(DNA#): Molecular Biology Computation Language Reference Manual

Aalhad Patankar, Min Fan, Nan Yu, Oriana Fuentes, Stan Peceny

{ap3536, mf3084, ny2263, oif2102, skp2140} @columbia.edu

1 Introduction

DNA# is a language designed to provide a platform and means for computation of genomic data, a rising area in the field of bioinformatics. This programming language is inspired by Biopython, a Python library providing data structures and methods for dealing directly with bioinformatics processes. However, our language is a more general version of Biopython geared specifically for the manipulation of genetic information natively, as is reflected in the language's native data types and syntax. The basic data unit of our language is the nucleotide, and the language provides data structures for higher levels of genetic modeling (e.g. DNA sequence, amino acid) composed of the fundamental nucleotide unit. Our language also provides methods for basic file I/O, and a method to interface with several types of files wherein genetic data is often stored.

1.1 Motivation

Inspired by the parallelism between genetic code and computer code, we would like to provide a platform to "code" genes easily and natively. We are implementing basic models of molecular biology in light of heightened interest in understanding genetics and its potential impact on tailoring medicine, understanding diseases and ultimately improving human life. We are designing DNA# for both the novice user, who is interested in learning the basics of genetic code, and the advanced researcher performing analyses on large data sequences. We would like to rethink genetic code as a form of data representation itself, and provide coders a platform to tinker with genes and clearly see the biological results without hours of laborious manual transcription, complement finding, and referencing external resources. In sum, we would like to create a language for programmers to code in the genetic language and learn synthetic biology, and for synthetic biologists with limited programming experience to open source and optimize their work.

1.2 Summary of Goals

- Provide basic, intermediate and advanced genetic operations that estimate physical properties and mimic real genetic processes, including but not limited to transcription (DNA->RNA) and translation (mRNA-> protein)
- Support primitive and complex data structures that can handle simple base sequences to full blown genetic maps

- Provide means to allow scientists to add physical properties (e.g. bond strength, annealing temperature) to existing data structures (nucleotide, codon, amino acid, etc.)
- Allow users to input and output files commonly used to store genetic data (e.g. FASTA) and convert data from such files into mutable native data structures
- Allow users to build higher level algorithms (e.g. finding optimum primer regions, sequence alignment) to model molecular biology and calculate optimal sub-sequences in DNA to perform operations such as designing primers for polymerase chain reaction (PCR), a basic genetic tool.

2 Lexical Conventions

2.1 Identifiers

The identifier rule inherits from the conventional C/C++ identifier rule and can be any string of letters, digits, and underscores, however, not beginning with a digit.

The regular expression for the identifiers is the following:

```
[‘a-z’ ’A-Z’ ‘_’][‘a-z’ ‘A-Z’ ‘_’ ‘0-9’]*
```

2.2 Keywords

DNA# reserves keywords specified in the table below. The keywords in the table cannot be used as identifiers.

true	false	if	else
elseif	for	while	continue
break	include	end	local
then	return	void	nuc
int	double	char	bool
aa	dna	rna	codon
peptide	dnatorna	rnatodna	

2.3 Data Types

The following keywords start the definition of various types

-Standard types

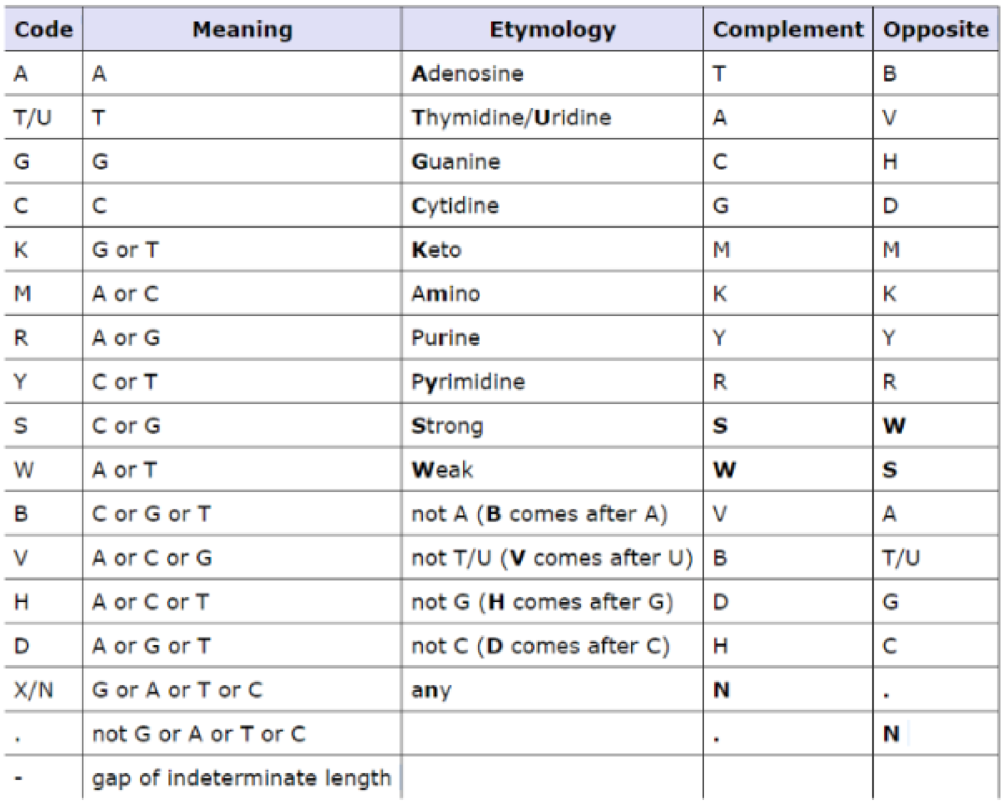
Type	Definition	Values
bool	Boolean	true, false
int	Integer	integers
double	Double Floating Point	real numbers
void	Valueless	no values
char	Character	Numbers using ASCII encoding
string	Sequence of characters	Char

-Primitive

Type	Definition	Values
nuc	Individual nucleotides and variable nucleotides	A, T, G, C, U K, M, R, Y, S, W, B, V, H, D, X, N

The relation between the different values of nuc and their complementary, transcribed and translated counterparts can be seen in Figure 1.

Figure 1



Code	Meaning	Etymology	Complement	Opposite
A	A	**A**denosine	T	B
T/U	T	**T**hymidine/**U**ridine	A	V
G	G	**G**uanine	C	H
C	C	**C**ytidine	G	D
K	G or T	**K**eto	M	M
M	A or C	A**m**ino	K	K
R	A or G	Pu**r**ine	Y	Y
Y	C or T	P**y**rimidine	R	R
S	C or G	**S**trong	**S**	**W**
W	A or T	**W**eak	**W**	**S**
B	C or G or T	not A (**B** comes after A)	V	A
V	A or C or G	not T/U (**V** comes after U)	B	T/U
H	A or C or T	not G (**H** comes after G)	D	G
D	A or G or T	not C (**D** comes after C)	H	C
X/N	G or A or T or C	any	**N**	.
.	not G or A or T or C		.	**N**
-	gap of indeterminate length			

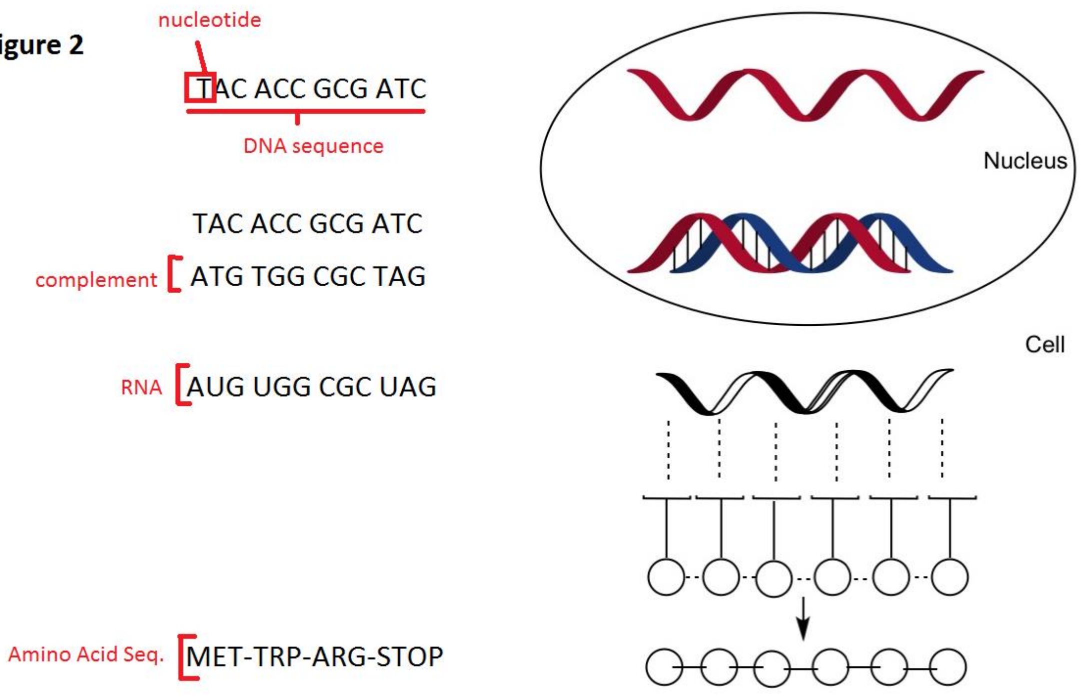
source: http://www.boekhoff.info/?pid=data&dat=fasta-codes

-Complex

Type	Definition/Value	Sample Values
dna	A Sequence of A,T, G, C Deoxynucleotides	AGTXWRCC
rna	A Sequence of A, U, G, C Nucleotides	AGUCC
codon	A three-nucleotide RNA sequence specifying a single amino acid	UGU, CGA, ACC, e.t.c
aa	Amino acid, basic chemical structures composing a protein	Ala, Trp, Cys, e.t.c
peptide	Sequence of amino acids (aas)	AlaTrpCys
array	An array list of values of the same datatype (0-index based)	[1, 4, 6, 3] ['A', 'T', 'G', 'T']

Figure 2 displays the relation between several of the complex data structures on a biological level.

Figure 2



2.4 Control Flow

Control Flow	Definition
if	Initialize a conditional if statement
then	Implements some functionality given that the if conditional is true
else	Initialize else clause in if statement if the initial if clause is deemed incorrect
elseif	Initialize else clause in if statement if the initial if clause is deemed incorrect and there is another condition to be satisfied
for	Initialize for loop to implement certain functionality a set number of times
while	Initialize while loop to implement certain functionality while a conditional following the 'while' keyword is correct
end	End is used at the end of if, for, and while statement to indicate the end of the clause

- if statement

A basic if statement is shown below:

```
if cond==true then
 …
end
```

Should there be more than one condition to be decided, use elseif:

```
if cond1==true then
 …
elseif cond2==true then
 …
else
 …
end
```

What need to be noticed is that here cond, cond1 and cond2 must be Logical Expressions, which are expressions whose value is either 'true' or 'false'. For more detailed information please read part 3.3.

- for statement

A basic for loop is shown below, which starts with a keyword 'for' and ends with an 'end'. Inside the for statement, 'start_num','end_num' and 'step_length' must be valid integers. And the loop variable (named 'i' in the example) can be any variable with a type of int, and it has to be pre-defined before use in the for loop.The loop runs from 'start_num' to 'end_num' with an increment of 'step_length'. If the step_length is 1, it can be omitted.

```
for i=start_num,end_num,step_length
 …
end
```

- while statement

Here is a basic example of while statement. It starts with a keyword 'while' and ends with an 'end'. Please notice that the keyword 'end' cannot be omitted for it is a common mistake users may make. Inside the while loop, cond must be a Logical Expression (For more detail about Logical Expression please read part 3.3). The loop runs as long as the given condition is true.

```
while cond==true
 …
end
```

2.5 Built in functions & Syntax

Function	Definition	Example Use
dnatorna	Cast type from DNA to RNA	(dnatorna) dna_name
rnatodna	Cast type from RNA to DNA	(rnatodna) rna_name
len	Returns the length of the genetic sequence, accepts any complex data type	Int len(dna), int len(rna), int len(peptide), int len(codon), int len(array), int len(aa)

2.6 Functions

Same as functions in C or C++, all functions have explicit return type and input table, which may look like:

```
return_type function_name(input_a_type A,input_b_type B)
      …
end
```

When declaring a function, the user must designate a ‘return_type’ of the function. The ‘return_type’ can be any supported data type of DNA#, including standard types, primitive and complex types. For more detail information about types please check part 2.3.

For the input list, the types of the input can be any supported data type supported by DNA#. And the number of inputs can be any number depends on the user's’ demand. DNA# also supports functions without any input, but the brackets ‘()’ can’t be omitted. Here is one example:

```
return_type function_name()
      …
end
```

Our language also supports Local functions:

```
local return_type function_name(input_a_type A,input_b_type B)
      …
end
```

2.7 Literals

Nuc	All the symbols of bases, including the 5 basic types and the variable types
AA	Amino acids are sequences of DNA
Integer	Define a decimal digit with the following regular expression: **digit = [’0’ - ’9’]** An int is a 32-bit signed integer, and consists of at least one digit **digit+**
Double	A double is a 64-bit floating point number. Define the regular expression of the exponential part as follows: **exp = ’e’ [’+’ ’-’]? [’0’-’9’]+** If the decimal point is present, at least one of the the integer and fractional parts must also be present –the compiler interprets an absent part as 0. If there is no decimal point, the integer part and the exponent must be present: **((digit+ ’.’ digit* | ’.’ digit+) exp?) | (digit+ exp)**
Bool	The boolean type has two predefined constants for each truth value, and no other values: **True | false**
Characters	Characters are single, 8-bit, ASCII characters
Strings	Sequence of characters, designated by “ “ or ‘ ‘

3. Syntax

3.1 Program structure

At the highest level, DNA# is a scripting language. Every program written in DNA# is compiled line by line. Therefore, the most important subdivision of a program is scopes. A DNA# program typically has two different scopes; one is line-scoping, denoted by end-of-line character, and the other is scoping blocks, denoted by a keyword-pair "begin end". All variables and functions must be declared before their usage. Regarding variable scoping, there are no global variables in DNA#. Every single .dna file is a scope, denoted by its filename, and inside the file there could be more local scopes, following the rule that variables in the most nested scopes have the highest overriding priority, followed by function formals if they are nested in a function.

```
Program:
    variable declarations (single lines of codes)
        function declarations (block scoping of codes)
        flow control (block of codes)
        block of codes denoted with 'begin' and 'end'
End-of-file
```

3.2 Variable declaration

Variables can be declared and initialized in a scope, be it inside a function, control flow statement or a block of code, and will be local to that scope. However, we enforce that every variable must be initialized upon declaration. The follow structure is used for variable declaration.

data-type variable-id = initial-value semi-colon

'data-type' is one of the type tokens indicating what type the newly declared variable belongs to. 'variable-id' can be a newly identifier token. 'initial-value' can be either a literal of the corresponding data-type or it can be an expression, which is evaluated to the corresponding data-type.

Multiple assignments in one line is allowed in the following way. 'variable-id1' and 'variable-id2' and the following 'variable-id's will have the same 'data-type' specified and also have the same initial value evaluated from the 'initial-value' expression or literal.

data-type variable-id1 = variable-id2 = … = initial-value semi-colon

The declaration, including single ones and multiple ones, ends with a semi-colon. The expressions used in the assignment will be detailed in the next section.
A special case for variable declaration is the declaration of arrays. The following syntax rule is used for array declaration.

data-type[] array-id = data-type[int-variable] semi-colon

The declaration of arrays does not support multiple declarations in a single. It supports the initialization of the array as the following. Please note, for the initialization of the array, the data-type of the passed must match the data-type specified in the declaration and the length must also match with the length specified in the declaration.

array-id = [data-type-expression1, data-type-expression2, …]

3.3 Expressions

- Primary expressions

Primary expressions will be comprised of the following two categories, as the terminals in the grammar.

1. Literals
   - The primitive literals mentioned in the *Lexical Convention* section have the following corresponding grammar calls, *INT_LIT*, *DOUBLE_LIT*, *BOOL_LIT*, and *CHAR_LIT*.
   - Character arrays of the token *STRING*
   - Character arrays of the token, starting with '#' and ending with '#' *SEQUENCE*
   - Identifiers of the token *ID*
2. Parenthesized expressions

   (expression)

- Arithmetic expressions

1. Expressions with binary operators

   numeric_expr bop numeric_expr

'numeric_expr' stands for all the expressions including the literals,which are evaluated to be either the data-type of int or double. However, the data-types of two 'numeric_expr's on the both sides of the 'bop' must be the same. And the final data-type the expression is evaluated to, is the same data-type.
The bop involved in the arithmetic expressions with binary operators can only one of the following.

bop	meaning
+	plus

-	minus
*	multiply
/	divide
%	modulo

2. Expressions with unary operators

- Expressions with left-associative unary operators

  uop numeric_expr

  ^exponential

In DNA#, there is only one left-associative unary operator, which is for the exponential calculation.

- Expressions with right-associative unary operators

  numeric_expr uop

  - flip the sign

In DNA#, there is only one right-associative unary operator, which is for the sign-flipping.

- Logical expressions

1. Expressions with binary operators

   logical_expr bop logical_expr

'logical_expr' stands for all the expressions including the literals,which are evaluated to be the data-type of bool. And the final data-type the expression is evaluated to, is the data-type of bool. The bop involved in the arithmetic expressions with binary operators can only one of the following.

bop	meaning
|	or
&	and

The truth table for the bool evaluation follows the convention of many programming languages. The | operator evaluates to false only when both sides are false and otherwise true. The & operator evaluates to true only when both sides are true and otherwise false.

2. Expressions with unary operators

   uop logical_expr

There is only one unary operator related to logical expression, which is the right-associative negation with following syntax. The symbol for this 'uop' is '!'.

!	negation

-Relational expressions

All the relational expressions have binary operators and the following syntax.

   expr bop expr

The expressions on either sides of the binary operator 'bop' must have to be comparable. In DNA#, the numeric values can be compared with other numeric values. And strings are comparable to other strings based on their lexical alphabetical order. The 'bop's are listed below with their meanings.

bop	meaning
<	less than
<=	less than or equal to
>	larger than
>=	larger than or equal to
==	equal to
!=	not equal to

-Biological Sequence Expressions

1. Expressions with binary operators

seq_expr bop seq_expr

'seq_expr' stands for all the biological sequence expressions including DNA sequences, RNA sequences, and Peptide data-types. There is only one 'bop' for this type of expressions which is concatenation, denoted as '^'.

^	concatenation

2. Expressions with unary operators

- Expressions with left-associative unary operators

uop numeric_expr

In DNA#, there is only one left-associative unary operator, which is for the complementing a biological sequence.

@	complement

- Expressions with right-associative unary operators

  numeric_expr uop

In DNA#, there are three right-associative unary operator. The first one is used for transcription ('->'), the second is for translation ('+>') from RNA sequence (mRNA) to another RNA sequence (a sequence of codons) and the last for translation2 ('%>') from a RNA sequence (a sequence of codons) to a Peptide. In this sense, the transcription operator can only be applied to DNA sequences and translation operator can only be applied to RNA sequences.

->	transcribe
+>	translate
%>	translate2

For all the unary operators on the biological sequences (including both left-associative and right-associative ones), when they are applied to a sequence, it returns another biological sequence as its meaning of the expression.

- ***Function Calls***
  - Function calls are also expressions that can be reduced to the return type of the function

3.4 Precedence of operators

The following precedence of operators used in DNA# is partially referred to C programming language.

High	(	)				
	^	!	@			
	+>	->				
	*	/	%			
| V	+	-				
	<	<=	==	>=	>	==
	&					
	|					
Low	=					

3.5 Associativity of operators

The following operators are left-associative:

‘()’ ^ +> -> * / % + - < <= == >= > == & |

The following are right-associative:

= @ !

3.6 Statements

-Assignments

statement:

defined-variable = expression semicolon

The assignment statements follow the previous rules used in the variable declaration section of exprprimary expressions. Left side of the assignment operator must be a defined variable, which is a valid identifier token. If it is a new declaration, it falls back to the declaration expression rules. The expression on the right side of the assignment operator must be an expression evaluated to the same data-type of the defined variable. The expression itself could be another assignment statement, meaning the whole statement is a line of multiple statements. The assignment is ended with a semicolon.
A special case for the assignment statements is designed for arrays. It has the following syntax rule.

```
array assignment statement:
    defined-array-id = defined-array-id2
    defiend-array[int-expression] = data-type-expression
```

-Function

A function is defined as a statement that abides by the following:

- takes a set of arguments, which are statements that are variable declarations with the following exception: these declarations do not have to be initialized and cannot be terminated by comma (e.g. int hi is valid)
- body is its own block of code as defined below, with the restriction that has to return according to below rule
- an expression must be followed after return keyword that can be evaluated to be the same type as specified by function declaration

- Syntax defined in later section

-Blocks and control flow

1. Blocks:
A block is defined inside the space between keywords ‘begin’ and ‘end’. Blocks can contain zero or more statements inside them.

```
begin
    …
end
```

2. Branching statements:
DNA# supports both the if and if-else branching statements with the following syntax rules. Every branching statement starts with a ‘begin’ and an ‘end’.
if branching statement:

```
if bool-expr begin
```

```
        …
    end
```

if-else branching statement:

```
    if bool-expr begin
        …
    end
    else begin
        ...
    end
```

Please note, that using these syntax rules, DNA# automatically takes care of the “dangling else” problem. In the case, “if bool-expr begin ... end if bool-expr begin … end else begin … end”, it is very obvious that the else will stick with the if immediately before it, since the first if statement was finished earlier.

3. Iteration statements:

DNA# supports two types of iteration statements. One is the regular for-loop statements, and the other is the while-loop statements.

for-loop statement:

```
    for(begin-expr_optional; bool-expr; increment-expr_optional) begin
        …
    end
```

while-loop statment:

```
    while(bool-expr_optional) begin
        …
    end
```

4. Jump statements:

As with other well-designed programming languages, DNA# does not support goto statements. The following control-jumping statements are supported.

function-return statement:

```
    return expression_optional semi-colon
```

The previous statement returns control to the code that calls the function.

break-statement:

```
    break semi-colon
```

The break-statements are used inside for-loop and while-loop to return control earlier than the normal end of iterations.

continue statement:

```
    continue semi-colon
```

The continue-statements are used inside for-loop and while-loops to skip current iteration and go to next iteration.

5. Function-call statement:

The following calls a previously defined function func.

```
funct(argument1, argument2, … ) semi-colon
```

The function-call statements use the conventional C-style function calls.

4. Scoping rules

DNA# uses the following scoping rules for variables.

a. Lexical scoping with blocks

*** you can scope anything with begin end, but functions and control flow has to include these features

b. Function closure

c. Assignment and parameter passing

5. Sample Program

- **Program 1:**
  Here is a basic program demonstrating several components (variables, control flow, functions) being used in a script. It performs the basic function of reading a DNA code, and outputting the peptide sequence that results from this sequence. NOTE: we intentionally picked the sequence not to contain any variable nucleotides. A more useful, and complex algorithm, would take a DNA sequence and output a list of possible peptides that could be generated from the sequence, along with properties such as weight and GC content of each sequence.

File BioExpData.dat

```
<sample1>
TCCCCAATGAAGGGTGCTTAGTAC
<\sample1>
```

File DNA2Protein.dnas

```
include “io.dnas”
include “basicBio.dnas”



(* Import the file with DNA sequences *)
DNA SampleA = import_dna("BioExpData.dat");
(*SampleA = TCCCCAATGAAGGGTGCTTAGTAC*)



DNA SampleB = @SampleA;
(*SampleB = AGGGGTTACTTCCCACGAATCATG*)

codon [] codonList = +>(->SampleB)
(* uses the unary operator for transcription (->) and translation (+>) to get a list of codons
corresponding to the DNA
codonList = UCC-CCA-AUG(s)-AAG-GGU-GCU-UAG(e)-UAC*)

(* defining a function findStartCodon, which will return index of start codon *)
int findStartCodon (codon [] cod)
      for i=0, i < len(cod), i++
          if cod[i] == AUG (* the start codon *)
              return i;
          end
       end
end

(* defining a function findEndCodon, which will return index of end codon *)
int findEndCodon (codon [] cod)
      for i=0, i < len(cod), i++
          if cod[i] == UAC (* the end codon *)
              return i;
          end
       end
end

(* call function findStartCodon, which will return an int, to b *)
int i = findStartCodon(codonList) ;
int j = findEndCodon(codonList) ;

(* i=3
* j=7
* codonList=AUG(s)-AAG-GGU-GCU-UAG(e)   *)

(* function cutOut, which will select a subset of codonList that falls between start and end
codon, will be replaced by inbuilt array function later *)

codon[] cutOut(codon [] clist, int start, int end)
       codon[] newList = [];
       int iterator = 0;
       for i=0, i< len(clist), i++
             if (i > start && i < end)
                   newList[iterator] = clist[i];
                   iterator = iterator + 1;
            end
      end
end

if ( i!=-1 && j!=-1 && i < j )
     codonList = cutOut(,i,j);
else
     codonList = null;
end

aa result1 = %> codonList

print("Below is the result:\n")
print_list(result1)
```

Running Result
Below is the result:

```
(origin)
->methionine
    ->lysine
    ->glycine
    ->alanine
     ->(terminal)
```

References:

[1] Funk Programming Language - 2012

[2] DNA# Project Proposal   - 2016