UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

MAC5754: CONCEITOS DE LINGUAGENS DE PROGRAMAÇÃO

LEANDRO TIBURSKE

NAHIM ALVES DE SOUZA

VITOR TERRA

ROSA: LINGUAGEM DE PROGRAMAÇÃO PARA ANÁLISE DE SEQUÊNCIAS BIOLÓGICAS

Professora:

Ana C. V. de Melo

São Paulo

2022

SUMÁRIO

1 DOMÍNIO DE APLICAÇÃO **3**
1.1 Requisitos 3
1.2 ROSA: Linguagem de programação para bioinformática 6

2 DEFINIÇÕES DA LINGUAGEM **7**
2.1 Tipos **7**
2.1.1 Primitivos 7
2.1.2 Compostos 7
2.1.3 Recursivo 8
2.2 Expressões e Comandos **8**
2.2.1 Expressões 8
2.2.2 Comandos 9
2.3 Abstrações **10**
2.3.1 ins(Sequence seq, Sequence fragment, Integer position) 10
2.3.2 del(Sequence seq, Integer start, Integer end) 10
2.3.3 point(Sequence seq, Integer position, Sequence monomer) 10
2.3.4 transcribe(Sequence seq) 11
2.3.5 translate(Sequence seq) 11
2.3.6 complement(Sequence seq) 11
2.3.7 motif(Sequence seq, Sequence motif) 11
2.3.8 qctrl(FASTQ file, Integer threshold, Integer range) 11
2.3.9 trim(FASTQ ou FASTA file, Sequence adapters, Sequence primers) 11
2.3.10 tofasta(FASTQ file) 11
2.3.11 read(String path) 12
2.3.12 readfa(String path) 12
2.3.13 readfq(String path) 12
2.3.14 write(file, String name, String path) 12
2.3.15 writefa(FASTA file, String name, String path) 12
2.3.15 writefq(FASTQ file, String name, String directory) 12
2.3.16 input(String prompt) 12
2.3.17 print(data) 12

3 SINTAXE **13**
3.1 Itens modificados - Fase 1 e 2 **13**
3.2 Notação Extended Backus Naur Form (EBNF) 13
3.3 Palavras reservadas 14
3.4 Símbolos terminais 14
3.5 Sintaxe de valores 14
3.5.1 Dígitos 14

3.5.2 Caracteres 15
3.5.3 Booleanos 15
3.5.4 Números inteiros 15
3.5.5 Números reais 15
3.5.6 String 16
3.5.7 Sequence 16
3.5.8 Quality 16
3.5.9 FASTA 16
3.5.10 FASTQ 17
3.6 Identificadores 17
3.7 Tipos de dados 17
3.8 Operadores 18
3.8.1 Operadores lógicos 18
3.8.2 Operadores aritméticos 18
3.8.3 Operadores relacionais 18
3.8.4 Operadores booleanos 18
3.8.4 Operadores de sequências 18
3.9 Regras sintáticas 19
3.9.1 Sintaxe de expressões 19
3.9.2 Sintaxe de comandos 20
3.9.3 Sintaxe dos programas 20
3.10 Exemplo de programa escrito em ROSA 20

5 SEMÂNTICA DENOTACIONAL 21
5.1 Domínios Sintáticos 21
5.2 Sintaxe Abstrata 21
5.3 Domínios Semânticos 22
5.4 Funções Semânticas 22
5.5 Equações Semânticas 23
5.6 Erros Semânticos 25

6 IMPLEMENTAÇÃO 29
6.1 Estrutura dos arquivos 29
6.2 Instruções para uso do interpretador 30
6.3 Funcionalidades do interpretador 31
6.4 Problemas encontrados 31

7 REFERÊNCIAS 32

1 DOMÍNIO DE APLICAÇÃO

1.1 Requisitos

Algo muito recorrente no campo de ciências biológicas é precisar lidar com sequências de macromoléculas, como DNA e proteínas, assim como com os diferentes formatos de arquivos para armazenar esses dados. Entretanto, embora existam bibliotecas que facilitam a manipulação de dados moleculares, como a Biopython (COCK *et al*., 2009), atualmente, não existe uma linguagem de programação específica para o domínio com o fim de facilitar a manipulação de tal tipo de dados. A partir disso, propõe-se a criação de uma linguagem de programação específica para a manipulação de dados e sequências biológicas, assim como de seus diferentes formatos de arquivos e as equivalências entre cada um.

Em geral, as informações moleculares são lidas de arquivos que armazenam os dados em formatos padronizados, como SAM/BAM, FASTA e FASTQ. Assim, é necessário que a linguagem seja capaz de carregar os dados desses arquivos e armazenar em tipos ou estruturas que permitam a sua manipulação. Os arquivos FASTA possuem um cabeçalho, o qual começa com “>”, seguido de uma uma sequência biológica cujos aminoácidos ou nucleotídeos são representados de acordo com o formato IUB/IUPAC (PEARSON; LIPMAN, 1998). Os símbolos suportados pelo formato FASTA para ácidos nucleicos e para proteínas são mostrados pela Figura 1 e Figura 2, respectivamente:

Figura 1 - Codificação de nucleotídeos

Symbol	Meaning	Description Origin
G	G	**G**uanine
A	A	**A**denine
T	T	**Thymine**
C	C	**C**ytosine
R	G or A	pu**R**ine
Y	T or C	p**Y**rimidine
M	A or C	a**M**ino
K	G or T	**K**etone
S	G or C	**S**trong interaction
W	A or T	**W**eak interaction
H	A or C or T	**H** follows G in alphabet
B	G or T or C	**B** follows A in alphabet
V	G or C or A	**V** follows U in alphabet
D	G or A or T	**D** follows C in alphabet
N	G or A or T or C	a**N**y

Fonte: KAMATH; DE JONG; SHEHU, 2014

Figura 2 - Codificação de aminoácidos

Amino Acid	3 Letter Abbreviation	IUPAC Notation	Translating Codons
Alanine	Ala	A	GCT, GCC, GCA, GCG
Arginine	Arg	R	CGT, CGC, CGA, CGG, AGA, AGG
Asparagine	Asn	N	AAT, AAC
Aspartic acid	Asp	D	GAT, GAC
Cysteine	Cys	C	TGT, TGC
Glutamine	Gln	Q	CAA, CAG
Glutamic acid	Glu	E	GAA, GAG
Glycine	Gly	G	GGT, GGC, GGA, GGG
Histidine	His	H	CAT, CAC
Isolucine	Ile	I	ATT, ATC, ATA
Methionine	Met	M	ATG
Leucine	Leu	L	TTA, TTG, CTT, CTC, CTA, CTG
Lysine	Lys	K	AAA, AAG
Phenylalanine	Phe	F	TTT, TTC
Proline	Pro	P	CCT, CCC, CCA, CCG
Serine	Ser	S	TCT, TCC, TCA, TCG, AGT, AGC
Threonine	Thr	T	ACT, ACC, ACA, ACG
Tryptophan	Trp	W	TGG
Tyrosine	Tyr	Y	TAT, TAC
Valine	Val	V	GTT, GTC, GTA, GTG
STOP	Stop	*	TAA, TGA, TAG

Fonte: IUPAC

Por meio das representações discutidas, é possível representar sequências biológicas, como mostrado abaixo:

```
>EXEMPLO_DE_PROTEÍNA

MTEITAAMVKELRESTGAGMMDCKNALSETNGDFDKAVQLLREKGLGKAAKKADRLAAEG
```

Os arquivos FASTQ, além de armazenarem informações referentes à composição de macromoléculas, também guardam dados referentes à qualidade do sequenciamento, ou seja, são arquivos gerados por conta de experimentos. Para cada nucleotídeo, há um *score* de qualidade que indica a certeza de que o monômero especificado realmente está presente na molécula. Geralmente, os arquivos FASTQ são divididos em quatro linhas diferentes: a primeira, opcional, começa com "@" e é equivalente ao cabeçalho do formato FASTA; a segunda linha são as representações das sequências em si; a terceira linha começa com "+" e é opcionalmente seguida pelo identificador da sequência; a quarta linha possui os símbolos

referentes à qualidade do sequenciamento e devem estar presentes na mesma quantidade que os caracteres na linha 2 (COCK *et al*., 2010). Os valores de qualidade Phred são representados por codificação ASCII, sendo que o valor de qualidade aumenta, a seguir, da esquerda para a direita. Possuem equivalência numérica no intervalo de 0 a 93.

```
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]
^_`abcdefghijklmnopqrstuvwxyz{|}~
```

Portanto, um exemplo de formato FASTQ é:

```
@SEQ_ID
GATTTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTTGTTCAACTCACAGTTT
+
!''*((((***+))%%%++)(%%%%).1***-+*''))**55CCF>>>>>>CCCCCCC65
```

Alguns exemplos de operações que podem ser realizadas nesses dados são citadas abaixo:

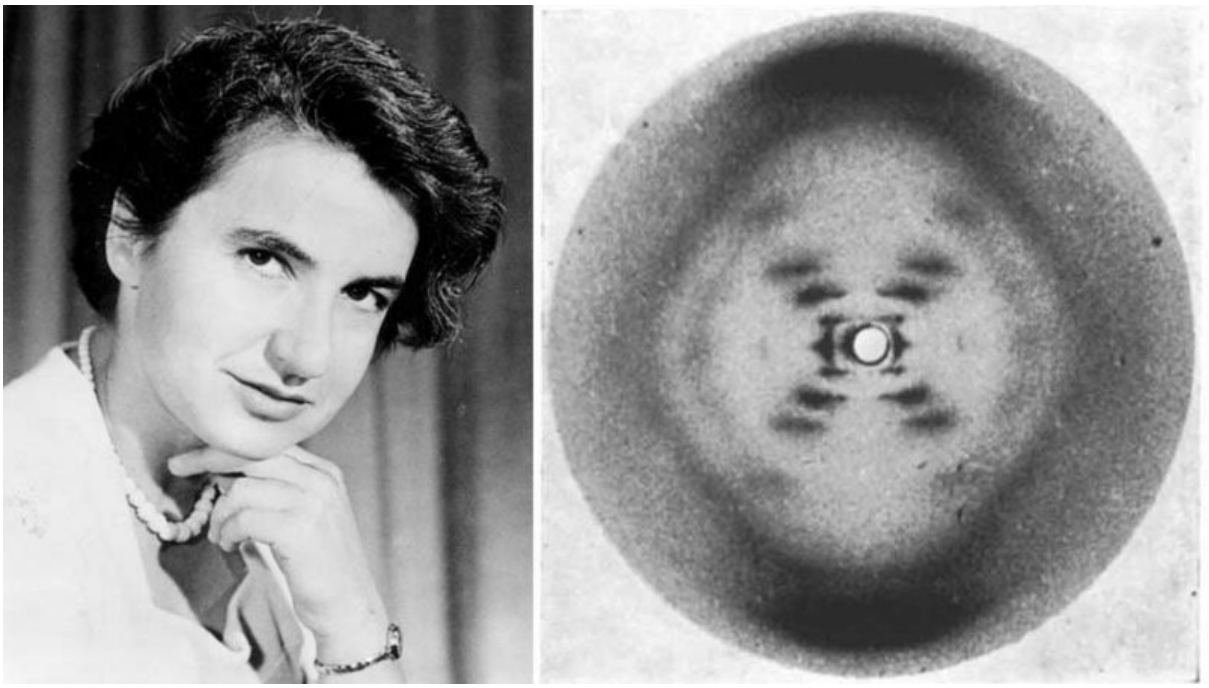
1. Obter *subset* de uma sequência;
2. Mutação de sequências (apenas sequências puras ou FASTA);
    a. Alteração de um único caracter;
    b. Deleção de parte da sequência;
    c. Inserção de outra sequência.
3. Transcrição e tradução de sequências;
4. Complemento de DNA e RNA;
5. Encontrar *motif* em sequências;
6. Alinhamento;
7. Controle de qualidade de dados FASTQ;
8. Trimar sequências primers e adaptadores;
9. Conversões entre FASTA e FASTQ.

1.2 ROSA: Linguagem de programação para bioinformática

Rosalind Franklin (1920-1958) foi uma química britânica responsável pela elucidação da estrutura do DNA, RNA, vírus, grafite e carvão mineral. Apesar de seu trabalho com difração de raios-X para estudar DNA ter sido uma das principais evidências da estrutura de dupla hélice da molécula, a cientista não foi reconhecida pelo prêmio Nobel e por muitos anos suas descobertas foram atribuídas aos seus colegas homens, os quais utilizaram dados e fotografias não publicados por Franklin sem autorização.

Como uma forma de homenagear Rosalind Franklin por suas descobertas científicas e tentar reparar a falta de reconhecimento dado à cientista, a linguagem de programação proposta a seguir será chamada ROSA. A linguagem ROSA será voltada para a análise de dados de macromoléculas biológicas, como DNA, RNA e proteínas, as quais só são bem compreendidas atualmente por conta do trabalho da pesquisadora homenageada.

Figura 3 - Retrato de Rosalind Franklin e fotografia da estrutura do DNA



2 DEFINIÇÕES DA LINGUAGEM

2.1 Tipos

A linguagem será fortemente tipificada e os tipos serão definidos estaticamente, de forma explícita - por exemplo: `Integer a = 5;` - a seguir, são apresentados os tipos primitivos e compostos permitidos na linguagem.

2.1.1 Primitivos

- `Integer`: números inteiros, que permitem a realização de operações aritméticas básicas (+, -, *, /);
- `Real`: números reais, que permitem a realização de operações aritméticas básicas (+, -, *, /);
- `Boolean`: valores True ou False, que podem ser utilizados em operações lógicas básicas (NOT, AND, OR);
- `String`: sequência de caracteres (letras, números e símbolos);
- `Sequence`: sequência de caracteres presentes na codificação IUPAC de aminoácidos ou na codificação IUPAC para nucleotídeos;
- `Quality`: sequência de caracteres ASCII, cujos valores de qualidade são correspondentes ao proposto pelo formato FASTA.

2.1.2 Compostos

- `FASTA`: Composto por uma string (a qual representa o cabeçalho do arquivo) e por uma sequence. Armazena dados de sequências biológicas no geral; pode conter só a informação para uma única molécula ou ser um multi-FASTA (como mostrado abaixo).

```
>gi|186704|Keratin Homo sapiens keratin x=42 [yyy] (xxx)
CCCAGGGTCCGATGGGAAAGTGTAGCCTGCAGGCCCACACCTCCCCCTGTGAATCACGCCT
GGCGGGACAAGAAAGCCCAAAACACTCCAAACAATGAGTTTCCAGTAAAATATGACAGACA
TGATGAGGCGGATGAGAGGAGGGACCTGCCTGGGAGTTGGCGCTAGCCTGTGGGTGATGAA
AGCCAAGGGGAATGGAAAGTGCCAGAC
```

- `FASTQ`: armazena dados de um formato de dado biológico parecido com o FASTA, mas criado por conta de um experimento de sequenciamento - temos como informações a sequência em si, mas também um *score* de qualidade que representa o quanto de certeza temos de que aquele nucleotídeo realmente está presente na molécula. Logo, é composto por valores do tipo Sequence, Quality e String. Exemplo:

```
@SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=36
GGGTGATGGCCGCTGCCGATGGCGTCAAATCCCACC
+SRR001666.1 071112_SLXA-EAS1_s_7:5:1:817:345 length=36
IIIIIIIIIIIIIIIIIIIIIIIIIIIII9IG9IC
```

2.1.3 Recursivo

- `List of <Tipo>:` lista de valores de qualquer tipo primitivo ou composto. Esse tipo será desenvolvido principalmente para permitir o agrupamento de diversas sequências biológicas, como ocorre, por exemplo, na leitura de um arquivo multi-FASTA ou FASTQ com mais de uma leitura.

2.2 Expressões e Comandos

Nesta seção, são definidos as expressões e os comandos que serão utilizados na linguagem para a manipulação dos dados e controle do estado do programa.

2.2.1 Expressões

1. Literais: são valores que representam uma instância dos tipos primitivos, apresentados na seção anterior, ou seja, números (Integer, Real), valores não-numéricos (Boolean) e sequências de um ou mais caracteres (String, Sequence, Quality);
2. Agregação de Valores: todos os literais poderão ser agrupados através de listas (*arrays*) onde todos os elementos são de um mesmo tipo e podem ser acessados individualmente através de um índice que representa a posição do elemento na lista;
3. Funções: a linguagem aceita a chamada de funções, compostas por um identificador e uma sequência de parâmetros entre parênteses - por exemplo: *func(param1, param2)*. São aceitas funções:
    a. Pré-definidas de acordo com tipo do elemento (algumas delas são citadas na seção 2.3 - Abstrações);

b. Definidas pelo usuário através da especificação do tipo de retorno e dos parâmetros da função;
c. Aplicadas através de operadores - no caso de valores numéricos (Integer e Real), são suportadas as quatro operações aritméticas básicas (+, -, *, /) e comparações (>, >=, <, <=, ==); para elementos do tipo Sequence, serão válidas operações de concatenação (+), inversão (^) e comparação de igualdade (==); para elementos do tipo Boolean, serão válidas operações lógicas NOT, AND e OR e comparação de igualdade (==).

4. Expressões condicionais: são aceitas expressões que retornam um valor de acordo com a condição (expressão lógica) definida, conforme o exemplo abaixo:
   a. `if condicao then valor1 else valor2`
5. Ordem de avaliação das expressões:
   a. Expressões entre parênteses possuem maior precedência;
   b. Operadores unários ^ e NOT possuem precedência sobre operadores binários;
   c. Para expressões aritméticas, divisão (/) e multiplicação (*) precedem soma (+) e subtração (-);
   d. Para expressões lógicas, a precedência é NOT, AND, OR.

2.2.2 Comandos

6. Atribuição - uma expressão ou literal pode ser vinculada a uma variável (com tipo pré-definido) através do operador '='
7. Instruções compostas e blocos
   a. As instruções serão separadas por ponto-e-vírgula ';'
   b. O início e o fim de blocos de instruções serão delimitados respectivamente pelos caracteres '{' e '}'
8. Condicionais - são válidos comandos de seleção n-direcional através das palavras reservadas: *if*, *else* e *elif*, como no exemplo a seguir:

```
if (condicao) {
     bloco_de_instrucoes;
} elif (condicao) {
     bloco_de_instrucoes;
} else {
```

```
        bloco_de_instrucoes;
    }
```

9. Iterativos - são aceitos comandos de laço iterativos, que executam um bloco de código repetidamente enquanto uma determinada condição não for satisfeita. Exemplo:

```
while (condicao) {
        bloco_de_instrucoes;
}
```

2.3 Abstrações

Tendo em vista que muitos profissionais que lidam com ciências biológicas não possuem base teórica em programação, é necessário que uma linguagem voltada para a manipulação de dados biológicos seja de fácil compreensão e siga uma estrutura lógica condizente com o raciocínio biológico ao empregar representações que retomam os jargões da área.

2.3.1 `ins(`**`Sequence`** `seq,` **`Sequence`** `fragment,` **`Integer`** `position)`

A abstração `ins`() permitirá que o usuário realize uma mutação do tipo *insertion*. Portanto, os parâmetros necessários serão a sequência de interesse (`sequence`), sequência a ser inserida (`fragment`) e posição da sequência original em que a mutação ocorrerá (`position`).

2.3.2 `del(`**`Sequence`** `seq,` **`Integer`** `start,` **`Integer`** `end)`

A abstração `del`() permitirá que o usuário realize uma mutação do tipo *deletion*. Portanto, os parâmetros necessários serão a sequência de interesse (`sequence`), posição inicial de deleção (`start`) e posição final de deleção `(end)`.

2.3.3 `point(`**`Sequence`** `seq,` **`Integer`** `position,` **`Sequence`** `monomer)`

A abstração `point`() permitirá que o usuário realize uma alteração do tipo *point mutation*. Portanto, os parâmetros necessários serão a sequência de interesse (`sequence`), posição de substituição (`position`) em que o nucleotídeo ou aminoácido passado através do parâmetro `monomer` se localizará.

2.3.4 `transcribe(`**`Sequence`** `seq)`

Permitirá a transcrição de uma sequência. Poderá ser aplicada somente a nucleotídeos presentes em DNA.

2.3.5 `translate(`**`Sequence`** `seq)`

Possibilitará a tradução de uma sequência de nucleotídeos em uma sequência proteica. Terá como parâmetro uma sequência de ácido nucleico de interesse.

2.3.6 `complement(`**`Sequence`** `seq)`

Retorna a complementaridade de uma sequência de DNA ou RNA no sentido 5'-3'.

2.3.7 `motif(`**`Sequence`** `seq,` **`Sequence`** `motif)`

Em biologia molecular, *motifs* são regiões de sequências encontradas em diferentes moléculas e que possuem significado funcional. Assim, ao receber como parâmetros uma sequência de interesse e um *motif*, a presente abstração retornará as posições de ocorrência.

2.3.8 `qctrl(`**`FASTQ`** `file,` **`Integer`** `threshold,` **`Integer`** `range)`

Realiza controle básico de qualidade das sequências oriundas de arquivos FASTQ. Recebe como parâmetro a molécula de interesse (`file`), o valor de qualidade mínimo (`threshold`) e quantos nucleotídeos precisam estar abaixo do limiar para que haja remoção (`range`).

2.3.9 `trim(`**`FASTQ`** `ou` **`FASTA`** `file,` **`Sequence`** `adapters,` **`Sequence`** `primers)`

Remove adaptadores e primers da sequência de interesse, os quais são resquícios experimentais do sequenciamento sem real valor biológico. Logo, recebe como parâmetro uma sequência (seja de FASTQ ou FASTA) e as sequências dos primers e adaptadores utilizados.

2.3.10 `tofasta(`**`FASTQ`** `file)`

Converte dado proveniente de um arquivo FASTQ para formato FASTA. Portanto, apenas retira a informação referente à qualidade do sequenciamento. É importante notar que o inverso não pode ser feito, tendo em vista que não há como inferir valores de qualidade para

nucleotídeos de uma sequência de referência (formato FASTA). Possui como parâmetro o dado de FASTQ e, opcionalmente, um novo cabeçalho.

2.3.11 `read(String path)`

Carrega os dados de um arquivo cujo nome e localização é indicado pelo parâmetro path.

2.3.12 `readfa(String path)`

Carrega dados armazenados em um arquivo em formato FASTA. O parâmetro necessário para a abstração é o local do arquivo de interesse.

2.3.13 `readfq(String path)`

Carrega dados armazenados em um arquivo em formato FASTQ. O parâmetro necessário para a abstração é o local do arquivo de interesse.

2.3.14 `write(file, String name, String path)`

Cria um arquivo, cujo nome segue a String fornecida através do parâmetro path, com os dados de file e no diretório indicado por path.

2.3.15 `writefa(FASTA file, String name, String path)`

Cria um arquivo em formato FASTA. O parâmetro necessário para a abstração é o diretório em que o usuário deseja salvar os dados.

2.3.15 `writefq(FASTQ file, String name, String directory)`

Cria um arquivo em formato FASTQ. O parâmetro necessário para a abstração é o diretório em que o usuário deseja salvar os dados.

2.3.16 `input(String prompt)`

Permite que o usuário escreva algo e retorna dados em forma de String. Possui como parâmetro opcional a String prompt, a qual é mostrada ao usuário antes de permitir a escrita.

2.3.17 `print(data)`

A abstração print() permite que dados de qualquer tipo (primário, composto ou recursivo) sejam mostrados ao usuário.

3 SINTAXE

3.1 Itens modificados - Fase 1 e 2

1. Acréscimo das abstrações read(), write(), print() e input();
2. Acŕescimo do tipo recursivo `List`;
3. O operador de concatenação (+) será sobrecarregado para os tipos Sequence e String;
4. Sequências terão que começar com um símbolo terminal para indicar a qual grupo de macromoléculas biológicas pertencem: d (DNA), r (RNA) ou p (proteína).

3.2 Notação Extended Backus Naur Form (EBNF)

Com a finalidade de definir a sintaxe da linguagem ROSA, o Formalismo de Backus-Naur Estendido (EBNF) foi utilizado. A seguir estão listados as anotações empregadas da EBNF e uma breve descrição de seus significados:

Tabela 1: Descrição das anotações da EBNF

Símbolo	Definição
<>	Indica símbolo não-terminal
::=	Indica a definição de um não-terminal
|	Representa alternação
[]	Representa um opcional
{}*	Indica zero ou mais vezes
{}+	Indica uma ou mais vezes
()	Representa grupamento
‘’	Representa símbolo terminal

3.3 Palavras reservadas

A linguagem de programação ROSA possuirá algumas palavras reservadas, as quais são:

if elif else then while True
False Integer Real String Boolean
FASTA FASTQ Quality Sequence List
of ins del point transcribe translate
complement motif qctrl trim tofasta
read readfa readfq writefa writefq
print input NOT AND OR

3.4 Símbolos terminais

- Dígitos;
- Caracteres;
- Booleanos;
- Operadores relacionais;
- Operadores lógicos;
- Operadores aritméticos;
- Operador de inversão;
- Operadores de sequências;
- Identificadores.

3.5 Sintaxe de valores

3.5.1 Dígitos

Os dígitos em ROSA são representados por:

```
<digito> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

3.5.2 Caracteres

Todos os caracteres aceitos pela linguagem ROSA estão definidos a seguir:

```
<caracter> ::= <letra> | <digito> | <simbolo> |
<letra_especial>


<letra> ::= a | b | c | d | e | f | g | h | i | j | k | l | m
| n | o | p | q | r | s | t | u | v | w | x | y | z | A | B |
C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R
| S | T | U | V | W | X | Y | Z


<simbolo> ::= ! | " | # | $ | % | & | ' | ( | ) | * | + | , |
- | . | / | : | ; | < | = | > | ? | @ | [ | \ | | ] | ^ | _ |
` | { | | | } | ~


<letra_especial> ::= á | â | à | ã | é | ê | è | í | î | ì |
ó | ô | ò | õ | ú | û | ù | ç | ñ | ý | Á | Â | À | Ã | É | Ê
| È | Í | Î | Ì | Ó | Ô | Ò | Õ | Ú | Û | Ù | Ç | Ñ | Ý
```

3.5.3 Booleanos

A definição de valores do tipo Booleano em ROSA é feita da seguinte maneira:

```
<Boolean> ::= True | False
```

3.5.4 Números inteiros

Os números inteiros formam um tipo primitivo em ROSA, chamado de Integer:

```
<Integer> ::= [ + | - ] {<digito>}+
```

3.5.5 Números reais

Os valores reais também compõem um tipo primitivo em ROSA:

```
<Real> ::= <Integer> . {<digito>}+
```

3.5.6 String

O tipo primitivo String é composto por uma sequência de um ou mais caracteres. Possui “ e ” como símbolos terminais:

```
<String> ::= ‘“’ {<caracter>}* ‘”’
```

3.5.7 Sequence

Como já discutido em 2.1.1, a linguagem ROSA possui como tipo primitivo Sequence, a qual serve para representar dados de macromoléculas biológicas.

```
<DNA> ::= A | C | G | T | R | Y | S | W | K | M | B | D | H |
V | N | . | -

<protein> ::= A | R | N | D | C | Q | E | G | H | I | M | L |
K | F | P | S | T | W | Y | v | *

<RNA> ::= A | C | G | U | R | Y | S | W | K | M | B | D | H |
V | N | . | -


<Sequence> ::= (‘d’ {<DNA>}* ) | (‘r’ {<RNA>}* ) |
(‘p’ {<protein>}*)
```

3.5.8 Quality

Além de possuir um tipo primitivo para representação de sequências biológicas, a linguagem ROSA representa valores de qualidade, como já apresentado em 2.1.2, por meio do tipo primitivo Quality. Dessa maneira, é possível analisar dados de arquivos FASTQ, os quais são provenientes de sequenciamento de nucleotídeos.

```
<Quality> ::= {<simbolo>}* | {<letra>}*
```

3.5.9 FASTA

```
<FASTA> ::= ‘>’ <String>
<Sequence>
```

3.5.10 FASTQ

```
<FASTQ> ::= '@' <String>
<Sequence>
'+' [<Sequence>]
<Quality>
```

3.6 Identificadores

```
<identificador> ::= <letra> {( _ | <letra> | <digito>)}*

<identificador_fun> ::= ins | del | point | transcribe |
translate | complement | motif | qctrl | trim | tofasta |
read | readfa | readfq | write | writefa | writefq | print |
input
```

3.7 Tipos de dados

```
<tipo> ::= <tipo_primitivo> |
<tipo_composto> |
<tipo_recursivo>
```

```
<tipo_primitivo> ::= Integer | Real | Boolean | String |
Sequence | Quality
```

```
<tipo_composto> ::= FASTA | FASTQ
```

```
<tipo_recursivo> ::= List of (<tipo_primitivo> |
<tipo_composto>)
```

3.8 Operadores

3.8.1 Operadores lógicos

```
<operador_log> ::= NOT | AND | OR
```

3.8.2 Operadores aritméticos

```
<operador_arit> ::= + | - | / | *
```

3.8.3 Operadores relacionais

```
<operador_rel> ::= > | >= | < | <= | ==
```

3.8.4 Operadores booleanos

```
<operador_bool> ::= <operador_log> | <operador_rel>
```

3.8.4 Operadores de sequências

```
<operador_concat> ::= +

<operador_invert> ::= ^
```

3.9 Regras sintáticas

3.9.1 Sintaxe de expressões

```
<expressao> ::= <Integer> | <Real> | <Boolean> | <String> |
<Sequence> | <FASTA> | <FASTQ> | <List> | <acessa_lista> |‘(’
<expressao> ‘)’ | <operacao_arit> | <operacao_log> |
<operacao_invert> | <operacao_concat> | <chamada>


<operacao_arit> ::= <expressao> <operador_arit> <expressao>


<expressao_log> ::= <operacao_log> <operador_bool>
<operacao_log>

<operacao_log> ::= <expressao_log> | ‘(’<expressao_log> ‘)’ |
<expressao>


<expressao_invert> ::= <operador_invert> <Sequence>

<operacao_invert> ::= <expressao_invert> | ‘(’
<expressao_invert> ‘)’ | <expressao>


<expressao_concat> ::= <Sequence> <operador_concat>
<Sequence> | <String> <operador_concat> <String>

<operacao_concat> ::= <expressao_concat> | ‘(’
<expressao_concat> ‘)’ | <expressao>


<List> ::= ‘[’ <expressao> {‘,’ <expressao>}* ‘]’

<acessa_lista> ::= <identificador> ‘[’ <Integer> ‘]’
```

3.9.2 Sintaxe de comandos

```
<comando> ::= (<atribuicao> | <condicional> | <iterativo> |
<chamada>) [<comando>]

<atribuicao> ::= <tipo> <identificador> [‘=’ <expressao>]

<condicional> ::= if ‘(’ <expressao_logica> ‘)’
‘{’ <comando> ‘}’
[{ elif ‘(‘ <expressao_logica> ‘)’
‘{’ <comando> ‘}’ }*]
[else <comando>]

<iterativo> ::= while ‘(’ <expressao_logica> ‘)’
‘{’ <comando> ‘}’


<chamada> ::= <identificador_fun> ‘(’ {<expressao>}+ ‘)’ ‘;’
```

3.9.3 Sintaxe dos programas

```
<programa> ::= <expressao> | <comando> | <bloco>

<bloco> ::= ‘{’ {<expressao> | <comando>}* ‘}’
```

3.10 Exemplo de programa escrito em ROSA

```
String path_fastq = “~/Documents/FASTQ/SRR001666_1.fastq”;
String path_adapters = “~/Documents/FASTQ/adapter.fasta”;
String path_primer = “~/Documents/FASTQ/primer.fasta”;

FASTQ data = readfq(path_fastq);
FASTA adapter = readfa(path_adapter);
FASTA primer = readfa(path_primer);

data = qctrl(data);
data = trim(data, primer, adapter);

FASTA dna = tofasta(data)
Sequence mRNA = transcribe(data);
```

```
List of Sequence miRNAs = [rUUCCCUUUGUCAUCCUAUGCCU,
                           rUUCGCUAUGUCAUCCUAUUCCU,
                           rAAGGCUUUGUCAUCGUAUGCCU];

Integer posicao = 0;
while (posicao < 3) {
     print(motif(mRNA, miRNAs[posicao]))
     posicao = posicao + 1
}
```

5 SEMÂNTICA DENOTACIONAL

5.1 Domínios Sintáticos

Os domínios sintáticos utilizados na linguagem ROSA são:

- *Expressao* = {*e* | *e* é uma expressão}
- *Expressao_Logica* = {*l* | *l* é uma expressão lógica}
- *Identificador* = {*i* | *i* é um identificador}
- *Comando* = {*c* | *c* é um comando}
- *Numero* = {*n* | *n* é um número inteiro ou real}
- *String* = {*s* | *s* é uma string}
- *Sequence* = {*sq* | *sq* é uma instância de Sequence}
- *Quality* = {*q* | *q* é uma instância de Quality}
- *FASTA* = {*fa* | *fa* é uma instância de FASTA}
- *FASTQ* = {*fq* | *fq* é uma instância de FASTQ}

5.2 Sintaxe Abstrata

Considerando os domínios definidos na seção anterior, a definição da gramática na linguagem ROSA é a seguinte:

- $E \in Expressao = n \mid s \mid i \mid sq \mid q \mid fa \mid fq \mid E + E \mid E * E \mid E - E \mid E / E \mid (E) \mid f\ (E)$
- $B \in Expressao\_Logica = True \mid False \mid i \mid B \wedge B \mid B \vee B \mid B == B \mid B <> B \mid B > B \mid B < B \mid B \geq B \mid B \leq B \mid f\ (B)$
- $C \in Comando = i=E; \mid C;C \mid if\ B\ \{C\}\ else\ \{C\} \mid while\ B\ \{C\}$

5.3 Domínios Semânticos

Os domínios semânticos são dados pelas equações abaixo.

- **Z** : Números inteiros
- **R** : Números reais
- **A*** : Sequência de caracteres
- **B** = {True, False}
- **dSeq** = Caracteres válidos em uma sequência de DNA
- **rSeq** = Caracteres válidos em uma sequência de RNA
- **pSeq** = Caracteres válidos em uma sequência de proteína
- **Seq** = (**dSeq** ∪ **rSeq** ∪ **pSeq**)
- **Q** = Caracteres válidos para representar um *score* de qualidade
- **Fa** = (A*, Seq) - Tupla composta por um String e Sequence
- **Fq** = (A*, Seq, Q) - Tripla composta por String, Sequence e Quality.
- **Σ** : Possíveis estados do programa
- **T** = (**Z** ∪ **R** ∪ **A*** ∪ **Seq** ∪ **Q** ∪ **Fa** ∪ **Fq**) - União de tipos primitivos e compostos
- **L** = {**null** ∪ (**Z** × **T**)} - Lista de elementos representada pelo produto cartesiano entre um inteiro (índice) e um elemento do tipo **T**.
- Função = {ins, del, point, transcribe, translate, complement, motif, qctrl, trim, tofasta, readfa, readfq, writefa, writefq, print, input}
- f ∈ Função

O operador ∪ representa a união disjunta e × o produto cartesiano de dois conjuntos. O símbolo **null** representa a lista vazia.

5.4 Funções Semânticas

As relações entre os domínios são dadas por:

E : Expressao → (Σ → (**Z** ∪ **R** ∪ **A*** ∪ **L** ∪ **Seq** ∪ **Q** ∪ **Fa** ∪ **Fq**))

L : Expressao_Logica → (Σ → **B**)

C : Comando → (Σ → Σ)

5.5 Equações Semânticas

Para as expressões básicas da linguagem temos as seguintes equações:

- $E[[n]] = \{(\sigma, n) \mid \sigma \in \Sigma \wedge (n \in \mathbf{Z} \vee n \in \mathbf{R} \vee n \in \mathbf{B} \vee n \in \mathbf{A^*} \vee n \in \mathbf{Seq} \vee n \in \mathbf{Q} \vee n \in \mathbf{Fa} \vee n \in \mathbf{Fq})\}$
- $E[[x]] = \{(\sigma, \sigma(x)) \mid \sigma \in \Sigma\}$
- $E[[f(e_1, \ldots, e_k)]] = \{(\sigma, \sigma(f(E[[e_1]]\sigma, \ldots, E[[e_k]]\sigma))) \mid \sigma \in \Sigma\}$

Na linguagem ROSA, podemos manipular uma expressão através dos operadores aritméticos (+, -, *, /), caso o valor seja um número inteiro ou real. Adicionalmente, também utilizamos o símbolo + para representar a concatenação de instâncias do tipo String ou Sequence. Desse modo, temos as seguintes equações semânticas que representam essas operações:

- $E[[a_0 + a_1]] = \{(\sigma, (n_0 + n_1)) \mid \sigma \in \Sigma \wedge n_0 = E[[a_0]]\sigma, n1 = E[[a_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R})\} \cup \{(\sigma, (n_0 n_1)) \mid \sigma \in \Sigma \wedge n_0 = E[[a_0]]\sigma, n_1 = E[[a_1]]\sigma \wedge (n_0, n_1 \in \mathbf{A^*} \vee n_0, n_1 \in \mathbf{Seq})\}$
- $E[[a_0 - a_1]] = \{(\sigma, (n_0 - n_1)) \mid \sigma \in \Sigma \wedge n_0 = E[[a_0]]\sigma, n1 = E[[a_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R})\}$
- $E[[a_0 * a_1]] = \{(\sigma, (n_0 * n_1)) \mid \sigma \in \Sigma \wedge n_0 = E[[a_0]]\sigma, n1 = E[[a_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R})\}$
- $E[[a_0 / a_1]] = \{(\sigma, (n_0 / n_1)) \mid \sigma \in \Sigma \wedge n_0 = E[[a_0]]\sigma, n1 = E[[a_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R}) \wedge n \neq 0\}$

O símbolo ^ é utilizado como operador unário, para representar a inversão de uma instância do tipo String ou Sequence. Visto que **n** representa um valor composto por uma sequência de caracteres $\{n_0 n_1 \ldots n_{m-1} n_m\}$ onde *m* é o tamanho da sequência, utilizamos o termo **inverso(n)** para representar a sequência inversa, ou seja, $\{n_m n_{m-1} \ldots n_1, n_0\}$.

- $E[[\hat{}a]] = \{(\sigma, \text{inverso}(n)) \mid \sigma \in \Sigma \wedge n = E[[a]]\sigma \wedge (n \in \mathbf{A^*} \vee n \in \mathbf{Seq})\}$

A linguagem ROSA também suporta expressões condicionais, que são denotadas da seguinte forma:

- $E[[\text{if } l \text{ then } a_0 \text{ else } a_1]] = \{(\sigma, n_0) \mid \sigma \in \Sigma \wedge L[[l]]\sigma = \text{True} \wedge n_0 = E[[a_0]]\sigma \wedge (n_0 \in \mathbf{Z} \vee n_0 \in \mathbf{R} \vee n_0 \in \mathbf{B} \vee n_0 \in \mathbf{A^*} \vee n_0 \in \mathbf{Seq} \vee n_0 \in \mathbf{Q} \vee n_0 \in \mathbf{Fa} \vee n_0 \in \mathbf{Fq})\} \cup \{(\sigma, n_1) \mid \sigma \in \Sigma \wedge L[[l]]\sigma = \text{False} \wedge n_1 = E[[a_1]]\sigma \wedge (n_1 \in \mathbf{Z} \vee n_1 \in \mathbf{R} \vee n_1 \in \mathbf{B} \vee n_1 \in \mathbf{A^*} \vee n_1 \in \mathbf{Seq} \vee n_1 \in \mathbf{Q} \vee n_1 \in \mathbf{Fa} \vee n_1 \in \mathbf{Fq})\}$

As expressões relacionais são utilizadas para representar a igualdade, diferença ou ordem de precedência entre valores. No caso de Strings (**A***), a ordem lexicográfica é utilizada como critério de comparação. A seguir, apresentamos as equações que representam essas expressões.

- $L[[e_0 == e_1]] = \{(\sigma, \text{True}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{B} \vee n_0, n_1 \in \mathbf{A^*} \vee n_0, n_1 \in \mathbf{Seq} \vee n_0, n_1 \in \mathbf{Q} \vee n_0, n_1 \in \mathbf{Fa} \vee n_0, n_1 \in \mathbf{Fq}) \wedge n_0 = n_1\} \cup \{(\sigma, \text{False}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{B} \vee n_0, n_1 \in \mathbf{A^*} \vee n_0, n_1 \in \mathbf{Seq} \vee n_0, n_1 \in \mathbf{Q} \vee n_0, n_1 \in \mathbf{Fa} \vee n_0, n_1 \in \mathbf{Fq}) \wedge n_0 \neq n_1\}$
- $L[[e_0 > e_1]] = \{(\sigma, \text{True}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0,n_1 \in \mathbf{Z} \vee n_0,n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{A^*}) \wedge n_0 > n_1\} \cup \{(\sigma, \text{False}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{A^*}) \wedge n_0 \leq n_1\}$
- $L[[e_0 < e_1]] = \{(\sigma, \text{True}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0,n_1 \in \mathbf{Z} \vee n_0,n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{A^*}) \wedge n_0 < n_1\} \cup \{(\sigma, \text{False}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{A^*}) \wedge n_0 \geq n_1\}$
- $L[[e_0 >= e_1]] = \{(\sigma, \text{True}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0,n_1 \in \mathbf{Z} \vee n_0,n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{A^*}) \wedge n_0 \geq n_1\} \cup \{(\sigma, \text{False}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{A^*}) \wedge n_0 < n_1\}$
- $L[[e_0 <= e_1]] = \{(\sigma, \text{True}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0,n_1 \in \mathbf{Z} \vee n_0,n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{A^*}) \wedge n_0 \leq n_1\} \cup \{(\sigma, \text{False}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{A^*}) \wedge n_0 > n_1\}$
- $L[[e_0 != e_1]] = \{(\sigma, \text{True}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge (n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{B} \vee n_0, n_1 \in \mathbf{A^*} \vee n_0, n_1 \in \mathbf{Seq} \vee n_0, n_1 \in \mathbf{Q} \vee n_0, n_1 \in \mathbf{Fa} \vee n_0, n_1 \in \mathbf{Fq}) \wedge n_0 \neq n_1\} \cup \{(\sigma, \text{False}) \mid \sigma \in \Sigma \wedge n_0 = E[[e_0]]\sigma, n_1 = E[[e_1]]\sigma \wedge$

$(n_0, n_1 \in \mathbf{Z} \vee n_0, n_1 \in \mathbf{R} \vee n_0, n_1 \in \mathbf{B} \vee n_0, n_1 \in \mathbf{A^*} \vee n_0, n_1 \in \mathbf{Seq} \vee n_0, n_1 \in \mathbf{Q} \vee n_0, n_1 \in \mathbf{Fa} \vee n_0, n_1 \in \mathbf{Fq}) \wedge n_0 = n_1\}$

As expressões lógicas são denotadas da seguinte forma:

- $L[[True]] = \{(\sigma,True) \mid \sigma \in \Sigma\}$
- $L[[False]] = \{(\sigma, False) \mid \sigma \in \Sigma\}$
- $L[[t]] = \{(\sigma, \sigma(t)) \mid \sigma \in \Sigma\}$
- $L[[b_0 \text{ AND } b_1]] = \{(\sigma, (n_0 \wedge n_1)) \mid \sigma \in \Sigma \wedge n_0 = E[[b_0]]\sigma, n_1 = E[[b_1]]\sigma \in \mathbf{B}\}$
- $L[[b0 \text{ OR } b_1]] = \{(\sigma, (n_0 \vee n_1)) \mid \sigma \in \Sigma \wedge n_0 = E[[b_0]]\sigma, n_1 = E[[b_1]]\sigma \in \mathbf{B}\}$
- $L[[\text{NOT } b_0]] = \{(\sigma, (\neg n_0)) \mid \sigma \in \Sigma \wedge n_0 = E[[b_0]]\sigma \in \mathbf{B}\}$

Expressões de Comandos

- $C[[i = e]] = \{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[n]]\sigma\}$
- $C[[c_0;c_1]] = C[[c1]] \circ C[[c0]]$
- $C[[\text{if } l \{c_0\} \text{ else } \{c_1\}]] = \{(\sigma, \sigma') \mid L[[l]]\sigma = True \wedge (\sigma, \sigma') \in C[[c_0]]\} \cup \{(\sigma, \sigma') \mid L[[l]]\sigma = False \wedge (\sigma, \sigma') \in C[[c_1]]\}$
- $C[[\text{if } l_0 \{c_0\} (\text{elif } l_i\{c_i\})^* \text{ else } \{c\}]] = \{(\sigma, \sigma') \mid L[[l_0]]\sigma = True \wedge (\sigma, \sigma') \in C[[c_0]]\} \cup \{(\sigma, \sigma') \mid L[[l_i]]\sigma = True \text{ ^ } n \in \mathbf{Z} \text{ ^ } 0 < i \leq n \wedge (\sigma, \sigma') \in C[[c_n]]\} \cup \{(\sigma, \sigma') \mid \forall i \; L[[l_i]]\sigma = False \text{ ^ } n \in \mathbf{Z} \text{ ^ } 0 < i \leq n \wedge (\sigma, \sigma') \in C[[c]]\}$
- $C[[\text{while } l \{c\}]] = \{(\sigma, \sigma') \mid L[[l]]\sigma = True \wedge (\sigma, \sigma') \in C[[\text{while } l \{c\}]] \vee C[[c]]\} \cup \{(\sigma, \sigma') \mid L[[l]]\sigma = False\}$

5.6 Erros Semânticos

Em expressões aritméticas, erros semânticos decorrem da incompatibilidade de tipos entre operandos, ou da divisão por zero, que é uma operação matematicamente indefinida. O operador + permite somente a concatenação apenas entre sequências do mesmo tipo (DNA com DNA, RNA com RNA, proteína com proteína). Nas expressões a seguir, os erros são descritos usando a notação $\perp_{\text{Tipo de erro}}$.

$(\sigma, (E[[a_0]] + E[[a_1]])), a_0, a_1 \in \mathbf{Z} \vee a_0, a_1 \in \mathbf{R}$;

- $E[[a_0 + a_1]] =$ $(\sigma, (E[[a_0 a_1]])), a_0, a_1 \in \mathbf{A^*} \vee a_0, a_1 \in \mathbf{dSeq} \vee a_0, a_1 \in \mathbf{rSeq} \vee a_0, a_1 \in \mathbf{pSeq}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- $E[[a_0 - a_1]] =$ $(\sigma, (E[[a_0]] - E[[a_1]])), a_0, a_1 \in \mathbf{Z} \vee a_0, a_1 \in \mathbf{R}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- $E[[a_0 * a_1]] =$ $(\sigma, (E[[a_0]] * E[[a_1]])), a_0, a_1 \in \mathbf{Z} \vee a_0, a_1 \in \mathbf{R}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- $E[[a_0 / a_1]] =$ $(\sigma, (E[[a_0]] / E[[a_1]])), (a_0, a_1 \in \mathbf{Z} \vee a_0, a_1 \in \mathbf{R}) \wedge a_1 \neq 0$;

  $\perp_{\text{Erro de divisão por zero}}$, $a_1 = 0$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

Em expressões relacionais, é possível realizar a comparação de igualdade (==) ou diferença (!=) entre tipos do mesmo domínio. Para tipos numéricos (inteiro ou real) e strings, é possível realizar comparações de ordem (<, >, <= ou >=):

- $E[[e_0 == e_1]] =$ $(\sigma, (E[[e_0]] == E[[e_1]])), e_0, e_1 \in \mathbf{Z} \vee e_0, e_1 \in \mathbf{R} \vee e_0, e_1 \in \mathbf{A^*} \vee e_0, e_1 \in \mathbf{B} \vee e_0, e_1 \in \mathbf{Seq} \vee e_0, e_1 \in \mathbf{R} \vee e_0, e_1 \in \mathbf{Q} \vee e_0, e_1 \in \mathbf{Fa} \vee e_0, e_1 \in \mathbf{Fq}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- $E[[e_0 \mathrel{!}= e_1]] =$ $(\sigma, (E[[e_0]] \mathrel{!}= E[[e_1]])), e_0, e_1 \in \mathbf{Z} \vee e_0, e_1 \in \mathbf{R} \vee e_0, e_1 \in \mathbf{A^*} \vee e_0, e_1 \in \mathbf{B} \vee e_0, e_1 \in \mathbf{Seq} \vee e_0, e_1 \in \mathbf{R} \vee e_0, e_1 \in \mathbf{Q} \vee e_0, e_1 \in \mathbf{Fa} \vee e_0, e_1 \in \mathbf{Fq}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- $E[[e_0 < e_1]] =$ $(\sigma, (E[[e_0]] < E[[e_1]])), e_0, e_1 \in \mathbf{Z} \vee e_0, e_1 \in \mathbf{R} \vee e_0, e_1 \in \mathbf{A^*}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- $E[[e_0 > e_1]] =$ $(\sigma, (E[[e_0]] > E[[e_1]])), e_0, e_1 \in \mathbf{Z} \vee e_0, e_1 \in \mathbf{R} \vee e_0, e_1 \in \mathbf{A^*}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- $E[[e_0 <= e_1]] =$ $(\sigma, (E[[e_0]] <= E[[e_1]])), e_0, e_1 \in \mathbf{Z} \vee e_0, e_1 \in \mathbf{R} \vee e_0, e_1 \in \mathbf{A^*}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- $E[[e_0 >= e_1]] =$ $(\sigma, (E[[e_0]] >= E[[e_1]])), e_0, e_1 \in \mathbf{Z} \vee e_0, e_1 \in \mathbf{R} \vee e_0, e_1 \in \mathbf{A^*}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

Em expressões lógicas (AND, OR, NOT), os operandos devem ser necessariamente do tipo booleano, assim como o primeiro operando de uma expressão ternária:

- $E[[b_0 \text{ AND } b_1]] =$ $(\sigma, (E[[e_0]] \wedge E[[e_1]])), e_0, e_1 \in \mathbf{B}$;

$\perp_{\text{Erro de domínio}}$, caso contrário.

- E[[b_0 OR b_1]] = $(\sigma, (E[[e_0]] \vee E[[e_1]]))$, $e_0, e_1 \in \mathbf{B}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- E[[NOT b_0]] = $(\sigma, (\neg E[[e_0]]))$, $e_0, e_1 \in \mathbf{B}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- E[[if l then a_0 else a_1]] = $(\sigma, (E[[a_0]]))$, $l \in \mathbf{B}$, l = True, $a_0, a_1 \in \mathbf{T}$;
  $(\sigma, (E[[a_1]]))$, $l \in \mathbf{B}$, l = False $a_0, a_1 \in \mathbf{T}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

Em expressões de atribuição, o identificador à esquerda que recebe a atribuição e a expressão atribuída à direita devem ter tipos do mesmo domínio:

- C[[Integer i = e]] = $\{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[e]]\sigma\}$, $e \in \mathbf{Z}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- C[[Real i = e]] = $\{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[e]]\sigma\}$, $e \in \mathbf{R}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- C[[Boolean i = e]] = $\{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[e]]\sigma\}$, $e \in \mathbf{B}$;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- C[[String i = e]] = $\{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[e]]\sigma\}$, e $\in$ **A***;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- C[[Sequence i = e]] = $\{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[e]]\sigma\}$, e $\in$ **Seq**;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- C[[Quality i = e]] = $\{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[e]]\sigma\}$, e $\in$ **Q**;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- C[[FASTA i = e]] = $\{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[e]]\sigma\}$, e $\in$ **Fa**;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

- C[[FASTQ i = e]] = $\{(\sigma, \sigma(n/i)) \mid \sigma \in \Sigma \wedge n = E[[e]]\sigma\}$, e $\in$ **Fq**;

  $\perp_{\text{Erro de domínio}}$, caso contrário.

6 IMPLEMENTAÇÃO

A implementação do interpretador da linguagem Rosa está disponível no GitHub[1]. Devido ao tempo, nem todas as funcionalidades descritas nas seções anteriores foram implementadas. As seções seguintes descrevem a estrutura dos arquivos no repositório e as funcionalidades que estão implementadas até o momento.

6.1 Estrutura dos arquivos

O interpretador foi desenvolvido na linguagem SML, com a utilização das bibliotecas ML-Lex e ML-Yacc e foi baseado no exemplo da calculadora, disponibilizado no manual da biblioteca ML-Yacc[2]. Os principais arquivos que compõem o interpretador são:

[1] https://github.com/nahimsouza/rosa-lang
[2] https://www.smlnj.org/doc/ML-Yacc/mlyacc007.html

- **rosa.lex** - contém as definições da parte léxica da gramática, baseada na EBNF. A partir deste arquivo a biblioteca ML-Lex consegue identificar os tokens que compõem o programa.
- **rosa.grm** - contém a definição das expressões, relativas à sintaxe e semântica, que serão avaliadas pela ML-Yacc e executadas posteriormente. Além disso, também foram definidas no início do arquivo, funções que auxiliam na avaliação dessas expressões.
- **rosa.sml** - ponto de entrada do programa, responsável pela leitura de dados e exibição dos resultados finais.
- **functions.sml** - contém a implementação das funções principais do programa escrito na linguagem Rosa. São utilizadas para manipulação de sequências genéticas
- **examples/success1.rosa** - Exemplo básico de arquivo em linguagem Rosa, contendo operações que são suportadas atualmente na linguagem.
- **examples/sample.rosa** - Exemplo mais realista da utilização da linguagem Rosa. *Algumas operações deste arquivo ainda não são suportadas*.
- **README.md** - Instruções para execução do interpretador em SML
- **.gitignore** - Arquivo de configuração do git para ignorar os arquivos que são gerados automaticamente durante a execução do interpretador.

6.2 Instruções para uso do interpretador

Para executar o interpretador, é necessário ter um interpretador de SML instalado (por exemplo, SML/NJ[3]). Seguem as instruções para execução:

- Abrir o terminal `sml` na pasta raiz do interpretador
- Compilar os arquivos fonte do interpretador usando o Compilation Manager (CM): `CM.make("sources.cm");`
- Caso a compilação ocorra com sucesso, o terminal deve mostrar a mensagem `val it = true : bool`
- Nesse caso, o interpretador pode ser executado inserindo o comando `Rosa.parse();`. Assim, o terminal passa a executar interativamente comandos da linguagem ROSA separados por ponto-e-vírgula.

[3] https://www.smlnj.org

6.3 Funcionalidades do interpretador

As principais funcionalidades implementadas são:

- Operações aritméticas (+, -, *, /) entre números inteiros e reais
- Atribuição de variáveis
- Operações de manipulação de sequências:
  - Concatenação e inversão de sequências
  - Funções de manipulação descritas nas seções anteriores: ins, del, point, transcribe, translate, complement, motif
- Operações booleanas (AND, OR, NOT)
- Operações relacionais (==, !=, <, >, <=, >=)

6.4 Problemas encontrados

Durante a implementação, encontramos algumas ambiguidades na gramática, na definição dos tipos Sequence, Quality, FASTA e FASTQ. Conseguimos solucionar esse problemas para variáveis do tipo Sequence diretamente na definição adicionada no arquivo rosa.lex.

No entanto, para os tipos Quality, FASTA e FASTQ, embora tenhamos implementado parte da tokenização, não conseguimos solucionar a ambiguidade causada pela utilização de símbolos e caracteres especiais. Devido a isso, também não foi possível implementar a manipulação de arquivos contendo dados do tipo FASTA e FASTQ.

Outras funcionalidades não concluídas foram:

- Implementação do tipo recursivo List
- Implementação de comandos condicionais
- Implementação de funções definidas pelo usuário
- Implementação de estrutura de repetição (while)

7 REFERÊNCIAS

COCK, P. J. A. et al. Biopython: freely available Python tools for computational molecular biology and bioinformatics. **Bioinformatics**, v. 25, n. 11, p. 1422–1423, 1 Jun. 2009.

PEARSON, W. R.; LIPMAN, D. J. Improved tools for biological sequence comparison. **Proceedings of the National Academy of Sciences of the United States of America**, v. 85, n. 8, p. 2444–2448, Apr. 1988.

KAMATH, U.; DE JONG, K.; SHEHU, A. Effective automated feature construction and selection for classification of biological sequences. **Plos One**, v. 9, n. 7, p. e99982, 17 Jul. 2014.

COCK, P. J. A. et al. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. **Nucleic Acids Research**, v. 38, n. 6, p. 1767–1771, Apr. 2010.