Flux Control Language

Aluno: Aleksander Luiz Lada Arruda
Orientador: Sérgio Vale Aguiar Campos

Departamento de Ciência da Computação
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte, MG - Brazil
{aleksander, scampos}@dcc.ufmg.br

Abstract. *This paper describes a programming languaged designed for controlling a robotic platform for manipulating biochemical samples in laboratories and the interpreter designed specifically for the language. It contains all of the language's documentation, as well as its lexical and syntactic definitions. It also contains a documentation for the structure of the interpreter and a list of instructions explained. The final result of this work is a formal description of the FCL language and its interpreter, which objective is to provide support for procedural programming in the equipment.*

Resumo. *Este paper descreve uma linguagem de programação projetada para controlar uma plataforma robótica de manipulação de amostras bioquímicas de laboratórios e o interpretador projetado especificamente para a linguagem. Nele está contido toda a documentação da linguagem, bem como as definições léxicas e sintáticas. O documento também contém uma documentação para a estrutura do interpretador e uma lista de instruções explicadas. O resultado final deste trabalho é uma descrição formal da linguagem FCL e seu interpretador, que tem como objetivo prover suporte à programação procedimental ao equipamento.*

1. Introdução

FCL (*Flux Control Language*) é uma linguagem de programação desenvolvida para controlar o equipamento ELISA - uma plataforma róbotica para manipulação de amostras laboratoriais arquitetada pela Celer Biotecnologia. A linguagem foi projetada para dar à plataforma suporte à programação procedimental, eliminando a necessidade de programar o equipamento no nível de instruções.

A FCL permite descrever os experimentos na plataforma de maneira dinâmica, e otimiza a forma como tais são executados seguindo determinadas restrições. Além de uma linguagem, a FCL é uma ferramenta capaz de abstrair o todo processo de criação de experimentos, removendo a necessidade de se avaliar as melhores formas de realizar tais experimentos.

2. Definição Formal da Linguagem

A FCL é uma linguagem script, cujo interpretador tem a responsabilidade de controlar o fluxo de instruções direcionadas ao equipamento ELISA. Não apenas isso, a FCL é também uma linguagem genérica que permite a criação de programas tradicionais desenvolvidos em qualquer outra linguagem.

Baseada em linguagens de programação como Lua e Python, conhecidas por uma sintaxe compacta e eficiente, a FCL procura satisfazer todos os casos de uso da plataforma através de abstrações que buscam facilitar a descrição de tarefas repetitivas a serem realizadas pelo equipamento. Todas as funcionalidades da linguagem são descritas nos tópicos que se seguem, bem como suas convenções léxicas e sintáticas.

2.1 Convenções Léxicas

Identificadores em FCL são usados para nomear variáveis e procedimentos. Estes podem ser quaisquer strings contendo letras, dígitos ou underscores, desde que não tenham um dígito como o primeiro caractere. Palavras-chave da linguagem também não podem ser utilizadas como identificadores. FCL é *case-sensitive*, o que significa que nomes são diferenciados por letras maiúsculas ou minúsculas. Palavras-chave são sempre minúsculas.

São reservadas as seguintes palavras-chave para o controle da linguagem:

```
uses        for         while       repeat
if          then        else        elseif
do          end         run         discard
print       sleep       move        function
true        false       as          procedure
none        foreach     pipettes    return
pickup      step
```

As strings a seguir denotam outros tokens também pertencentes à linguagem:

```
+     -     *     /     %     ..    =     ,
==    !=    <     >     <=    >=    !     .
(     )     [     ]     “     :     ;     ~
```

Literais podem ser strings escapadas por aspas duplas ou números. Strings podem conter caracteres de formatação ASCII, e números são limitados à inteiros e decimais. Comentários de linha podem ser feitos por hífens duplos (--). Linhas de código em FCL não precisam ser terminadas em ponto e vírgula, mas a fim de manter a afinidade com outras linguagens de programação este tipo de terminação é aceita.

2.2 Tipos e Valores

FCL é uma linguagem de tipagem dinâmica, com conversões implícitas. À qualquer variável pode ser atribuído qualquer valor, a qualquer momento, desde que esta tenha sido declarada. Variáveis podem ser passadas como parâmetros de procedimentos, com passagens unicamente por valor. Isto significa que quaisquer variáveis passadas como parâmetro não podem ser modificadas dentro do escopo do procedimento. FCL possui seis tipos de dados: *number, boolean, string, position, object* e *list*. Qualquer variável declarada sem valor atribuído é inicializada com o valor especial *none*.

O tipo *number* pode receber números inteiros ou decimais de 32-bits, negativos ou positivos. O tipo *boolean* é representado como um *number* e pode receber os valores *true* ou *false*, os quais são representados respectivamente como *1* e *0*.

Strings são cadeias de caracteres delimitadas por aspas duplas, que podem ou não conter caracteres ASCII. Neste caso, caracteres especiais devem ser escapados com "\". FCL realiza a coerção de todos os tipos para *strings* diante do operador de concatenação "..".

O tipo *position* pode receber valores de posições ou de *range* de posições, relativas às coordenadas de cada *placa de 96 poços* ou *rack* presentes no equipamento. O valor deste tipo sempre é atribuído entre colchetes, definindo uma posição única ou um *range*, variando entre *a1* até *h12*, sendo o range definido pelo operador "~".

```
var range = [a1 ~ a12]
```

O tipo *object* representa objetos implícitos presentes na linguagem, os quais são interfaces para os objetos reais existentes na plataforma. Variáveis do tipo *object* não podem receber novos valores, mas apenas referenciar tais objetos implícitos. Os objetos implícitos são: *Sample, Plate* e *Rack*, definitos no interpretador por diferentes índices para cada objeto presente na plataforma. Objetos implícitos são globais (*i.e. podem ser acessados em qualquer escopo*), mas a utilização destes fora do procedimento *main* é desencorajada por razões explicadas no tópico *2.4.4*.

Por fim, o tipo *list* é um tipo que pode conter valores de todos os tipos em uma lista. Uma lista é representada por chaves fechadas, e pode conter quaisquer valores dentro da mesma separados por vírgula. Listas são sempre ordenadas por inserção, e seus membros indexados de um até o tamanho da lista.

2.3 Variáveis

Variáveis em FCL são utilizadas para armazenar e referenciar valores. Em FCL variáveis podem ser globais ou locais, sendo as primeiras declaradas no escopo global. Com exceção de variáveis do tipo *object*, todas as variáveis são restritas a armazenar

valores. No caso de variáveis do tipo *object*, estas são usadas para referenciar objetos implícitos na linguagem.

Uma variável pode ser declarada ao se atribuir um valor a um identificador qualquer, ou opcionalmente utilizando-se, opcionalmente, a palavra-chave *var*:

```
vardecl ::=
      [var] Identifier |
      [var] attr
```

Atribuições são integralmente descritas no tópico 2.4.1.

2.4 Declarações

FCL define um conjunto de declarações similares à outras linguagens de programação. Estas declarações podem ser expressões lógicas, atribuições, estruturas de controle, chamadas de funções e declaração de variáveis.

2.4.1 Atribuição de Valores

FCL permite atribuir valor a apenas uma variável por vez, não sendo permitidas cadeias de atribuições. A atribuição de valores de FCL é definida por:

```
attr ::=
Identifier `=´ Value |
Identifier `=´ procedurecall
```

É possível realizar atribuições dentro de declarações *for*. Também é possível atribuir valores de retorno de procedimentos. *Values* são descritos pela sintaxe formal da linguagem, no tópico *6*.

2.4.2 Blocos

Blocos são unidades mínimas executáveis da FCL. Estes são constituídos por declarações sequenciais não necessariamente terminadas em ponto e vírgula. Na prática, apesar de ser considerado um operador da linguagem, o ponto e vírgula é descartado pelo compilador. Blocos são definidos por:

```
block ::= statement
statement ::=
      while exp do `:´ block end |
      if exp then block {elseif exp then block} [else block] end
[...]
```

Todas as declarações são descritas pela sintaxe formal da linguagem no tópico 6.

2.4.3 Estruturas de Controle

As estruturas de controle presentes em FCL foram projetadas de maneira a simplificar a descrição de ações realizadas pelo equipamento. O laço *for* foi particularmente feito para iterar sobre *ranges* do tipo *position*.

```
statement ::=
      while exp do `:´ block end |
      for Identifier `=´ Range [step Value] do `:´ block end|
      foreach Identifier as Identifier do `:´ block end |
      if exp then `:´ block {elseif exp then `:´ block} [else
      `:´ block] end
```

As estruturas de controle da FCL são muito semelhantes àquelas presentes em outras linguagens de programação, exceto que esta usa uma sintaxe mais verbosa. Isto se deve ao fato que, por uma decisão de projeto, foi idealizado que a linguagem deve ser amigável à usuários que desconhecem linguagens de programação. Assim sendo, a linguagem foi projetada de forma que esta possa ser lida como uma descrição de procedimentos, ainda que esta ofereça abstrações mais complexas ao desenvolvedor.

FCL oferece as estruturas de controle *if/elseif/else, while, for, foreach* e *procedures*. As estruturas condicionais e o laço *while* já são bem conhecidas de outras linguagens de programação, e não variam em aplicação. O laço *for,* por outro lado, tem um funcionamento peculiar, iterando apenas sobre variáveis do tipo *position*, permitindo a iteração sobre linhas e colunas inteiras dos objetos implícitos, utilizando ou não passos (*steps*). O laço *foreach* tem a funcionalidade de iterar sobre variáveis do tipo *list*, percorrendo todos os valores presentes na mesma, em ordem.

```
if x < 10 then: [...]
elseif x == 10 then: [...]
else: [...] end

while x == 10 do: [...] end

for range = [a1 ~ a12] step 2 do: [...] end

foreach list_of_ranges as range do: [...] end
```

Procedures são descritos no tópico seguinte.

2.4.4 Procedimentos

Procedimentos são a essência da FCL, tendo sido projetados para descrever as ações que serão realizadas sobre os objetos da plataforma em um escopo fechado do código.

Procedimentos permitem a padronização de experimentos, a reutilização de código e podem ser aplicados sobre quaisquer objetos implícitos. Desta forma, ao definir um procedimento, também hão de ser definidos os objetos sobre o qual o experimento irá atuar. Estes objetos, chamados de *objetos de interação,* são passados por parâmetros especiais aos procedimentos, de maneira diferente aos parâmetros normais, e são referências aos objetos implícitos. A sintaxe dos *procedures* é descrita por:

```
objlist ::= Identifier as Identifier { `,´ Identifier as Identifier }
procedure ::= procedure Identifier `(´ [ varlist ] `)´ { uses objlist } `:´ block end
```

Procedimentos recebem os parâmetros simples por passagens de valores, enquanto os parâmetros de *objetos de interação* são passagens por referência. Isto se deve ao fato que os objetos implícitos são globais e imutáveis, e portanto não é possível duplicar o valor dos mesmos. Desta forma, os valores recebidos pelo procedimento são referências aos objetos globais, a fim de permitir que um mesmo experimento seja executado sobre diferentes objetos. Por esta razão, é recomendado que apenas a função *main* seja responsável por gerenciar os objetos globais, ainda que outros procedimentos também possuam acesso aos mesmos. Segue um exemplo de um procedimento, com um *objeto de interação* p1:

```
procedure fill(line) uses Plate as p, Sample as s:
end
```

2.4.5 Chamada de Procedimentos

A chamada de procedimentos é feita dentro de qualquer procedimento, sendo o procedimento *main* considerado o procedimento de entrada pelo interpretador. Um procedimento é chamado através da palavra-chave *run*. Ao chamar um procedimento, é possível passar parâmetros simples entre parênteses e *objetos de interação* através da palavra-chave *with*, que por sua vez irá executar seu bloco de declarações e retornar ou não um valor. Procedimentos só podem retornar valores e nunca referências, tão logo *objetos de interação* não podem ser retornados por um procedimento.

A sintaxe para invocação de procedimentos é definida por:

```
procedurecall ::= run Identifier `(´ [ varlist ] `)´ [ with objlist ]
```

A lista de parâmetros ou de *objetos de interação* são omitidas para procedimentos que não recebem parâmetros. A execução do procedimento de exemplo do tópico anterior se dá por:

```
run fill ([A]) with Plate[1], Sample[1]
```

2.5 Expressões

Expressões em FCL são um conjuntos de operações que retornam valores. Estas operações podem ser relacionais, aritméticas ou chamadas a procedimentos. Abaixo seguem as expressões em FCL:

```
exp ::=
`(´ exp `)´ |
none | true | false |
Number |
String |
Position |
var |
procedurecall |
exp binaryop exp |
unaryop exp
```

2.5.1 Operadores Aritméticos

Os operadores artiméticos em FCL são:

```
+     -     /     *     %
```

Note que todos os operadores são binários.

2.5.2 Operadores Relacionais

Os operadores relacionais em FCL são:

```
==    !=    <     >     <=    >=    !
```

Note que o operador de negação "*!*" é o único operador unário.

2.5.3 Outros Operadores

O único operador em FCL além dos já descritos é o operador binário de concatenação e o operador de range:

```
..    ~
```

2.5.4 Precedência

A precedência dos operadores de FCL segue abaixo, da maior para a menor prioridade:

```
!
*       /       %
+       -
..
==      !=      <       >       <=      >=
```

O operador de negação possui a maior precedência, seguido dos operadores aritméticos de multiplicação, divisão e resto. Operadores de menor precedência aritmética vêm a seguir, para por fim os operadores relacionais atuarem. A precedência do operador de concatenação se deve à coerção de valores da FCL.

2.5 Regras de Visiblidade

FCL possui escopo estático. O escopo de uma variável começa ao ser declarada, e termina ao fim do bloco em que esta for declarada. Note, no entanto, que o escopo de uma variável não engloba chamadas de procedimentos. Isto significa que sempre que um procedimento é chamado, um novo escopo é criado em que apenas as variáveis globais estão presentes.

2.6 Coletor de Lixo

FCL possui gerenciamento automático de memória - ou coletor de lixo. Sempre que uma variável não for mais acessível, isto é, sempre que seu escopo for encerrado, esta será deletada pelo interpretador. O coletor de lixo não executa a cada ciclo do interpretador, mas em intervalos pré-definidos.

2.7. Casos de Uso

FCL é usada para controlar o fluxo de instruções enviadas ao equipamento. Para tal, são embutidos comandos na linguagem que efetivamente realizam as operações na plataforma. Abaixo encontra-se uma lista de comandos presentes na linguagem:

Nome	Descrição

MOVE	Move amostras entre objetos e coordenadas. Note que o equipamento consegue preencher 4 poços por vez, cada um separado por 2 poços. As instruções de movimentação são traduzidas na maneira mais rápida de realizar a tarefa.
	
	Uso:
	
	`move from Sample to Plate[a1 : a12]`
	`move from Plate[1][a1 : 12] to Plate[2][a1 : a12]`

	Para as instruções acima, o melhor caso de execução é em 3 movimentos, assumindo que todas as pontas do equipamento tenham todas as pipetas. Caso não haja nenhuma pipeta nas pontas, um erro é gerado.
PICKUP	Pega pipetas no rack especificado. As pipetas sempre serão colocadas nas pontas mais à direita. Uso: `pickup 3 pipetts from Rack[1]` O exemplo acima irá colocar 3 pipetas nas pontas mais à direita.
DISCARD	Descarta as pipetas nas pontas do equipamento.

Os comandos podem ser inseridos dentro de qualquer bloco, e serão extraídos pelo compilador para se tornarem comandos diretos ao interpretador. Nos próximos tópicos são expostos exemplos de uso da linguagem FCL. Note que os comandos de *PICKUP* e *DISCARD* são omitidos, a fim de reduzir o código.

2.7.1 Preenchimento de Placa

O código a seguir preenche uma placa com amostras da bandeja.

```
-- Note the definition of the interaction objects
procedure fill() uses Sample as s, Plate as p:
      move from s to p[a1 ~ h12]
end

procedure main():
      run fill() with Sample[1], Plate[1] --Implicit objects
end
```

2.7.2 Transferência

O código a seguir transfere os poços da coluna a e b da placa 1 para as colunas g e h da placa 2.

```
procedure transfer(p1_r, p2_r) uses Plate as p1, Plate as p2:
      -- Only works if p1_r has the same size as p2_r
      move from p1[p1_r] to p2[p2_r]
end
```

```
procedure main():
      var p1_range = [a1 ~ b12] -- Can also be [a : b]
      var p2_range = [g1 ~ h12] -- Can also be [g : h]
      run transfer(p1_range, p2_range) with Plate[1], Plate[2]
end
```

2.7.3 Diluição Linha Sim, Linha Não

O procedimento a seguir realiza a diluição de amostras na primeira linha [a] utilizando as linhas seguintes em passos de um, isto é, pulando uma linha por laço. Neste caso são usadas apenas as linhas [c, e, g] para diluição.

```
procedure dilute() uses Plate as p:
      for l = [a ~ h] step 1 do:
            for c = [1 ~ 12] do
                  move from p[l][c] to p[l][c+1]
            end
      end
end
```

2.7.4 Descarte de Amostras em Uma Linha

```
procedure discard() uses Plate as p1:
      move from p1 to discard
end
```

2.7.5 Omissão de Parâmetros de Posição

A omissão de parâmetros de posição é permitida. Caso seja omitida a linha a coluna inteira será considerada, e vice-versa. No caso da omissão completa de parâmetros de posição, o objeto inteiro é considerado. Neste caso, comandos como:

```
move from p1[a ~ d] to p2[a ~ d]
move from p1 to p2
```

São considerado válidos.

3. Interpretador

O interpretador FCL foi projetado para funcionar como uma aplicação completa de controle, que não só é responsável pela compilação de um script FCL para um código portável (ou p-code) mas também é responsável por executar o código na plataforma e por garantir a segurança e o controle de acesso à mesma. O interpretador FCL foi projetado para ser um serviço, e não apenas um interpretador avulso. Desta forma, o interpretador é ainda responsável por garantir que erros sejam detectados antes da execução através de mecanismos de simulação que são descritos nos próximos tópicos. O p-code para a qual a FCL é compilada foi também projetado para atender às necessidades do sistema, e é explicado nos tópicos que se seguem.

3.1 Princípios Básicos

O interpretador FCL é baseado em uma máquina de pilha. Este design foi adotado pois tende a manter o conjunto de instruções enxuto e de fácil compreensão, além de ter uma implementação mais simples. Máquinas virtuais destinadas à execução de linguagens de programação são regularmente baseadas em pilhas, como no caso da JVM do Java, do interpretador Python e do framework .NET da Microsoft em C#; Isso também significa que a literatura sobre o assunto é extensa, o que corroborou a escolha.

As instruções FCL não possuem representação binária (i.e. byte code) e devem ser escritas diretamente em sua forma textual. Uma vez que o interpretador não está limitado por pipelines de hardware (i.e. o hardware é muito mais rápido que a plataforma), a conversão para o byte code só serviria ao propósito de reduzir o volume do código final. Tendo em vista que o código FCL é destinado à manipulação de um equipamento robótico, longos códigos não serão comuns devido às limitações físicas da plataforma. Algumas considerações sobre o desempenho também influenciaram nesta decisão, que são apresentadas abaixo.

3.2 Interpretador Como um Serviço (SaaS)

O interpretador FCL, tendo sido projetado como um serviço (de maneira análoga à JVM), não tem a função de executar códigos sem estar conectado à plataforma. O interpretador é responsável por compilar (utilizando o compilador avulso da FCL), simular e executar o código fornecido através do serviço local ou web. O interpretador provê um serviço completo de controle da plataforma, incluindo o controle de acesso e uma fila de execução de experimentos pausável. O diagrama de sequência abaixo demonstra o fluxo de execução de um experimento no interpretador FCL, desconsiderando a autenticação:

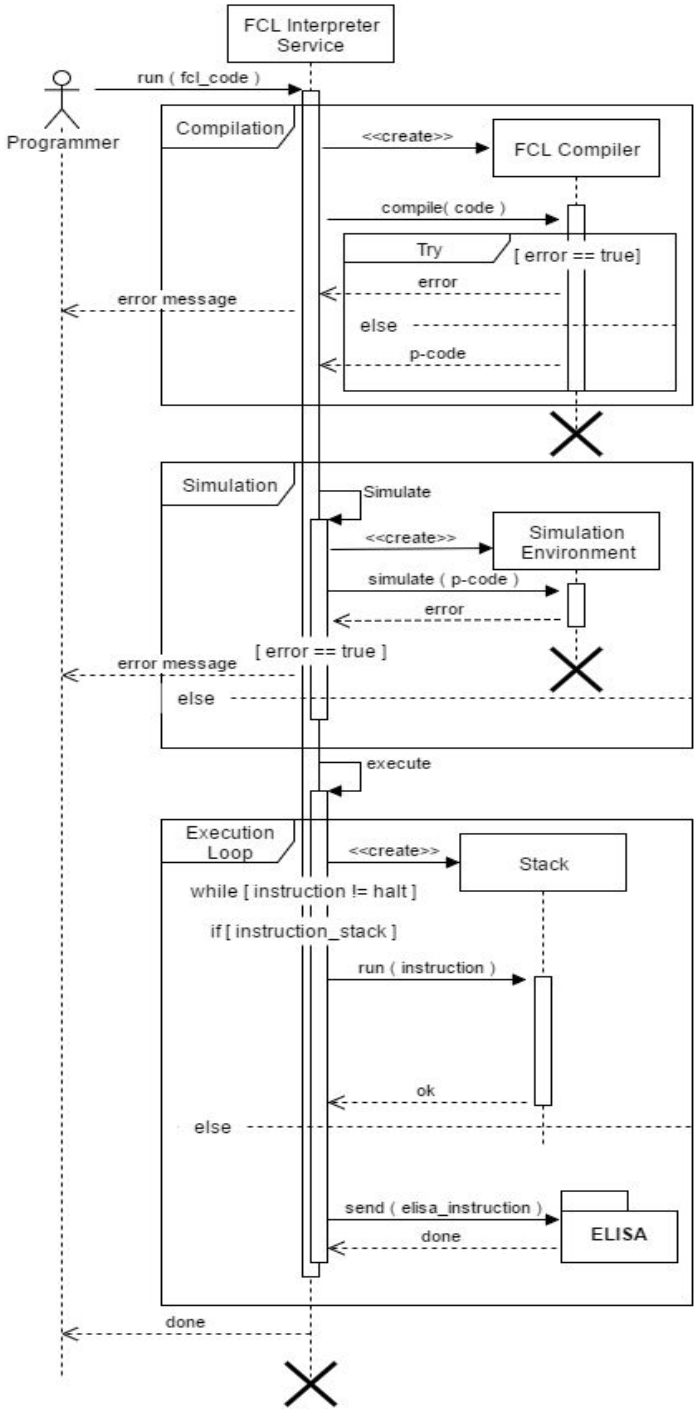

Imagem 1 - Fluxo do Interpretador FCL

Note que, na ocorrência de erros, o experimento é interrompido e o erro é retornado. Caso nenhum erro ocorra, a execução do experimento se torna transparente ao usuário, que deverá notar apenas a movimentação do robô.

3.3 Detecção de Erros Lógicos, Semânticos e Desempenho

O processo de geração do p-code da FCL não inclui uma análise semântica, e tão logo erros semânticos não são acusados durante a compilação. Isto se deve ao fato de que detectar erros semânticos é mais fácil durante a execução do código, em comparação à implementação de uma análise semântica completa.

Tradicionalmente erros semânticos são verificados em tempo de compilação a fim de evitar o caro processo de recompilação do código, além de evitar erros que possam de alguma forma comprometer a estrutura do hardware. No caso da FCL, não só a semântica do código deve ser testada, mas também a lógica, a fim de evitar a execução desnecessária de experimentos com custos financeiros reais. Para tal, antes da execução efetiva do código, uma simulação é executada para verificar a possível ocorrência de falhas ao longo do processo. Desta forma, tendo em vista que a lógica do código FCL já é testada através de uma simulação, esta mesma simulação fica encarregada de realizar a análise semântica. Uma vez que tanto a compilação quanto a simulação são executadas pelo serviço da FCL, o fato da análise semântica se passar na etapa de simulação se torna invisível ao usuário.

Note que os custos gerados pela simulação ao desempenho do interpretador são irrelevantes; Uma única instrução enviada à plataforma em uma execução real pode levar vários segundos para executar, enquanto uma simulação pode executar dezenas de milhares de instruções por segundo. Desta forma, um código que levaria algumas horas em uma execução real pode ser simulado em menos de um segundo, e por conseguinte esta simulação não causa impacto relevante ao desempenho do sistema.

3.3 Conjunto de Instruções

A tabela abaixo dispõe todas as instruções implementadas pelo interpretador FCL. Tais instruções são suficientes para executar todas as estruturas de controle previstas no projeto da lingugem. As instruções, em grande parte, não recebem parâmetros, por se tratarem apenas de operações sobre o topo da pilha.

Instrução	Descrição	Exemplo
`Instruções de Memória`		
`LOAD <var>`	Carrega o valor de uma variável à pilha. Apenas variáveis globais ou no escopo da chamada são acessíveis.	`LOAD counter`
`STORE <var>`	Salva o valor do topo da pilha à variável determinada. Esta operação resulta na remoção (pop) do valor no topo da pilha. Variáveis não existentes serão criadas no escopo local da chamada. Caso nenhum escopo esteja definido, variáveis não existentes serão criadas no escopo	`STORE result`

	global.	
PUSH <val>	Carrega um valor à pilha.	PUSH 10 PUSH [A1 ~ B1] PUSH “str” PUSH nil PUSH Plate[1]
POP	Remove o valor presente no topo da pilha.	POP
Instruções Aritméticas		
ADD	Soma os dois valores no topo da pilha.	ADD
SUB	Subtrai os dois valores no topo da pilha. O valor no topo da pilha é subtraído do valor abaixo dele.	SUB
MUL	Multiplica os dois valores no topo da pilha.	MUL
DIV	Divide os dois valores no topo da pilha. O valor no topo da pilha é o divisor.	DIV
MOD	Calcula o resto da divisão entre os dois números no topo da pilha. O valor no topo da pilha é o divisor.	MOD
POW	Calcula a potência entre dois números. O valor no topo da pilha é o expoente.	POW
Instruções de Concatenação		
CONCAT	Concatena os dois valores no topo da pilha. O valor no topo da pilha é concatenado ao final do valor abaixo dele.	CONCAT
Instruções Lógicas		
NOT	Aplica o operador de negação ao valor no topo da pilha. O resultado sempre será true ou false, a depender do tipo do dado no topo da pilha. Uma string ou list vazias, um number igual a zero ou uma variavel igual a none resultam em false. Todos os outros resultam em true.	NOT

`COMPARE <op>`	Compara o valor no topo da pilha com o valor abaixo utilizando o operador indicado. O valor no topo da pilha está no lado direito da comparação. O valor retornado será será true ou false se a comparação for possível. Caso contrário, um erro é disparado.	`COMPARE <`
		`COMPARE >`
		`COMPARE ==`
		`COMPARE !=`

Instruções de Controle de Fluxo		
`JMPTRUE <addr>`	Desvia o fluxo de execução para o endereço ou label indicado caso o valor no topo da pilha seja true. Um endereço deve sempre começar com #.	`JMPTRUE #10` `JMPTRUE label`
`JMPFALSE <addr>`	Desvia o fluxo de execução para o endereço ou label indicado caso o valor no topo da pilha seja false. Um endereço deve sempre começar com #.	`JMPFALSE #10` `JMPFALSE label`
`JMP <addr>`	Desvio incondicional de execução para o endereço ou label indicado. Um endereço deve sempre começar com #.	`JMP #10` `JMP label`
`CALL <func>`	Chamada por nome a um procedimento conhecido. O endereço de retorno é empilhado, um novo contexto é iniciado e apenas variáveis globais são acessíveis, além dos parâmetros do procedimento. Parâmetros devem ser empilhados antes da chamada.	`CALL func_name`
`RETURN`	Retorna o fluxo de controle ao endereço no topo da pilha. Caso haja um valor a ser retornado, este será colocado no topo da pilha.	`RETURN`
`LABEL`	Cria um novo rótulo referenciando o endereço onde este foi declarado. Pode ser referenciado por jumps.	`LABEL label1`

PROCEDURE <vars> <&objs>	Cria um novo rótulo no endereço declarado, determinando o início de um procedimento que recebe determinadas variáveis e objetos. Pode ser referenciado pela instrução CALL. Quando executado, o interpretador irá validar se o número de argumentos empilhado antes da chamada é igual ao número de argumentos declarados no label do procedimento.	PROCEDURE pos1 Sample:&s1 Plate:&p1
Instruções da Plataforma		
MOVE	Envia um comando à plataforma para mover amostras à posição indicada no topo da pilha, a partir da posição seguinte na pilha. As duas variáveis no topo da pilha devem necessariamente ser do tipo posição, ou um erro é disparado.	MOVE
PICKUP	Envia um comando à plataforma para pegar o número indicado de pipetas no rack indicado no topo da pilha. O número indicado de pipetas deve estar em segundo lugar no topo da pilha. Espera-se que o valor no topo da pilha seja um objeto implícito do tipo Rack.	PICKUP
DISCARD	Envia o comando à plataforma para descartar todas as pipetas em uso. Caso nenhuma pipeta esteja em uso, NOOP é executado.	DISCARD
Instruções NOOP e HALT		
NOOP	Instrução em que o interpretador permanece inoperante até o próximo ciclo.	NOOP
HALT	Instrução que finaliza a execução. Note que esta instrução não precisa ser utilizada no final do script, que é finalizado automaticamente ao encontrar EOF. Pode ser usada para finalizar a execução antes do EOF.	HALT
Instruções de Sistema		
PRINT	Imprime o valor do topo da pilha no terminal em que o interpretador está executando.	PRINT

3.4 Conversões Implícitas

O interpretador FCL realiza conversões implícitas para os operadores aritméticos, lógicos e de concatenação. Operadores aritméticos podem ser aplicados à strings, forçando uma conversão numérica implícita se possível, ou gerando um erro caso contrário; Operadores aritméticos também podem ser aplicados à tipos de dados de posição, facilitando a iteração sobre tais tipos. Operadores lógicos transformam implicitamente strings e listas em booleanos caso estas estejam vazias ou não. Operadores de concatenação, por sua vez, funcionam para todos os tipos de dados, concatenando-os em suas representações textuais.

4. Conclusão e Próximos Passos

Dada a complexidade do sistema, efetivar a integração do interpretador FCL com a plataforma ELISA será uma tarefa conjunta com a Celer Biotecnologia S/A. Obtivemos neste trabalho sucesso na concepção e na implementação de uma linguagem de programação inédita, ainda que baseada no modelo tradicional de linguagens funcionais. As próximas etapas envolvem um trabalho mais prático, com testes ostensivos para garantir a funcionalidade sem falhas da linguagem em conjunto com o equipamento.

O interpretador como o serviço responsável pelo controle completo do equipamento ainda não foi implementado. É evidente que, por se tratar de um sistema que irá operar em um equipamento de alto valor monetário, é necessário que tal ferramenta seja muito bem desenvolvida e testada, Para as finalidades deste trabalho de conclusão de curso, no entanto, nosso projeto teve êxito em implantar o que foi inicialmente proposto: a linguagem, o compilador e o interpretador.

5. Referências

Modelo SBC. Disponível:

http://www.sbc.org.br/documentos-da-sbc/category/169-templates-para-artigos-e-capitulos-de-livros

Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman: Compiladores:

Princípios, Técnicas e Ferramentas. 2ª edição. Dezembro / 2012.

Kaptur, A e Múltiplos outros autores: 500 Lines or Less. Disponível: http://aosabook.org/en/500L/a-python-interpreter-written-in-python.html

Ierusalimschy, R, Figueiredo, L e Celes, W: Lua 5.1 Reference Manual. Disponível: https://www.lua.org/manual/5.1/manual.html, Novembro/2012.

Ierusalimschy, R, Figueiredo, L e Celes, W: The Implementation of Lua 5.0. Disponível: https://www.lua.org/doc/jucs05.pdf

6. Sintaxe Formal

Segue a sintaxe formal da linguagem, na forma aumentada de Backus-Naur (BNF):

```
block ::= { statement }

statement ::=
      while exp do `:´ block end |
      for Identifier `=´ Range [step Value] do `:´ block end |
      foreach Identifier {`=´ Value} as Identifier do `:´ block end |
      if exp then block {elseif exp then block} [else block] end |
      procedurecall |
      procedure |
      vardecl |
      attr

procedure ::=
      procedure Identifier `(´ [ varlist ] `)´ { uses objlist } `:´
block end

procedurecall ::= run Identifier `(´ [ varlist ] `)´ [ with objlist ]

objlist ::= Identifier as Identifier { `,´ Identifier as Identifier }

attr ::=
      Identifier `=´ Value |
      Identifier `=´ procedurecall |
      Identifier `=´ `{´ [ valuelist ] `}´

varlist ::= Identifier { `,´ Identifier }

vardecl ::=
      [ var ] Identifier |
      [ var ] attr

exp ::=
      `(´ exp `)´ | none | true | false | Number | String | var |
      Position | procedurecall | exp binaryop exp | unaryop exp

binaryop ::= `+´ | `-´ | `*´ | `/´ | `%´ | `..´ | `==´ | `!=´ |
`<´ | `>´ | `<=´ | `>=´

unaryop ::= `!´
```