Universidade Federal de Santa Catarina
INE5426 - Construção de Compiladores
Grupo Lambari

Projeto 2 - Linguagem Falk (Functional Calculator)

Relatório Final da Linguagem

Pontos implementados:

- Nova linguagem (+1)
- Interpretador (+2)
- Escopos dinâmicos (+1)
- Operações com conjuntos de dados (?)

Equipe de desenvolvimento:

- Marleson Graf - Arquiteto do Sistema, Projetista de Linguagem
- Ghabriel Nunes - Testador, Projetista de Linguagem
- Vinicius Freitas - Gerente de Projetos, Projetista de Linguagem

Informações básicas e premissas do projeto:

Falk é uma proposta de linguagem simples, altamente matemática, de uso rápido e fácil. Suas principais funcionalidades envolvem a resolução de problemas matemáticos disponibilizando operadores aritméticos usuais, permitindo também operações utilizando conjuntos de dados (matrizes e vetores), facilitando sua aplicação para Álgebra Linear e Geometria Analítica.

O principal intuito da linguagem é permitir que o usuário crie suas próprias funções, sendo capaz de utilizá-las no futuro tanto com chamada direta, quanto na definição de novas funções, sendo capaz de facilitar o estudo de disciplinas da área de matemática para alunos de computação, uma vez que serão capazes de tirar a prova do resultado de seus exercícios.

A linguagem também conta com o uso de escopos dinâmicos. Como Falk é um interpretador de cálculos, isso pode facilitar a implementação de funções.

Falk tem uma sintaxe simples e seu uso após definidas as funções pelo usuário permitem um acesso rápido a suas funcionalidades via plataformas móveis. O desenvolvimento de um interpretador para essas plataformas não está previsto neste projeto, mas pode ser visto como um possível trabalho futuro.

Como consequência de sua sintaxe simplificada, Falk também poderá ser utilizada para ensino de computação na escola em nível médio, permitindo que os alunos entrem em contato com uma linguagem mais complexa, mas ainda tendo diversas facilidades e rápidas respostas de suas implementações, por ser uma linguagem interpretada.

Requisitos de compilação e execução:

- clang++ 3.8

```
sudo apt-get install clang++-3.8
ln -s /usr/bin/clang-3.8++ /usr/bin/clang++
```

  É necessário criar um link simbólico para o clang como clang++. Para solucionar isso sem criar esse link é necessário modificar o compilador “CXX” no arquivo “config.mkd”
- Bison 3.0.4
- Flex 2.5.35
- ncurses-dev

```
sudo apt-get install ncurses-dev
```

- gtest

```
sudo apt-get install libgtest-dev
```

Técnica de desenvolvimento:

A linguagem foi separada em módulos funcionais, explicados individualmente a seguir. Alguns módulos são interdependentes e outros são independentes, mas a divisão feita desta forma - ao invés de uma divisão feita em versões incrementais - permitiu que o trabalho fosse mais facilmente dividido e particionado.

Durante o tempo de desenvolvimento foram utilizadas técnicas de programação em par (pair programming) e também de programação individual. Cada etapa do desenvolvimento está aqui documentada e seu crescimento pode ser acompanhado no git, contudo, os commits não estão necessariamente associados aos módulos correspondentes, uma vez que essa organização surgiu de forma natural durante o desenvolvimento.

Especificação dos Módulos

Módulo 0: Sintaxe e Operações Aritméticas

A primeira versão do Falk deve permitir as seguintes características:

I. Interpretação de valores ou operações descritas;
II. Parênteses ();
III. Dois tipos: real (números reais) e complex (números complexos);
IV. Operadores aritméticos binários para escalares: + soma, - subtração, * multiplicação, / divisão, ** potência e % resto da divisão;
V. Operadores unários para escalares: - menos unário e + mais unário;
VI. Comentários de linha e multi-linha, descritos respectivamente por // e /* comments */.

Essa primeira versão da linguagem é apenas um interpretador de operações aritméticas simples, ainda sem permitir o uso de variáveis.
Sempre que uma operação for feita diretamente no terminal, seu resultado será impresso em sequência.
Para separar instruções em Falk, existem duas possibilidades. Pode ser usada a quebra de linha ou um ponto e vírgula (;).

Exemplos de código:

```
falk> 235 * 22
falk> 22i + 38 - 7i + 12
falk> -(7 * -1) % 2
falk> 1/0
falk> 0/0
```

Saída do interpretador:

```
res = 5170
res = 50 + 15i
res = 1
res = inf
res = -nan
```

Erros da versão:

1. Módulo envolvendo um complexo:

```
falk> 48i % 3
```

[Line 0] semantic error: illegal operation: complex modulus

```
falk> 48 % 3i
```

[Line 1] semantic error: illegal operation: complex modulus

Módulo 1: Operações Booleanas

Este módulo adiciona as seguintes características à linguagem:

I. Operadores binários booleanos para escalares: > maior, < menor, == igual, <> ou != diferente, >= maior ou igual e <= menor ou igual;
II. Um novo tipo: bool com valores *true* e *false*;
III. Operadores binários booleanos para booleanos: & AND (e) e | OR (ou);
IV. Operador unário booleano: ! NOT (negação lógica).

Ao comparar os tamanhos de reais e complexos, será levado em conta a ordem lexicográfica, onde são comparadas primeiro as partes reais e depois as partes imaginárias.
Um valor real ou complexo de valor diferente de 0 sempre que coagido a booleano será dado como *true*, caso contrário, *false*. O compilador sempre usará o valor coagido para realizar os operadores binários e unários para booleanos.

Exemplo de código:

```
falk> 2 > 3
falk> 3 > 2
falk> 1 <> 1
falk> 1 == 1
falk> 1 <> 2
falk> true & false
falk> true | false
```

Saída do Compilador:

```
res = false
res = true
res = false
res = true
res = true
res = false
res = true
```

Erros da versão:

1. Comparação entre tipos estruturais diferentes:

```
falk> [1, 2, 3] > [[1, 2],[2, 3]]
```

[Line 0] semantic error: cannot compare array and matrix

```
falk>  2 > [[1]]
```

[Line 1] semantic error: cannot compare scalar and matrix

```
falk> 2 < [1, 4]
```

[Line 2] semantic error: cannot compare scalar and array

Módulo 2: Matrizes e Vetores

Este módulo adiciona as seguintes características à linguagem:

I. Dois tipos de conjuntos de valores literais: array (vetores) e matrix (matrizes);
II. Operadores aritméticos anteriormente definidos adaptados para funcionamento nos novos conjuntos (regras detalhadas abaixo).

Operações de escalares com matrizes e matrizes com matrizes serão definidas conforme as regras matemáticas já conhecidas para essas operações.

Operações de escalares por vetores são definidas como se o escalar operasse cada valor do vetor um a um.

Operações de vetores por vetores são definidas para valores de posição equivalente.

Em operações entre vetores e matrizes, o vetor é coagido a uma matriz-linha.

Exemplo do código:

```
falk> [[1,2,4],[8,16,32]] * [[1,2,3],[4,5,6],[7,8,9]]
falk> 2 * [2,4]
falk> [2,4] * [[1,2,3],[4,5,6]]
```

Saída do interpretador:

```
res = [[37, 44, 51], [296, 352, 408]]
res = [4, 8]
res = [[18, 24, 30]]
```

Erros da versão:

1. Estrutura com três ou mais dimensões:

```
falk> [[[0]]]
```

[Line 0] semantic error: 3d or bigger structures are not supported

2. Estrutura heterogênea:

```
falk> [1,[2]]
```

[Line 0] semantic error: heterogeneous arrays are illegal

3. Operações entre arrays de tamanhos diferentes:

```
falk> [1, 2] + [1]
```

[Line 1] semantic error: array size mismatch (2 and 1)

4. Soma ou subtração entre matrizes com número de linhas ou colunas diferente:

```
falk> [[1],[2]] + [[1]]
```

[Line 2] semantic error: row count mismatch (2 and 1)

```
falk> [[1,2]] + [[1]]
```

[Line 3] semantic error: column count mismatch (2 and 1)

5. Multiplicação de matrizes incompatíveis:

```
falk> [[1,2,3]] * [[1],[2]]
```

[Line 4] semantic error: the number of columns of the first matrix (3) must be equal to the number of rows of the second matrix (2)

6. Divisão, módulo ou exponenciação envolvendo uma matriz no segundo operando:

```
falk> [[1,2],[2,1]] / [[2,3],[1,2]]
falk> [[1]] % [[2]]
falk> [[2,3],[1,2]] ** [[2]]
```

[Line 5] semantic error: illegal operation: matrix division
[Line 6] semantic error: illegal operation: matrix modulus
[Line 7] semantic error: illegal operation: matrix exponentiation

Módulo 3: Variáveis

Este módulo adiciona as seguintes características à linguagem:

I. Declaração de variáveis: var (escalares), array (vetores) e matrix (matrizes);
II. Operador de atribuição: =;
III. Coerção entre qualquer tipo de escalar;
IV. Operadores de atribuição com uso de operadores: +=, -=, *=, /=, **=, %=;
V. Variável res nativa do sistema, que guarda o resultado da última chamada no terminal.

Quando uma operação não tem destino no meio do código como uma linha com um "2+2" solitário, seu resultado será também guardado na variável res.

Exemplo de código:

```
var x : real
x = 2
var y = 3
x + y            // Essa chamada guardará seu resultado em res
var b = x < res
b
```

```
falk> array a = [1, 2, 3]
falk> a = 4
falk> a
```

```
falk> matrix b = [[1, 2]]
falk> b = [7, 8]
```

```
falk> matrix c = [[1, 2], [3, 4]]
falk> c = 7
falk> c
```

Saída do compilador:

```
res = 5
res = true
```

```
res = [4, 4, 4]
```

```
res = [[7, 8]]
```

```
res = [[7, 7], [7, 7]]
```

Erros da versão:

1. Utilizar variáveis indefinidas:

```
falk> a + 2
```

[Line 0] semantic error: undeclared variable a

2. Re-declaração de variável:

```
falk> var b : real
falk> var b = 2
```

[Line 1] semantic error: re-declaration of symbol b

3. Atribuição entre arrays ou matrizes de tamanhos diferentes:

```
falk> matrix x = [[2,3],[2,1]]
falk> x = [[1,2]]
```

[Line 1] semantic error: row count mismatch (2 and 1)

```
falk> matrix x = [[2,3],[2,1]]
falk> x = [[1],[2]]
```

[Line 1] semantic error: column count mismatch (2 and 1)

```
falk> array x = [1, 2, 3]
falk> x = [2, 4]
```

[Line 1] semantic error: array size mismatch (3 and 2)

4. Operador *= para matrizes quando o produto existe mas não é compatível com a matriz-destino:

```
falk> matrix a = [[1,2],[3,4]]
falk> a *= [[1,2,3],[4,5,6]]
```

[Line 1] semantic error: second matrix must be square (found a 2 x 3 matrix instead)

5. Atribuição entre tipos diferentes:

```
falk> var a = [1, 2]
falk> var b = [[2]]
falk> array c = 2
falk> array d = [[2,3]]
falk> matrix e = 2
falk> matrix f = [2,3,4]
```

[Line 0] semantic error: cannot assign array to scalar
[Line 1] semantic error: cannot assign matrix to scalar
[Line 2] semantic error: cannot assign scalar to array
[Line 3] semantic error: cannot assign matrix to array
[Line 4] semantic error: cannot assign scalar to matrix
[Line 5] semantic error: cannot assign array to matrix

Módulo 4: Blocos Condicionais

Blocos if-then-else nessa linguagem funcionam de forma semelhante a linguagens de programação conhecidas como Java e C. Após um if devem ser abertos parênteses e colocada a condição para o caso *true*, em seguida é iniciado um bloco de código *then*. Após o final do bloco *then*, um bloco *else* pode ser declarado, usando a palavra else seguida da abertura de um novo bloco. Um novo bloco condicional (ou um escopo) é definido a partir de um símbolo de : (dois pontos) e é finalizado por um . (ponto final).

Com a adição de blocos condicionais, aparece a necessidade de múltiplos escopos na linguagem. Em Falk escopos são dinâmicos, ou seja, quando uma variável tem seu valor modificado ou é redefinida neste escopo, ela tem seu valor atualizado e, até que o fluxo de execução saia deste escopo, o valor atualizado é o que será utilizado.

Exemplo de código:

```
var a = 42;
```

```
if (a > 3):
    2 + 2
.
```

```
if (a < 30):
    2 + 2
. else:
    3 + 3
.
```

```
var a = 42
if (a > 10):
    var a = 50
    a
.
a
```

Saída do interpretador:

```
res = 4
```

```
res = 6
```

```
res = 50
res = 42
```

Erros da versão:

1. Expressão não-escalar no condicional:

```
falk> if ([1,2]) : .
```

[Line 1] semantic error: non-boolean condition

Módulo 5: Laços

Dois tipos de laços são permitidos em Falk:

O primeiro é o for-each, onde se itera sobre cada elemento de um conjunto. Ele é escrito com a palavra for, seguida de um parêntese, um identificador para o item da iteração, a palavra in, o conjunto em questão e fecha parênteses, seguido da abertura do bloco de laço. Esta modalidade de laço não pega os valores por referência, mas sim por cópia, ou seja, não é possível usá-lo para alterar os valores de um conjunto. Seu comportamento para matrizes pega suas linhas uma a uma, no formato de vetores.

O segundo tipo é o while, que irá executar enquanto uma condição for verdadeira. Ele é escrito como a palavra while, seguida de um parêntese, uma condição e fecha parênteses, seguido da abertura de um bloco de laço.

Exemplo de código:

```
falk> array a = [1, 1, 1, 1, 1]
falk> var r : real
falk> for (e in a) : r += e .
falk> r
falk> var t = 1
falk> while (r > 1) :
r -= 1
t *= r
.
falk> t
```

Saída do compilador:

```
res = 5
res = 24
```

Erros da Versão:

1. Expressão não escalar na condição do while:

```
falk> while ([2]) :
2+2
.
```

[Line 2] semantic error: non-boolean condition

2. Escalar como segundo parâmetro do for

```
falk> var b = 2
falk> for (e in b) :
e + 2
.
```

[Line 3] semantic error: for expects an array or matrix, scalar given

3. Identificador de iteração do for já declarado

```
falk> var a = 2
falk> array b = [1,1]
falk> for (a in b) : a + 2 .
```

[Line 2] semantic error: variable a already declared

Módulo 6: Funções

Este módulo da linguagem traz a possibilidade de definir e usar funções previamente definidas na linguagem.

Uma função é definida da seguinte forma:

```
function a(var x, array y, matrix z) :
    code
    return w
.
```

Os argumentos entre parênteses devem ter seu tipo estrutural explicitados. Uma função pode ou não ter um return (retorno), caso não tenha, seu retorno padrão é 0 (zero).

As chamadas de função são mais simples, podendo se aproveitar dos retornos das funções tendo situações como:
`var f = b();`
Ou podem ser usadas para obter resultados diretamente do interpretador, como:

```
sum(y) // retornando o somatório do array y
```

Funções também podem ser redefinidas. Para isso é preciso primeiro livrar-se da última versão delas. Isso é feito através do comando undef. A principal utilidade disso é poder redefinir funções enquanto se tem o ambiente do interpretador aberto.

Exemplo de código:

```
var x = 1

function test(var a) :
    var b = x + a
    return b
.

function sec(var a) :
    x
    x = a;
    x = test(x)
    return x
.

sec(2)
```

```
function hello(var a) : return a .
hello(2)
```

```
undef hello
function hello() : return -1 .
hello()
```

Saída do compilador:

```
res = 1
res = 4
```

```
res = 2
```

```
res = -1
```

Erros da versão:

1. Múltiplas definições de uma mesma função:

```
falk> function f(): return 2 .
falk> function f(): return -1 .
```

[Line 1] semantic error: re-declaration of symbol f

2. Utilizar uma variável como função ou vice-versa:

```
falk> function f() : return -1 .
falk> var g = 42;
falk> f
falk> g()
```

[Line 2] semantic error: f is not a variable
[Line 3] semantic error: g is not a function

3. Utilizar uma função indefinida:

```
falk> j()
```

[Line 0] semantic error: undeclared function j

4. Array de tamanho diferente do especificado:

```
falk> function f(array[1] x): x.
falk> f([1,2])
```

[Line 1] semantic error: mismatching array size for parameter x in function f (expected 1, got 2)

5. Matriz de tamanho diferente do especificado:

```
falk> function f(matrix[2,3] x): x.
falk> f([[1,2]])
```

[Line 1] semantic error: mismatching matrix size for parameter x in function f (expected 2 x 3, got 1 x 2)

6. Quantidade incorreta de parâmetros na chamada de função:

```
falk> function f(var x): x.
falk> f(1, 2)
```

[Line 1] semantic error: mismatching parameter count for function f (expected 1, got 2)

7. Return fora de função:

```
falk> return 2
```

[Line 0] semantic error: return outside of function scope

8. Tentar remover a definição de uma função nunca definida:

```
undef f
```

[Line 0] semantic error: undeclared function f

9. Tentar remover a definição de uma variável:

```
var g = 42
undef g
```

[Line 1] semantic error: g is not a function

Módulo 7: Inclusão de bibliotecas e funções pré-definidas

Para facilitar o uso de funções previamente definidas em arquivos em separado, é possível importá-las em Falk. Isso irá garantir que elas estarão carregadas no ambiente e serão de livre uso do usuário enquanto estiverem. No entanto, o usuário não poderá importar funções de nome conflitantes com funções já definidas no sistema.

Exemplo de código:

Erros da versão:

1. Função já definida no ambiente
2. Múltiplas inclusões da mesma biblioteca

Este módulo foi planejado, mas nunca implementado. Ele será aqui considerado um trabalho futuro, umas vez que pode ser feito, mas por questões de tempo, não foi realizado.

Módulo 8: Dedução de Tipos

Há duas formas de se fazer a dedução de tipo em Falk.

O primeiro é relacionado ao tipo de conjunto: uma variável pode ser definida como var, array ou matrix, mas a funcionalidade auto acaba com a necessidade dessa determinação. Uma variável declarada como auto terá seu tipo estrutural definido automaticamente.

O segundo tipo de dedução está relacionado ao tipo das variáveis. Nota-se que esta dedução só pode ser feita com escalares (e não com conjuntos). O typeof pode substituir a determinação do tipo na declaração de uma variável, ou seja, ao invés de se fazer:

```
falk> array a = [1,2,3]
falk> var x : real
falk> x += a[2]
```

Pode-se fazer:

```
falk> array a = [1,2,3]
falk> var x : typeof a[0]
falk> x += a[2]
```

Exemplo de código:

```
falk> function sum(array x):
    var c : typeof x[0];
    for (e in x):
       c += e;
    .
    return c
.

falk> sum([1, 2, 3, 4])
```

```
falk> auto a = 42
falk> auto b = [42]
falk> auto c = [[42]]
falk> a
falk> b
falk> c
```

Saída do compilador:

```
res = 10
```

```
res = 42
res = [42]
res = [[42]]
```

Erros da versão:

1. Tentativa de dedução de tipo de container

```
falk> array a = [1, 2]
falk> var x : typeof a
```

[Line 1] semantic error: typeof expects a scalar

Módulo 9: Vetores e Matrizes como variáveis

Este módulo finaliza o relacionamento dos containers com variáveis, permitindo que eles tenham todas as características de uma variável escalar, podendo ser guardados e operados com facilidade.

I. Acesso a elementos de vetores e matrizes através de [] (colchetes);
II. Coerção de tipos ao gerar um container, para o tipo mais alto encontrado (complex > real > bool);
III. Instanciação de vetores e matrizes por seus tamanhos, inicializando seus valores com o padrão do tipo especificado.

Exemplo de código:

```
falk> array a = [1, 2, 3]
falk> a[1] = 4
falk> a
```

```
falk> matrix b = [[1, 0], [0, 1]]
falk> b[1] = [4, 5]
falk> b
falk> b[,1] = [7, 8]
falk> b
falk> b[1,1] = 3
falk> b
```

Saída do compilador:

```
res = [1, 4, 3]
```

```
res = [[1, 0], [4, 5]]
res = [[1, 7], [4, 8]]
res = [[1, 7], [4, 3]]
```

Erros da versão:

1. Inicialização de array ou matriz com tamanho não-escalar:

```
falk> array[[1]] a : real
```

[Line 0] semantic error: the size must be a scalar

2. Tentativa de acesso a posições inexistentes do container;

```
falk> matrix a = [[1, 2], [3, 4]]
falk> a[3]
```

[Line 1] semantic error: index out of bounds (limit = 2, actual = 3)

Arquitetura do Compilador

A arquitetura se baseia na construção de uma *abstract syntax tree* (AST), a qual é analisada por uma classe interpretadora (*evaluator*) que visita cada um dos nós dessa árvore e executa as ações semânticas correspondentes.

Nós da AST:

A classe virtual *node* é um container genérico para a AST, capaz de guardar qualquer tipo de informação na árvore. *Nodes* contém um método *visit*, que faz com que o analisador visite este nó e processe seus dados e os dados de seus filhos, caso existam. Esse processo ocorre através de chamadas de métodos *analyse* que recebem o dado do nodo e seus filhos.

As classes *list*, *lvalue* e *rvalue* provém abstrações sobre a montagem da árvore sintática. Elas instanciam os nós internamente e garantem operadores para construção da árvore, por exemplo: caso haja dois *rvalues* e na gramática eles sejam somados, o operador de soma criará um novo nó de soma, adicionando os nós já existentes dentro dos *rvalues* como filhos.

Interpretador:

O interpretador processa comando a comando, chamando o método visit de cada nó, o que desencadeia uma sequência de *analyse*'s. Cada *analyse* toma as ações semânticas necessárias para execução do código, dependendo da informação que ele recebeu. Por exemplo, ao receber a operação de soma, o interpretador visita seus nodos filhos, guardando em pilhas os valores finais que encontra (escalar, matriz ou vetor). Ao voltar ao método principal (o que recebeu a soma), ele desempilha os valores correspondentes aos filhos e faz a soma dos mesmos. O resultado é empilhado, de forma que somas aninhadas funcionem corretamente. Essa lógica se aplica a todas as operações.

Tipos:

Há uma divisão nas variedades de tipos em Falk. Os tipos fundamentais, tipos estruturais e os tipos simbólicos.

Os tipos fundamentais da linguagem são *real*, *complex* e *bool*. Sendo o primeiro representando os números reais, o segundo os números complexos e o terceiro os booleanos. Em casos em que é necessário fazer coerção para um tipo, eles devem seguir uma ordem de maior representatividade, portanto se um *bool* e um *real* estiverem sendo coagidos com um *complex*, todos se tornarão *complex*, seguindo sempre esta ordem: *complex* > *real* > *bool*.

Os tipos estruturais são *scalar*, *array* e *matrix*. Representando escalares, vetores e matrizes, respectivamente. Estruturas de dimensões maiores não são suportadas.

Há coerção de *scalar* para *array* e de *array* para *matrix*, como é visto na seção Módulo 3, exemplos 2, 3 e 4, da Especificação dos Módulos.

Os tipos simbólicos são as variáveis e funções. Sendo o primeiro responsável por guardar valores e o segundo comportamentos. São eles que são guardados na tabela de símbolos, guardando os identificadores referentes às variáveis e funções do programa.

Tabela de símbolos e escopos dinâmicos:

Representada pela classe *symbol_mapper*, a tabela de símbolos possui uma pilha de escopos, cada qual é composto por uma tabela de símbolos geral, uma tabela de variáveis e uma de funções, cada tabela é indexada por uma string (representativa do elemento em questão).

A pilha de escopos é o que garante que Falk tenha escopos dinâmicos, dado que quando uma variável é acessada, ela é antes buscada no escopo mais recente, depois nos anteriores, até ser encontrada.

A linguagem só permite que seja declarada uma variável ou função caso esta não exista na tabela de símbolos geral do escopo atual.

Testes

Os testes foram desenvolvidos usando *gtest*, foram realizados testes de sistema para todas as funcionalidades especificadas, checando todos os erros esperados. Para executá-los é necessário executar:

```
make
make tests && ./bin/test
```

O arquivo principal de erros se encontra em tests/test.cpp. Na pasta tests/cases/ encontram-se alguns testes mais completos que também são executados no arquivo principal.