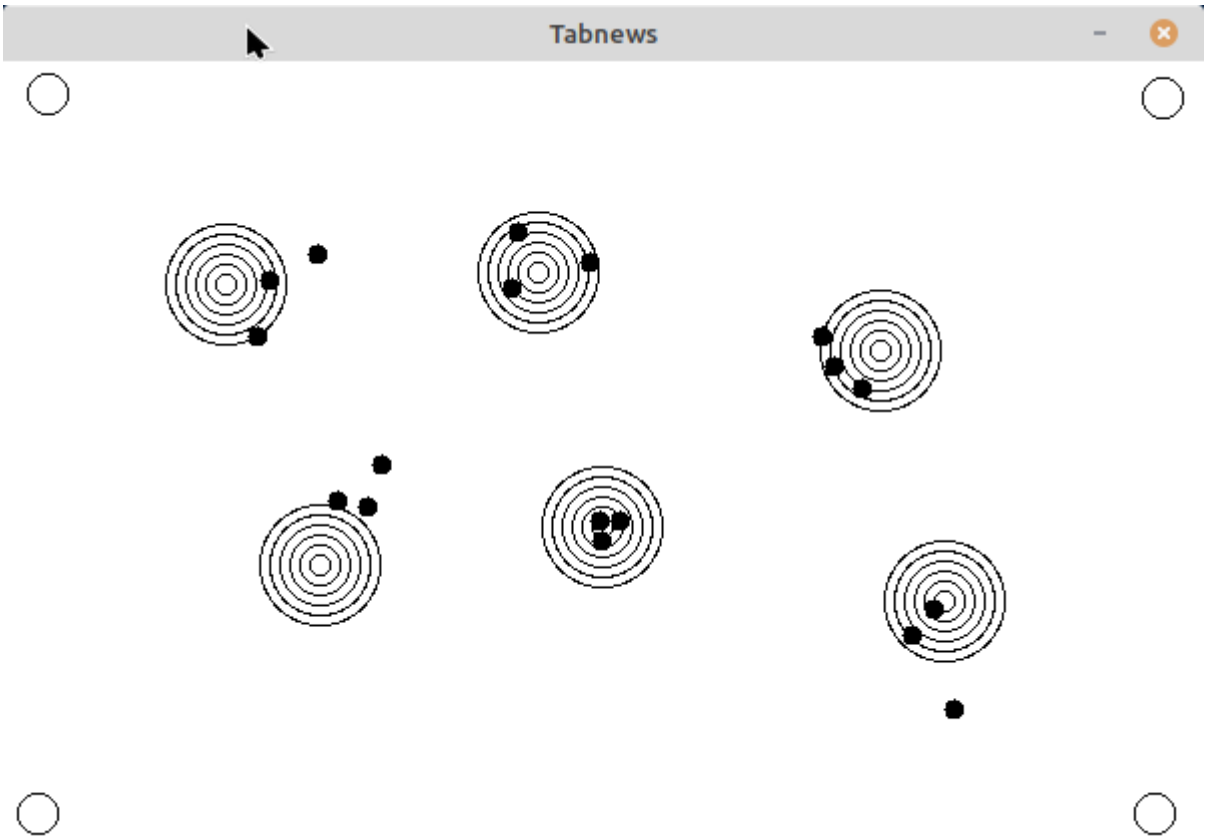
Icon possui muitas facilidades para trabalhar com gráficos. Existe até uma parte para GUI. Mas é meio datada é com aparência motif. Provavelmente não é do agrado da maioria.

```
link graphics
procedure main()
    WOpen("label=Tabnews","size=600,400") | stop("can't open window")
    repeat case Event() of {
        "q"|"Q": exit()
        &lpress: FillCircle(&x, &y, 5)
        &mpress: DrawCircle(&x, &y, 10)
        &rpress: every i:=0 to 30 by 5 do DrawCircle(&x, &y, i)
    }
end
```

- O programa cria uma janela com o título *Tabnews*, dimensões de 600x400 e fica aguardando por eventos específicos como:
- Se for pressionada a tecla "q" o programa encerra.

- Botão esquerdo do mouse desenha um circulo preenchido com a cor atual e raio de 5 pixels nas coordenadas do ponteiro do mouse.
- Botão do centro desenha um círculo com raio de 10 pixels.
- Botão da direita desenha 6 círculos concêntricos com raio entre 0 até 30 pixels e incremento de 5 pixels.

Um exemplo do resultado



Referências

Algumas referências para a linguagem Icon. Você deverá encontrar todas ou a grande maioria no próprio site da linguagem. Só separei alguns caso alguém tenha alguma curiosidade mais específica.

- Fundamentals of the Icon Programming Language (pdf): Uma breve introdução e, depois, um passeio pelos principais aspectos da linguagem (não aborda gráficos). Um bom início para quem deseja saber mais sobre a linguagem.
- The Icon Programming Language : Terceira edição do manual sobre Icon, em pdf.
- Graphics Programming in Icon : Livro em pdf abrangendo a parte gráfica da linguagem.

- The Implementation of the Icon Programming Language : Livro em pdf descrevendo a implementação da linguagem, máquina virtual e outros aspectos. Bom para se aprofundar na linguagem ou se você quiser ~~não faça isso~~ desenvolver a sua.
- Este documento no formato pdf.

TL;DR;

- Icon é uma linguagem interessante pelo paradigma de processo e falha para controle do fluxo do programa.
- Por pedido do autor, o desenvolvimento da linguagem está congelado. Se você deseja algo ainda em desenvolvimento deverá utilizar Unicon. Tudo que tem neste texto continua válido mas Unicon permite trabalhar com protocolos de internet, base de dados, OO, gráficos 3D, suporte a VSCode e outras coisinhas.
- Apresar da grande capacidade de trabalhar com textos, utiliza a tabela ASCII então, se você usa UTF-8, por exemplo, poderá ter problemas. Existe uma versão objecticon OO que trabalha com unicode mas a última versão é de 2021.