(DNA#): Molecular Biology Computation Language Reference Manual

Aalhad Patankar, Min Fan, Nan Yu, Oriana Fuentes, Stan Peceny
{ap3536, mf3084, ny2263, oif2102, skp2140} @columbia.edu
December 20th, 2016

Contents

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. Further, DNA# includes operations such as transcribe and translate which can be performed on these native data types. Our language also provides methods for basic file input, 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

1.2.1 Functional

- Provide basic, intermediate and advanced genetic operations that 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
- 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.

1.2.2 Familiarity

The syntax for DNA# is highly similar to and inspired from C++ and MATLAB. The similarities with MATLAB arise from the fact that few parentheses or braces are required, replaced instead by keywords that indicate scoping. As a consequence, the code is less cluttered and enables for easier debugging, particularly for less experienced programmers. Since the language is not meant for computer scientists, the aim was to make the syntax simple to comprehend and natural to implement. Although DNA# does not force indentation, improper indenting can result in bugs difficult to discover, and thus DNA# prefers an option where statements can be explicitly closed.

1.2.3 Domain Specific

The idea was to make DNA# focus and do well on a single task, handling common genetic operations. The language is not meant to be broad for general purpose programming and is directed at a particular domain. While the language is able to express arithmetic operations as well as many common algorithms, DNA#'s focus is specifically meant for operations on nucleotide sequences.

1.2.4 Ease of Use

DNA# was designed to be comfortable for use. The goal was to create a language for biologists, and hence it was essential to make a high-level language wherein the user does not have to think about difficulties associated with low-level programming such as memory management, pointer manipulation and complex data structure design. As a consequence, DNA# does not support dynamic memory allocation, classes and multi-dimensional arrays.

1.3 Related Work

As many modern languages, DNA#'s low level implementation stems from C. Hence, the precedence and associativity rules are very similar, however, additional operators are implemented. Furthermore, DNA# is procedural, the scope is static, and function parameters are evaluated in applicative order. On the other hand, DNA# has no main function and globals do not exist.

There is a similar work implemented as a library in Python called BioPython which offers freely available tools for biological computation. The project is part of an effort striving for addressing the needs of current as well as future work in bioinformatics.

The hope of the DNA# project is to create means from the beginning to address the needs of biologists and chemists without having to built on languages not intended for these purposes.

2 Tutorial

2.1 Compiling DNA#

The instructions below were tested on the following system:

```
stan@Australia1:~/PTLFall16$ uname -a
```

```
Linux Australia1 4.4.0-53-generic #74-Ubuntu SMP Fri Dec 2
15:59:10 UTC 2016 x86_64 x86_64 x86_64 GNU/Linux
```

Start with cloning the repository into your folder of choice:

```
git clone https://github.com/PLT-Gurus/PTLFall16.git
```

As DNA# is compiled into LLVM Intermediate representation, one first needs to install the OCaml llvm library, which can be done in a terminal through opam.

Install LLVM and its development libraries, the m4 macro preprocessor, and opam, then use opam to install llvm.

Furthermore, install clang compiler for the c built-in functions inside the compiler. Type the following line into the shell:

```
sudo apt-get install clang
```

Access the DNA# folder and type ‘make’. This compiles DNA# and creates an executable ./runDNAs.sh

To run your programs use the following command and replace filename with the actual file name:

```
./runDNAs.sh -r filename.dnas
```

If the terminal shows "can't find command", modify permissions by typing "chmod +x runDNAs.sh" into the terminal.

To run all the test files, run the following command:

```
./test_DNAs.sh
```

In order to clean up all the executables and other built files, run ‘make clean’ in the terminal.

Print “Hello world!” in DNA#

2.2 Writing Programs

2.2.1 Declaring Variables

Variables can be declared outside of functions, within functions and in function definitions as formals. The syntax for declaring variables is as follows: type variable_id = value, where type is a valid data type supported by the language, variable_id is the unique identifier for the variable and value is an expression that reduces to a value the variable can hold (Note: you can declare a variable *b* with a previously declared variable *a*’s id, but the previous variable *a* will be overridden and inaccessible) . The variable must be initialized with a valid value for the assigned data type, except in the function declaration formals. Additionally, variable identifiers must begin with a letter or underscore character.

Standard Data Types

Below are examples of declarations of standard data types, note that doubles should be initialized with a decimal:

```
bool a = true;
int b = 4;
double c = 5.7;
char d = ‘k’;
string e = “hello”;
```

void types can only be assigned to functions.

Primitive Data Types

DNA#’s primitive data types are the nucleotides and amino acids, which is stated using the datatype ‘nuc’ for a nucleotide or ‘aa’ for amino acid type .

The nuc data type only accepts the five bases A, C, G, T, U (either lowercase or uppercase). It is declared as follows:

```
nuc a = #G;
```

The amino acid data type accepts A, C, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, U,V, W, Y

```
aa = ‘T’;
```

Value	Amino Acid
A	alanine
B	terminal
C	cysteine
D	aspartic
E	glutamic

F	phenylalanine
G	glycine
H	histidine
I	isoleucine
K	lysine
L	leucine
M	methionine
P	proline
Q	glutamine
R	arginine
S	serine
T	threonine
V	valine
W	tryptophan
Y	tyrosine

Complex Data Types

Complex data types include DNA, RNA, amino acid, peptide and arrays. Arrays are indexable collections of items of the same data type. DNA, RNA and codons are built as arrays of nucleotides. The nucleotides can be accessed the same way an element of an array is accessed. Arrays can be built out of any of the standard, primitive and complex data types, except other arrays (The exception is that you can construct arrays of DNA, RNA and peptides). Arrays cannot be initialized upon declaration, but all other complex data types must be initialized upon declaration.They are declared by stating the type of the elements in the array followed by the number of elements and then the identifier. Examples of valid declarations are:

```
int [5] myarray;
DNA a = #ATGCCGGG;
RNA b = #ATTGGGGCCUU;
aa = #GGATCCGAGTCAGTGTATCGTCGTCGTGTT;
```

```
peptide = #GCTAGTCAGTACGTCGTACTGAGCTCTAG;
```

2.2.2 Writing Expressions

Primary

Primary expressions include literals and primary expressions. There are literals for integer, boolean, char, DNA, RNA, and peptide.

Arithmetic

DNA# includes standard arithmetic operations addition, subtraction, multiplication, division, modulo and exponentiation.

Logical

Logical expressions include and (&), or (|) and negation (!). These are used for operations as well as for logical expressions in control flow. Use examples of logical expressions:

```
int i = 0;
int b = 4;
inc c = 6;
For i = 0; i<b*c; i = i +1 then
print(i);
end
```

Relational

Relational operators are also used for control flow and for logical operations. These are the standard operations such as less than, equal, not equal etc.

Biological

Biological expressions include the unary operators Complement (@), Transcribe (->) and two versions of Translate (+> and %>). Complement can only operate on DNA, Transcribe can only be executed on DNA to produce RNA. Out of the two versions of Translate, the first version takes RNA as an input and outputs a peptide, the second version takes DNA as an input and outputs a peptide. For example:

```
DNA sample  = #GCTGATAGCTCCTGATGCGTCAA
print(@sample)
```

The code above will output the complement of the original strand sample

2.2.3 Loops and Conditional Statements

Loops and conditionals are controlled by boolean statements, DNA# supports If...Else/Elseif, For and While. The language also allows nesting of these statements:

```
int i =0;
int j =0;
int b = 7;
int c = 6;
while (i +1)<c then
  print(i);
  i=i+1;
  for j=0; j<7; j=j+1 then
    print(j);
  end
end
```

2.2.4 Using Built-in Functions

DNA# has the following built-in functions: print, read and readFASTA. The print function will output results. Both read functions are described in 2.2.6.

```
print(“hello”);
      //hello
```

2.2.5 Defining and Calling Functions

Functions are defined by stating their return type, function name, formals and then a list of statements that define functionality and operations. A return value for the function can be defined which must match the type defined for the function. The function is ended with keyword ‘end’. Example of a function declaration:

```
int adder (int a, int b, int c)
      c = a +b;
      return c;
end
print(adder(6,7,0))
```

2.2.6 Reading files

There are two ways to import files with genetic sequences, readFASTA and read. The first type of read imports FASTA files that contain genetic sequences, these imports are stored as DNA types because they original files only contain genetic sequence information. The second type of read imports text files are are stored as strings since they may contain information that is not of genetic sequences and may have to be parsed by the user before casting to a DNA type.

```
readFASTA(‘samplesequence.fasta’)
read(‘geneticdata.txt’)
```

2.2.7 Generating Sample Programs to Comprehend Program Structure

```
      //Sample program
DNA sample = readFASTA(‘sample.fasta’);
print(sample);
      //aggc
RNA sample2 = sample->;
int i = 0;
for i=0; i<sample.length; i++
then
print(sample[i]);
end

      //ucch
Pep protein = sample2+>;
DNA David = “atgc”;
DNA Watkins = “cgta”;
DNA CoolGuy = David + Watkins;
      //atgccgta
```

3 Language Reference Manual

3.1 Introduction

DNA# is a procedural language used for carrying out genetic computational calculations and is ideal for writing scripts to make such computation easier. Designed to assist chemists and biologists with a basal knowledge of programming, DNA# focused on simplicity and flexibility across different skill levels.

3.2 Lexical Conventions

3.2.1 Identifiers

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

The regular expression for the identifiers is the following:

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

3.2.2 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	print
then	return	void	nuc
int	double	char	bool
aa	DNA	RNA	cast
Pep	read	length	readFASTA

3.2.3 Whitespace and Comments

DNA# is mostly a free-format language, and whitespace is ignored in the code. The exceptions are that whitespace is used to differentiate different words of the language, such as keywords, identifiers and constants. Between words, however, the amount of whitespace has no bearing on the execution of the program.
DNA# supports comments, which are text in the program ignored by the code, with the same syntax as C++ comments. Single line comments can be made with two forward slashes, //, after which every character in the same line is ignored by the code. Multi-line comments can be made with an opening forward slash followed by an asterisk, /*, and a closing asterisk followed by a forward slash, */. Examples of both types of comments are below:

```
// This is a single line comment
/* This is a
    multi-line
        comment */
```

3.2.4 Literals

Nuc	Nuc type data should always be referenced with a #, following a character as shown in the following example: `nuc sample = #a;`

AA	Amino acid data should always be referenced with a single quotes, as shown in following example: `aa = ‘A’;`
Integer	Integers should always be referenced as digits, as shown in the following example: `int a = 3;`
Double	Doubles should be referenced as digits, with a mandatory . after, as well as any digits after that, as seen in the following example: `double a = 3.5;`

Bool	When dealing with bool literals, they can only be specified as the true keyword or the false keyword, as shown in the example below:
	
	`bool hello = true;`
	`bool hi = false;`

Characters	Character literals should always be referenced with single quote, as shown in the following example: `char a = ‘a’;`
Strings	String literals should always be referenced with double quotes, as shown in the following example: `string a = “hello”;`
DNA/RNA/Pep	DNA, RNA and Peptide literals should always be referenced with an opening #, as shown in the following example: `DNA sample = #atgc;`

3.3 Data Types

The following keywords indicate the data types supported by DNA# and can be used as types in variable declarations. Please refer to section 2 for the specific syntax of how to declare and access variables.

Standard types

Type	Definition	Values
`bool`	Boolean	true, false

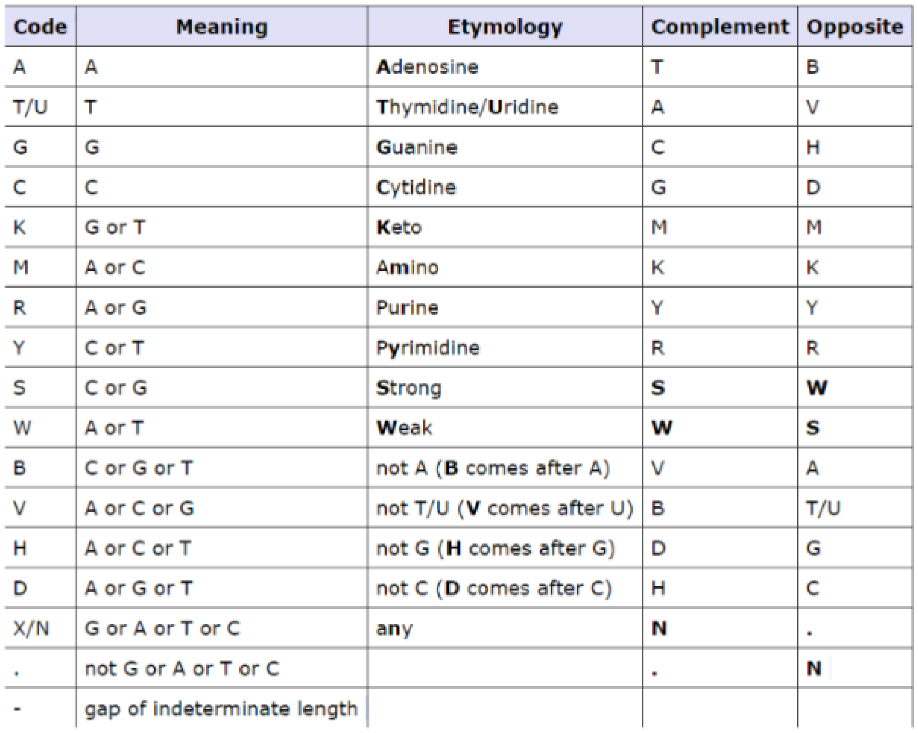
`int`	Integer	32-bit integers, values ranging from -2,147,483,648 to 2,147,483,647 inclusive
`double`	Double Floating Point	64-bit floating point numbers, ranging from 2.2E-308 to 1.8E+308 with 53 significant bits
`void`	Valueless	no values
`char`	Character	numbers using ASCII encoding
`string`	Sequence of characters	array of char

Primitive

Type	Definition	Values
`nuc`	Individual nucleotides	`A, T, G, C, U (upper or lowercase)`
`aa`	Amino acid, basic chemical structures composing a protein	`A, C, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, U, V, W, Y (upper or lowercase)`

The relation between the different values of nuc and their complementary, transcribed and translated counterparts can be seen in Figure 1 (Note: DNA# currently only supports the A,T,G,C and U nucleotides.

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			

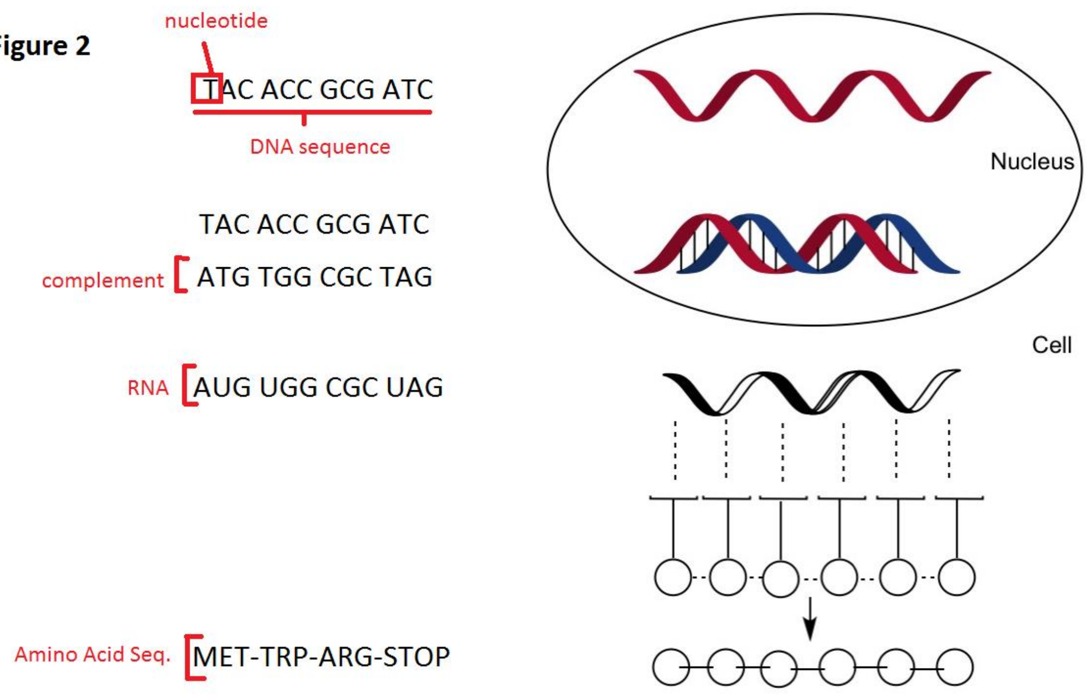
source: http://www.boekhoff.info/?pid=data&dat=fasta-codes

Complex

Type	Definition/Value	Sample Values
`dna`	A Sequence of A,T, G, C Deoxynucleotides	AGTCC
`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`
`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



3.4 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
```

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’. Please note that neither the intermediary then keyword nor the end keyword may be omitted. 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 then
 …
end
```

while statement

Below is a basic example of while statement. It starts with a keyword ‘while’ and ends with an ‘end’. Please note that neither the intermediary then keyword nor the end keyword may be omitted. 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 then
 …
end
```

3.5 Built in functions

Function	Definition	Syntax
print	Prints the value of any data type. For boolean values, will print true or false. Returns non-zero integer value upon success, but return value should not be used.	`print(DNA)`
length	Returns length of variable as integer. For primitive and standard data types, prints a value of 1. For arrays, prints the number of elements and for DNA, string, RNA and Peptide types, will print the amount of characters or nucleotides contained in the variable.	`variablename.length`
read	Reads the text contents of a file with any extension and returns contents as a string.	`read(“sample.txt”)`
readFASTA	Reads the text contents of a single sequence standard format FASTA file with no variable nucleotides and returns contents as DNA.	`readFASTA(“samplefile.fasta”)`

3.6 Functions

Like functions in C or C++, all functions have explicit return type and input arguments, 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 except arrays. 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# except arrays. Users can specify as many arguments as the user may wish. DNA# also supports functions without any input, but the brackets ‘()’ can’t be omitted. Here is one example:

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

3.7 Blocks

Blocks in DNA# represent segments of code that carry their own scope as defined by the scoping rules of DNA#. All segments that terminate in an end statement are considered blocks. Additionally, users can define their own blocks with the begin and end keywords with the following syntax:

```
begin

// block code with local scope here

end
```

3.8 Type Casting

DNA# allows for fluid data conversion between all DNA types and strings. DNA, RNA, Peptides and strings can all be type casted to each other, provided the value being casted fits the rules of the container variable (i.e. the string “xqt” cannot be cast to a DNA container because DNA only allows a,t,g,c characters). The syntax for type casting is as follows, and returns the casted data:

```
cast<type_to_cast_into>(expression_to_be_casted)

An example is provided below:
      string a = “atgc”;
      DNA b = cast<DNA>(a);
      print(b);
            // atgc
```

3.9 Scoping rules

The scoping rules for DNA# are similar to C, without the existence of global variables and a main function. The rules can be broken down as follows:

Lifetime of a variable: A variable’s lifetime where it can be accessed exists from its declaration until the end of its scope. What defines what a scope is is defined in the following rules.

Functions: Every function body has its own local scope that only has accesses to the function's formal arguments and local variables. With the exception of the return value, nothing outside the scope of the function body has access to variables in the function's local scope.

Blocks: Every block that ends with the keyword "end" has its own local scope (e.g. loops, logic flow and blocks). The lifetime of variables declared inside a block terminate after the "end" keyword corresponding to the block, and with the exception of functions, blocks can be nested. This means that all nested inner blocks have access to the variables of all outer blocks. However, in the case of identifiers clashing, precedence is given to the inner-most declaration of the variable. In this way, scope precedence is "inside out."

Outer statement list: The outermost statement list, oftentimes the beginning point of a DNA# program, can be considered to be in its own, outermost block. Thus, every other scope, with the exception of function scopes, have access to the outermost statement list variables.

The following snippet of code illustrates some of the scoping rules in DNA#:

```
int a = 1;
int b = 2;
print(a);
     // 1
int i = 0;
for  i; i<5; i = i+1 then
     print(b);
          //2
     int a = 3;
     print(a);
          // 3
end
print(a);
     //1
print(b)
     //2

int foo()
     int a = 3;
     print(a);
          //3
     print(b);
          //ERROR: variable b is undeclared in this scope
end
```

3.10 Expressions and Operators

3.10.1 Primary Expressions

Literals: these are the terminal expressions which all other expressions boil down to, and include all literal data types as described in section 2.2.1.

Operators

The following are the operators allowed in our language. In the syntax, all italicized items are variables or literals accepted by the operator.

- Arithmetic operators: DNA# supports addition, subtraction, multiplication, division, modulo and raise to power as its arithmetic operators.

Operator	**Syntax**
Addition (Note: this operator is overloaded to concatenate strings, as described previously)	*expression* + *expression*
Subtraction	*expression* - *expression*
Division	*expression* / *expression*
Multiplication	*expression* * *expression*
Exponent	*expression1* ^ *power*
Modulo	*expression* % *expression*

- Relational and logical operators: DNA# supports the following relational and logical operators. All return a Boolean

Operator	**Syntax**
And	*expression* & *expression*
Or	*expression* | *expression*
Negation	! *expression*
Equal to	*expression* == *expression*
Not equal to	*expression* != *expression*
Less than	*expression* < *expression*
Greater than or equal to	*expression* >= *expression*
Less than or equal to	*expression* <= *expression*
Greater than	*expression* > *expression*

- Biological operators:
- Relational and logical operators: DNA# supports the following relational and logical operators. All return a Boolean

Operator	Syntax
Complement of sequence (DNA or RNA)	@*sequence*
Transcribe (DNA->RNA)	*DNA*->
Translate (RNA->Pep)	*RNA*+>
Translate2(DNA->Pep)	*DNA* %>

Precedence of operators

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

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

Associativity of operators

The following operators are left-associative:

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

The following are right-associative:

`=`	`@`	`!`

Function Calls

Function calls can be done from any part of the program, except if the function being called is undeclared or if a function is calling itself (DNA# does not support recursion). A function The syntax for function calls is as follows:

```
functionname(argument1, argument2)
```

Array Access

Arrays, DNA, RNA and Peptides can be indexed for specific elements. The syntax to do so is below, where ID is the identifier of the array being indexed, and expression is some expression that reduces to an integer.

```
ID[expression]
```

Parenthesized Expressions

Parenthesized expressions are also expressions in DNA#, and take precedence over non-parenthesized expressions. The syntax for parenthesized expressions is below, with expression being any expression described above:

```
(expression)
```

Built in functions and type casts

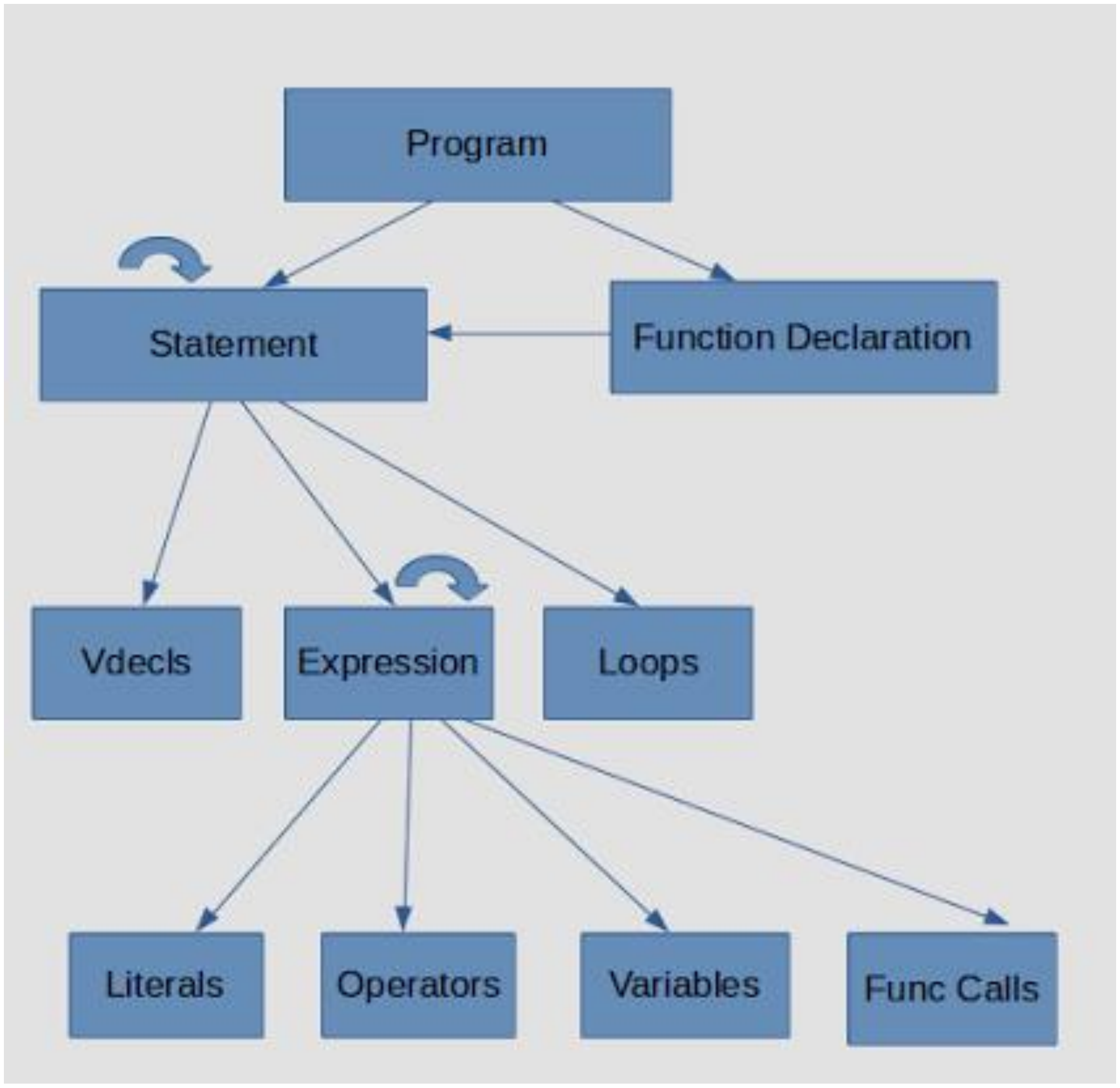
All built in functions and type are also expressions in DNA#. Please refer to Section 3.5 and Section 3.6 for their specific syntax

3.11 Statements

In DNA#, statements represent individual instructions of code, and are always terminated by a semicolon. It is recommended practice to separate successive statements with a new line. The following are considered valid statements in DNA#, the syntax and meaning of all of which are described in the previous sections: all expressions, function declarations, returns, variable declaration, array declarations, logic flow, loops, blocks and other statements.

3.12 Grammar

Below is a simplified overview of our grammar, consistent with the design described in the document. A more detailed version is available in the source code in section 10.1



4 Project Plan

4.1 Planning

The project progressed in the form of an Agile methodology wherein the team set milestones for each iteration. There were weekly standup meetings wherein each team member stated what had been accomplished and further steps were discussed. The goals for each iterations were recommended by the professor and the team followed them as shown in the Project Timeline and Project Log below. Plenty of code was produced in pairs to generate better quality, bug free, and more efficient code. The pairs of coding for most of the semester were Min, Stan and Oriana as pair 1 and Cloud and Aalhad as pair 2.

Additionally, much of the design of the language was done as a collaborative effort between all team members and outside advisors. During each meeting, significant time was allotted to discuss the inclusion, exclusion or design of features of the language. Outside experts, such as Dr. Yaniv Erlich, professor of computer science at Columbia University, Anuved Verma, M.S student at University of Chicago and Michael Cheng, M.D student at Weill Cornell School of Medicine were consulted.

4.2 Language Specifications

The DNA# project was started by setting features the language ought to be able to implement. Furthermore, the basic approach on how the compiler was going to work was determined. As the language was supposed to be simple for non computer scientists, the team initially decided to be inspired by the syntax of Python, however, without the need for indentation. Much of the syntax was also inspired from C++ and MATLAB. The syntax was fully determined prior to the initiation of any coding. While the syntax did not change much during the project, the expected implementation of individual functionalities was altered. Major changes included not incorporating the main function as well as omitting global variables.

4.3 Development

The development proceeded by first finishing the lexical analysis followed by parsing and abstract syntax tree construction. Consequently, code generation was started along with static semantic analysis. Static semantic analysis was incorporated after code generation was finished. When the basic pipeline was completed more DNA# specific features were added.

4.4 Testing

Testing was completed by evaluating key aspects of each level of the compiler, mainly the expected outputs of the semantic analyzer and code generation. Every node of the abstract syntax tree was tested positively and negatively, first by observing output of programs that were deemed correct, then by looking for exceptions of programs that were deemed incorrect syntactically. In addition, some of the recurring aspects of the tree were testing, mainly nesting different types of statements within others. The second level of testing focused on the correctness of code generation, ie. whether outputs of operations matched what was expected. In order to test correctness of operations produced by code generation we examined cases such as empty or inappropriate inputs.

4.5 Team Responsibilities

While all team members worked wherever most work was necessary to follow the project timeline, each team member focused on the following tasks:

Team Member	**Responsibility**
Oriana Fuentes	Semantics, Compiler Front End, Testing, Documentation
Stan Peceny	Semantics, Compiler Front End, Documentation
Aalhad Patankar	Compiler Front End, Code Generation
Cloud Yu	Compiler Front End, Code Generation
Min Fan	Compiler Front End, Testing, Semantics

4.6 Project Timeline

The project timeline is displayed in the table below:

Date	Milestone
September 28, 2016	Language Proposal
October 15, 2016	Language Reference Manual
November 1, 2016	Lexical Analysis, Parsing, AST
November 15, 2016	Code Generation
November 21, 2016	Hello World Program
December 5, 2016	Static Semantic Checking
December 10, 2016	Testing/Debugging
December 18, 2016	Final Report

4.7 Project Log

The project proceeded as displayed in the table below:

Date	Milestone

September 23, 2016	Language Proposal
October 15, 2016	Language Reference Manual
October 25, 2016	Lexical Analysis, Parsing, AST
November 15, 2016	Code Generation
November 20, 2016	Hello World Program
December 15, 2016	Static Semantic Checking
December 17, 2016	Testing/Debugging
December 18, 2016	Final Report

4.8. Software Development Environment

Compiler was coded in OCaml version 4.02.3

OCaml's extensions OCamllex and OCamlyacc were used for compiling the scanner and parser, respectively.

Vim was the environment of choice for all team members.

4.9 Programming Style Guide

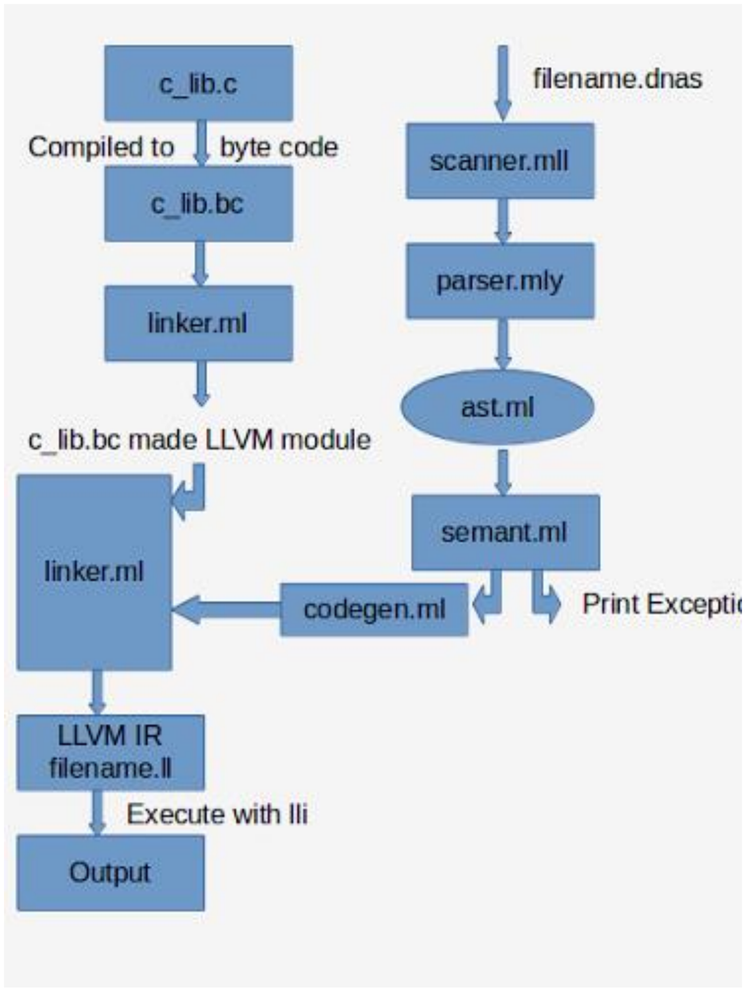
The team strived to use OCaml programming guidelines to make the code simple, readable, and efficient.

One tab was used for indentation throughout the project and each block was preceded with a comment to clarify the functionality implemented.

5 Architecture

5.1 Compiler

Below is a picture of components constituting the language's architecture. The major components constitute the lexical analysis, parsing, static semantic analysis, code generation, and C-implemented functions.



There are two main modules in the architecture that need to be linked. The first comes directly through scanner directly to linker while the other part is implemented as part of the C-implemented functions in the left branch.

5.1.1 Lexical Analysis

The scanner.mll performs the lexical analysis by taking the user program as the input and converting it into tokens. Hence, lexical analysis deletes whitespace characters and comments.

5.1.2 Parsing

Parsing is responsible for taking the tokens from the scanner, parsing it, and putting it into an abstract syntax tree. Hence, it calls ast.ml to create the tree structure. The abstract syntax tree is passed for static semantic analysis.

5.1.3 Static Semantic Checking

The abstract syntax tree is passed into semant.ml, which performs the static semantic analysis. Hence, the abstract syntax tree is traversed to resolve symbols and check their types. Static semantic analysis throws an exception if the abstract syntax tree is not valid. On the other hand, the code proceeds to code generation from the parser if static semantic checking passes.

5.1.4 Code Generation

Code generation takes as input the abstract syntax tree and generates the LLVM intermediate representation. The program in the abstract syntax tree is split into statements (stmts) list and functions (funcs) list. In code generation, the statements are put into a pseudo-main function where the program starts being executed. The pseudo-main function is added to the funcs list from the abstract syntax tree. Consequently, a function declaration contains a formals list and a stmts list. The formals are added into varis, a map to save all local vars. On the other hand, the stmts list is passed to a add_stmt function which goes over each statement to generate intermediate representation. If the type of statement is a declaration, it is further added to varis.

5.2 C Libraries

5.2.1 C_lib.c/c_lib.bc

Since some functionalities cannot be implemented well in OCaml, a set of DNA# functions has been implemented in C language within c_lib.c. These functions include complement, concatenate, readFASTAFile, transcribe, and translate. The functions in c_libc.c are compiled into bytecode c_lib.bc before being merged with the 'main' code in the linker.

5.2.2 Linker.ml

The C functions need to be merged with the main module in order to generate the intermediate representation and using lli obtain the output of the input program. This happens in the linker.ml file with a link_bc function.

6 Testing

Testing was separated into black box testing and grey box testing. Once the code was completed, blackbox tests were done first to confirm that the program seemed to be working as expected without peeking into the code. Afterward, grey box tests were implemented. As everything in DNA# is stored on the stack, no memory leaks are present. The only memory issues that could occur are out of bound errors in the array. We strived to check for these errors in codegen.ml. Thus, greybox testing primarily focused on determining if the out of bounds checks were implemented correctly. A comprehensive set of tests was added at the end.

6.1 Testing Phases

6.1.1 Unit Tests

As part of the test intensive development, unit tests were written simultaneously with each function. Hence, by the time the code was completed a comprehensive set of unit tests had been present although many more needed to be implemented. The tests were small programs written in DNA# designed to test the syntax as well as the semantics of the language. As a consequence, these small programs further tested whether the entire pipeline of the compiler was functional. The tests were separated into test-to-pass as well as test-to-fail categories.

6.1.2 Integration Tests

The tests in the 'Unit Tests' section were further integration tests since they tested the entire pipeline. Hence, the pipeline was tested by inputting a diverse range of DNA# programs to employ all sections in the pipeline. The tests included the following features:

- Identifiers
  - Testing whether variable and function names start with an alphabetic letter or an underscore and are followed by any amount of alphanumeric characters or underscores.
- Keywords
  - Data types (int, bool, void, etc.) as well as complex data types (Seq, codon, DNA, RNA, Pep) were included in the programs for testing.
- Control Flow
  - for, while, etc.
- Types
  - dna, rna, peptide, etc.
- Arrays
  - Array tests stressed edge cases
- Built-in Functions
  - print
- Constants/Literals
  - bool, int, double, string
- Comments
  - C-style comments
- Operators
- Variable and Function Declaration

6.1.3 System Tests

The system tests were included along with the other tests and the sample programs were meant to encompass as much of the system architecture as possible. Hence, they were created to also cover the C-written portion of the architecture.

6.2 Automated Compilation Script

All *.dnas files are run in the dnas_tests folder when a ./test_DNAs.sh script is executed.

6.3 Test Suites

The folder dnas_tests contains the entire set of tests. The folder contains tests to fail as well as tests to pass. The tests are manually run using the test_DNAs.sh shell script. Below are the primary features tested for DNA#:

The tests consisted but were not limited to declarations, assignments, primitive as well as complex data operations, and control flow.

6.4 Sample Tests Run

Below are several examples of tests implemented.

Tests to pass:

The test below tests whether control flow, particularly, the for loop and the while loop work correctly.

test-hw2-ForWhile.dnas

```
int i=1;
for i=1;i<2;i=i+1 then
   print("Hello World !");
end


while i>1 then
   i=i-1;
   print("What is next?");
end
```

The next program tests the declaration and printing of DNA#'s data types.

test-hw13-printDataTypes.dnas

```
Pep oriana = #A-T-G-C;
int hi = 42;
double hello = 42.0;
bool isTrue = true;
string sample = "hello";
char samp = sample[0];
```

```
DNA min = #atgc;
RNA stan = #gcgc;
print(hi);
print(isTrue);
print(sample);
print(samp);
print(min);
print(hello);
print(stan);
print(oriana);
```

Tests to Fail:

The test below fails because an array is assigned to a DNA. The two types do not match.

fail-test-hw18-failArrayBoundaries.dnas

```
DNA test = #AA;
test = myarray[0];
print(myarray);
print(test);
```

The test below fails since it tries to add undeclared b to an integer.

fail-test-hw21-failAddition.dnas

```
int sample(int a, int a, double c)
 a = b +1;
end
```

6.5 Testing Roles

Cloud created the testing infrastructure and all other team members helped to write individual unit tests for DNA# functions to be implemented. The first set of tests were general and with very simple programs to verify the integrity of the compiler. In the later phases, tests were built when sections of the semantic checker and of the code generation were written. Stan further implemented some black-box tests to test the overall pipeline.

7 Lessons Learned

7.1 Oriana Fuentes

- Experience: The project and class overall were very different to anything I've worked on previously. Learning the principles behind how a language is built has expanded my understanding of the field, specifically how languages can be designed for certain kinds of data and operations. Working on a project related to genetics was also very rewarding because it's something I've been interested in since I was very young. Aside from the project itself, working with the team was a very positive experience as we all collaborated on most of the sections and worked well together throughout the semester.
- Advice: Select your team members carefully. Focus on finding peers you will get along with since you will be spending a lot of time together during the semester. Ensure you understand everything on the first assignment and work on understanding how MicroC is built early on.

7.2 Stan Peceny

- Experience: Objective Caml, known as OCaml, was developed in France and extends the Caml language with object-oriented constructs. Should you not have a prior experience with programming in a functional language, it is essential to get a deep understanding of the language at the very beginning of the semester prior to initiating the project. The project was a truly enjoyable experience since I have made some very close friends.
- Advice: I strongly believe it is important to start code generation as soon as possible and not wait until the static semantic analysis is finished. Furthermore, select a team that you will enjoy working with throughout the semester rather than picking a team based on the amount of programming experience. Communication as well as camaraderie is critical to performing well on the project. One ought to feel proud to have finished the project.

7.3 Aalhad Patankar

- Experience: DNA# was surely my favorite work of the semester. I have spent countless hours working on the code generation and truly enjoyed it. In fact, I found it fascinating to see the code put together and was very enthusiastic about seeing our programs produce correct outputs.
- Advice: I was very fortunate about having such amicable peers and I would impel others to select their teams carefully. The typical advice of 'start your project early' does apply and it is essential not to wait until the last minute to finish the project. **Finally, do not use tuples. Use records instead.**

7.4 Min Fan

- Experience: Interestingly, my PLT team members have become my closest friends at Columbia. I have only the most positive memories of our time working together. We had

plenty of discussions unrelated to the project which in turn improved our working environment.
- Advice: Your team members are essential. It is not as relevant whether they have experience in functional programming, however, you must get along well. If possible, try to learn OCaml before the semester has started so that you can start coding your compiler immediately.

7.5 Nan Yu (Cloud)

- Experience: I have worked a lot on the code generation. My team members were great, the project was fun, and I have learned much as well as became a better programmer. Since we were on very good terms the team worked efficiently and the work was evenly split.
- Advice: I would recommend to pick the team with caution. Since the project is very time consuming I would sincerely recommend to select team members that will also become your close friends. Having a good team where everyone is enthusiastic and friendly will help to increase efficiency of your work.

8 Appendix

The following are two demo programs, which were also used for our live demo. The first one (Demo/demo.dnas) implements the dynamic programming approach to find the longest common subsequences of two DNA strands. The second one (Demo/demo_pep.dnas) implements finding the long common subsequences of peptides sequences, which are translated from the two DNA strands (one DNA strand is mutated from the other sequence). The second sample program adds less than ten lines of code to the first sample program, which demonstrates the easiness of our language to mimic biological process.

8.1 Demo/demo.dnas

```
-----------------------------------------------------------------------------------------------------------------------
DNA d1 =
#CACGCCCCAGCTCTGCCCTTGCAGAGGCAGAGTAGGGAAGAGCAAGCTGCCCGAGACGCAGGGGAAGGA
GGATGAGGGCCCTGGGGATGAGCTGGGGTGAACCAGGCTCCCTTTCCTTTGCAGGTGCGAAGCCCAGCGG
TGCAGAGTCCAGCAAAGGTGCAGGTATGAGGATGGACCTGATGGGTTCCTGGACCCTCCCCTCTCACCCT
GGTCCCTCAGTCTCATTCCCCCACTCCTGCCACCTCCTGTCTGGCCATCAGGAAGGCCAGCCTGCTCCCC
ACCTGATCCTCCCAAACCCAGAGCCACCTGATGCCTGCCCCTCTGCTCCACAGCCTTTGTGTCCAAGCAG
GAGGGCAGCGAGGTAGTGA;
DNA d2 =
#CACGCCCCAGCTCTGCCCTTGCAGAGGGAGAGGAGGGAAGAGCAAGCTGCCCGAGACGCAGGGGAAGGA
GGATGAGGGCCCTGGGGATGAGCTGGGGTGAACCAGGCTCCCTTTCCTTTGCAGGTGCGAAGCCCAGCGG
TGCAGAGTCCAGCAAAGGTGCAGGTATGAGGATGGACCTGATGGGTTCCTGGACCCTCCCCTCTCACCCT
GGTCCCTCAGTCTCATTCCCCCACTC;
print(d1);
print('\n');
print('\n');
print(d2);
print('\n');
int res = lcs_DNA(d1,d2);
// lcs_DNA longest common subsequence with DNA inputs
int lcs_DNA(DNA A, DNA B)
	int sizeA = A.length;
	int sizeB = B.length;
	int arr_size = (sizeA +1) * (sizeB + 1);
	int [arr_size] C;
	int i2 = 0;
	for i2; i2< arr_size; i2 = i2+1 then
		C[i2] = 0;
	end
	int i = sizeA; int j = sizeB;
	for i=sizeA; i >= 0; i = i-1 then
		for j=sizeB; j >= 0; j = j-1 then
			if i == sizeA  | j == sizeB then
				int index = getIndex(i, j, sizeA, sizeB);
				C[index] = 0;
			elseif (A[i] == B[j]) then
				int index1 = getIndex(i,j,sizeA,sizeB);
				int index2 = getIndex(i+1,j+1,sizeA,sizeB);
				C[index1] = 1 + C[index2];
			else
				int index3 = getIndex((i+1), j, sizeA, sizeB);
				int index4 = getIndex(i,(j+1), sizeA, sizeB);
				int index5 = getIndex(i,j, sizeA, sizeB);
				int firstVal = C[index3];
				int secondVal = C[index4];
				int max = getMax(firstVal, secondVal);
				C[index5] = max;
			end
		end
	end

/*	for i = 0; i<sizeA*sizeB; i=i+1 then
		//print(C[i]);
		end		*/
	i= 0; j= 0;
	print('\n');
	print("Longest subsequence:");print('\n');print('\t');
	while (i < sizeA  & j<sizeB) then
		if (A[i] == B[j] ) then
			char temp = A[i];
			print(temp);
			i = i + 1; j = j+ 1 ;
		elseif C[getIndex(i+1, j, sizeA, sizeB)] >= C[getIndex(i,
j+1, sizeA, sizeB)] then
			i = i +1 ;
		else
			j = j + 1;
		end
	end
	print('\n');
	print("The length of the longest
subsequence:");print('\n');print('\t');
	print(C[0]);print('\n');
	return C[0];
```

```
end

int getIndex(int x, int y, int col, int row)
     return row * x + y;
end

int getMax(int x, int y)
     int result = -1;
     if x > y then result = x; else result = y; end
     return result;
end
-----------------------------------------------------------------------
------------------------------------------------------
```

```
Output (command: ./runDNAs.sh -r Demo/demo.dnas):
-----------------------------------------------------------------------
------------------------------------------------------
ACGCCCCAGCTCTGCCCTTGCAGAGGCAGAGTAGGGAAGAGCAAGCTGCCCGAGACGCAGGGGAAGGAGG
ATGAGGGCCCTGGGGATGAGCTGGGGTGAACCAGGCTCCCTTTCCTTTGCAGGTGCGAAGCCCAGCGGTG
CAGAGTCCAGCAAAGGTGCAGGTATGAGGATGGACCTGATGGGTTCCTGGACCCTCCCCTCTCACCCTGG
TCCCTCAGTCTCATTCCCCCACTCCTGCCACCTCCTGTCTGGCCATCAGGAAGGCCAGCCTGCTCCCCAC
CTGATCCTCCCAAACCCAGAGCCACCTGATGCCTGCCCCTCTGCTCCACAGCCTTTGTGTCCAAGCAGGA
GGGCAGCGAGGTAGTGA

CACGCCCCAGCTCTGCCCTTGCAGAGGGAGAGGAGGGAAGAGCAAGCTGCCCGAGACGCAGGGGAAGGAG
GATGAGGGCCCTGGGGATGAGCTGGGGTGAACCAGGCTCCCTTTCCTTTGCAGGTGCGAAGCCCAGCGGT
GCAGAGTCCAGCAAAGGTGCAGGTATGAGGATGGACCTGATGGGTTCCTGGACCCTCCCCTCTCACCCTG
GTCCCTCAGTCTCATTCCCCCACTC

Longest subsequence:

CACGCCCCAGCTCTGCCCTTGCAGAGGAGAGAGGGAAGAGCAAGCTGCCCGAGACGCAGGGGAAGGAGGA
TGAGGGCCCTGGGGATGAGCTGGGGTGAACCAGGCTCCCTTTCCTTTGCAGGTGCGAAGCCCAGCGGTGC
AGAGTCCAGCAAAGGTGCAGGTATGAGGATGGACCTGATGGGTTCCTGGACCCTCCCCTCTCACCCTGGT
CCCTCAGTCTCATTCCCCCACTC
The length of the longest subsequence:
    233
-----------------------------------------------------------------------
------------------------------------------------------
```

```
Demo/demo_pep.dnas
-----------------------------------------------------------------------
------------------------------------------------------
DNA d1 =
#CACGCCCCAGCTCTGCCCTTGCAGAGGCAGAGTAGGGAAGAGCTAGCTGCCTGAGACCCAGGGGATGGA
GAATGAGGGACCTAGGGATGAGCTGGGGTGAACCAGGCTCCCTTTCCTTTGCAGGTGCGAAGCCCAGCGG
TGCAGAGTCCAGCAAAGGTGCAGGTATGAGGATGGACCTGATGGGTTCCTGGACCCTCCCCTCTCACCCT
GGTCCCTCAGTCTCATTCCCCCACTCCTGCCACCTCCTGTCTGGCCATCAGGAAGGCCAGCCTGCTCCCC
ACCTGATCCTCCCAAACCCAGAGCCACCTGATGCCTGCCCCTCTGCTCCACAGCCTTTGTGTCCAAGCAG
GAGGGCAGCGAGGTAGTGA;
Pep p1 = d1%>;
print(p1);
print('\n');

DNA d2 =
#CACGCCCCAGCTCTGCCCTTGCAGAGGGAGAGGAGGGAAGAGCAAGCTGCCCGAGACGCAGGGGAAGGA
GGATGAGGGCCCTGGGGATGAGCTGGGGTGAACCAGGCTCCCTTTCCTTTGCAGGTGCGAAGCCCAGCGG
TGCAGAGTCCAGCAAAGGTGCAGGTATGAGGATGGACCTGATGGGTTCCTGGACCCTCCCCTCTCACCCT
```

```
GGTCCCTCAGTCTCATTCCCCCACTCCTGCCACCTCCTGTCTGGCCATCAGGAAGGCCAGCCTGCTCCCC
ACCTGATCCTCCCAAACCCAGAGCCACCTGATGCCTGCCCCTCTGCTCCACAGCCTTTGTGTCCAAGCAG
GAGGGCAGCGAGGTAGTGAAGAGACCCAGGCGCTACCTGTATCAATGGCTGGGGTGAGAGAAAAGGCAGA
GCTGGGCCAAGGCCCTGCCTCTCCGGGATGGTCTGTGGGGGAGCTGCAGCAGGGAGTGGCCTCTCTGGGT
TGTGGTGGGGGTACAGGCAGCCTGCCCTGGTGGGCACCCTGGAGCCCCATGTGTAGGGAGAGGAGGGATG
GGCATTTTGCACGGGGGCTGATGCCACCACGTCGGGTGTCTCAGAGCCCCAGTCCCCTACCCGGATCCCC
TGGAGCCCAGGAGGGAGGTGTGTGAGCTCAATCCGGACTGTGACGAGTTGGCTGACCACATCGGCTTTCA
GGAGGCCTATCGGCGCTTCTACGGCACTCCGGTCTAGGGTGTCGCTCTGCTGGCCTGGCCGGCAACCCCA
GTTCTGCTCCTCTCCAGGCACCCTTCTTTCCTCTTCCCCTTGCCCTTGCCCTGACCTCCCAGCCCTATGG
ATGTGGGGTCCCCATCATCCCAGCTGCTCCCAAATAAACTCCAGAAG;
Pep p2 = d2%>;
print(p2);

int res = lcs_Pep(p1,p2);
int lcs_Pep(Pep A, Pep B)
	int sizeA = A.length;
	int sizeB = B.length;
	int arr_size = (sizeA +1) * (sizeB + 1);
	int [arr_size] C;
	int i2 = 0;
	for i2; i2< arr_size; i2 = i2+1 then
		C[i2] = 0;
	end
	int i = sizeA; int j = sizeB;

	for i=sizeA; i >= 0; i = i-1 then
		for j=sizeB; j >= 0; j = j-1 then
			if i == sizeA  | j == sizeB then
				int index = getIndex(i, j, sizeA, sizeB);
				C[index] = 0;
			elseif (A[i] == B[j]) then
				int index1 = getIndex(i,j,sizeA,sizeB);
				int index2 = getIndex(i+1,j+1,sizeA,sizeB);
				C[index1] = 1 + C[index2];
			else
				int index3 = getIndex((i+1), j, sizeA, sizeB);
				int index4 = getIndex(i,(j+1), sizeA, sizeB);
				int index5 = getIndex(i,j, sizeA, sizeB);
				int firstVal = C[index3];
				int secondVal = C[index4];
				int max = getMax(firstVal, secondVal);
				C[index5] = max;
			end
		end
	end

	i= 0; j= 0;
	print('\n');
	print("Longest subsequence:");print('\n');print('\t');
	while (i < sizeA  & j<sizeB) then
		if (A[i] == B[j] ) then
			char temp = A[i];
			print(temp);
			print('-');
			i = i + 1; j = j+ 1 ;
		elseif C[getIndex(i+1, j, sizeA, sizeB)] >= C[getIndex(i,
j+1, sizeA, sizeB)] then
			i = i +1 ;
```

```
            else
                  j = j + 1;
            end
      end
      print('\n');
      print("The length of the longest 
subsequence:");print('\n');print('\t');
      print(C[0]);print('\n');
      return C[0];
end

int getIndex(int x, int y, int col, int row)
      return row * x + y;
end

int getMax(int x, int y)
      int result = -1;
      if x > y then result = x; else result = y; end
      return result;
end

--------------------------------------------------------------------------------------------------------------------------------------

Output (command: ./runDNAs.sh -r Demo/demo_pep.dnas):
--------------------------------------------------------------------------------------------------------------------------------------
V-R-G-R-D-G-D-V-S-V-S-S-L-L-D-R-R-T-L-G-P-L-P-L-T-P-W-I-P-T-R-P-H-L-V-
R-G-K-G-D-V-H-A-S-G-R-H-V-S-G-R-F-H-V-H-T-P-T-W-T-T-Q-G-P-G-R-G-E-W-D-
Q-G-V-R-V-R-G
V-R-G-R-D-G-D-V-S-L-S-S-L-L-V-R-R-A-L-R-P-L-P-P-T-P-G-T-P-T-R-P-H-L-V-
R-G-K-G-D-V-H-A-S-G-R-H-V-S-G-R-F-H-V-H-T-P-T-W-T-T-Q-G-P-G-R-G-E-W-D-
Q-G-V-R-V-R-G
Longest subsequence:
      V-R-G-R-D-G-D-V-S-S-S-L-L-R-R-L-P-L-P-T-P-P-T-R-P-H-L-V-R-G-K-G-
D-V-H-A-S-G-R-H-V-S-G-R-F-H-V-H-T-P-T-W-T-T-Q-G-P-G-R-G-E-W-D-Q-G-V-R-
V-R-G-
The length of the longest subsequence:
      70
--------------------------------------------------------------------------------------------------------------------------------------
```

8.2 Translation Table

DNA			RNA			AminoAcid	AA Acronym
A	A	A	U	U	U	phenylalanine	F
A	A	G	U	U	C	phenylalanine	F
A	A	T	U	U	A	leucine	L
A	A	C	U	U	G	leucine	L
A	G	A	U	C	U	serine	S
A	G	G	U	C	C	serine	S

A	G	T	U	C	A	serine	S
A	G	C	U	C	G	serine	S
A	T	A	U	A	U	tyrosine	Y
A	T	G	U	A	C	tyrosine	Y
A	T	T	U	A	A	terminal	terminal
A	T	C	U	A	G	terminal	terminal
A	C	A	U	G	U	cysteine	C
A	C	G	U	G	C	cysteine	C
A	C	T	U	G	A	terminal	terminal
A	C	C	U	G	G	tryptophan	W
G	A	A	C	U	U	leucine	L
G	A	G	C	U	C	leucine	L
G	A	T	C	U	A	leucine	L
G	A	C	C	U	G	leucine	L
G	G	A	C	C	U	proline	P
G	G	G	C	C	C	proline	P
G	G	T	C	C	A	proline	P
G	G	C	C	C	G	proline	P
G	T	A	C	A	U	histidine	H
G	T	G	C	A	C	histidine	H
G	T	T	C	A	A	glutamine	Q
G	T	C	C	A	G	glutamine	Q
G	C	A	C	G	U	arginine	R
G	C	G	C	G	C	arginine	R
G	C	T	C	G	A	arginine	R
G	C	C	C	G	G	arginine	R
T	A	A	A	U	U	isoleucine	I
T	A	G	A	U	C	isoleucine	I
T	A	T	A	U	A	isoleucine	I
T	A	C	A	U	G	methionine	M (origin)
T	G	A	A	C	U	threonine	T
T	G	G	A	C	C	threonine	T
T	G	T	A	C	A	threonine	T
T	G	C	A	C	G	threonine	T
T	T	A	A	A	U	aspartic	D
T	T	G	A	A	C	aspartic	D
T	T	T	A	A	A	lysine	K
T	T	C	A	A	G	lysine	K
T	C	A	A	G	U	serine	S
T	C	G	A	G	C	serine	S
T	C	T	A	G	A	arginine	R

T	C	C	A	G	G	arginine	R
C	A	A	G	U	U	valine	V
C	A	G	G	U	C	valine	V
C	A	T	G	U	A	valine	V
C	A	C	G	U	G	valine	V (origin)
C	G	A	G	C	U	alanine	A
C	G	G	G	C	C	alanine	A
C	G	T	G	C	A	alanine	A
C	G	C	G	C	G	alanine	A
C	T	A	G	A	U	aspartic	D
C	T	G	G	A	C	aspartic	D
C	T	T	G	A	A	glutamic	E
C	T	C	G	A	G	glutamic	E
C	C	A	G	G	U	glycine	G
C	C	G	G	G	C	glycine	G
C	C	T	G	G	A	glycine	G
C	C	C	G	G	G	glycine	G

9 Final Code

DNAs.ml

```
open Ast

type action = Ast | Llvm | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
   List.assoc Sys.argv.(1) [ ("-a", Ast);
           ("-l", Llvm);
           ("-c", Compile) ]
 else Compile in

 let lexbuf = Lexing.from_channel stdin in
 let ast = Parser.program Scanner.token lexbuf in
 Semant.check ast;
 match action with
   Ast-> print_string (Ast.string_of_program ast)
 | Llvm-> print_string (Llvm.string_of_llmodule (Codegen.translate
ast))
 | Compile->let m = Codegen.translate ast in
   Llvm_analysis.assert_valid_module m;
   print_string (Llvm.string_of_llmodule m)
```

Scanner.mll

```
(* Ocamllex scanner for DNA# *)

{
     open Parser
     let unescape s = Scanf.sscanf ("\"" ^ s ^ "\"") "%S%!" (fun x -
> x)
}

let seq = ['A''G''C''a''g''c']*
let dna = ['A''G''T''C''a''g''t''c']+
let rna = ['A''G''U''C''a''g''u''c']+
let nuc = ['A''G''U''C''a''g''u''c''T''t']
let dnuc = ['A''G''T''C''a''g''t''c']
let rnuc = ['A''G''U''C''a''g''u''c']
let aa = ['A''C'-'I''K'-'N''P'-'T''U'-'W''Y''a''c'-'i''k'-'n''p'-
't''u'-'w''y']
let pep = (aa '-')+ aa
let escape = '\\' ['\\' ''' '"' 'n' 'r' 't']
let escape_char = ''' (escape) '''
let ascii = ([' '-'!' '#'-'[' ']'-'~'])
let digit = ['0'-'9']
let char = ''' ( ascii | digit ) '''

rule token = parse
 [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }          (* Comments *)
| "//"     {singlecom lexbuf}          (* single line comment*)
| "variable"{VARIABLE}                       (* Test, remove after *)
(* Assignment Bop *)
| '='       {ASSIGN}
(* Non-functional stuff & flow control key words *)
| '.'       {PERIOD}
| "length" {SIZEOF}
| '('      {LPAREN}
| ')'      {RPAREN}
| '['       {LBRACK}
| ']'       {RBRACK}
| ';'      {SEMI}
| ','      {COMMA}
| "end"    {END}
| "if"     {IF}
| "elseif"  {ELSEIF}
| "else"   {ELSE}
| "for"    {FOR}
| "while"   {WHILE}
| "continue"{CONTINUE}
| "break"   {BREAK}
| "include" {INCLUDE}
| "local"   {LOCAL}
| "then"   {THEN}
| "return"  {RETURN}
| "readFASTA" {FREAD}
| "read"     {READ}
| "cast"     {CAST}
(* Arithmetic Binary Operators*)
| '+'       {PLUS}
| '-'       {MINUS}
| '*'       {TIMES}
| '/'       {DIVIDE}
| '%'      {MODULO}
| '^'      {EXPONENTIAL}
(* Logical Binary Operators *)
| '&'      {AND}
| '|'      {OR}
| '!'      {NOT}
(* Relational Binary Operators *)
| "=="     {EQ}
| "!="     {NEQ}
| '<'      {LT}
| "<="     {LEQ}
| '>'      {GT}
| ">="     {GEQ}
(* Bio Expression Operators *)
| '@'      {COMPLEMENT}
| "->"     {TRANSCRIBE}
| "+>"     {TRANSLATE}
| "%>"     {TRANSLATETWO}
(*Data Types *)
| "int"    {INT}
| "bool"   {BOOL}
| "void"   {VOID}
| "char"   {CHAR}
| "double"  {DOUBLE}
| "aa"     {AA}
| "nuc"    {NUC}
| "string"     {STRING}
(*Complex Data Types *)
| "Seq"    {SEQUENCE}
| "DNA"    {DNA}                           (* MAKE GENERAL FUNCTION
TO ALLOW RNA INPUT*)
| "RNA"    {RNA}
| "Pep"    {PEPTIDE}                            (*WRITE CODE TO
ALLOW FOR PEPTIDE ENTRIES *)
(* Literals *)
| "true"   {TRUE}
| "false"   {FALSE}

(* Removed char lit *)
| ['0'-'9']+ as lxm { INT_LIT(int_of_string lxm) }

| ((['0'-'9']+ '.' ['0'-'9']* | '.' ['0'-'9']+ )(['e''E']['+''-']?
['0'-'9']+)?) | (['0'-'9']+ (['e''E']['+''-']? ['0'-'9']+))) as lxm
{DOUBLE_LIT(float_of_string lxm)}

| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm {
ID(lxm)}

| '"'([^'"']* as lxm)'"'  {STRING_LIT(lxm)}
| '''(nuc as lxm)'''  { NUC_LIT(lxm) }
| '''(aa as lxm){ AA_LIT(lxm)}
| char as lxm         { CHAR_LIT( String.get lxm 1 ) }
| escape_char as lxm{ CHAR_LIT( String.get (unescape lxm) 1) }

| '#'(seq as lxm)     {SEQUENCE_LIT(lxm)}
| '#'(dna as lxm)     {DNA_LIT(lxm)}
| '#'(rna as lxm)     {RNA_LIT(lxm)}
| '#'(pep as lxm)     {PEP_LIT(lxm)}
| '#'(dna)([^';'] as char)  {raise (Failure("SCAN ERROR : illegal
character '" ^ Char.escaped char ^ "' in DNA sequence ."))}
| '#'(rnuc)+([^';'] as char)  {raise (Failure("SCAN ERROR : illegal
character '" ^ Char.escaped char ^ "' in RNA sequence ."))}
| '#'(aa '-')*([^';'] as char)   {raise (Failure("SCAN ERROR :
illegal character '" ^ Char.escaped char ^ "' in Peptide sequence
."))}
| eof { EOF }

| _ as char { raise (Failure("Main: illegal character " ^
Char.escaped char)) }

and comment = parse
 "*/" { token lexbuf }
| _  { comment lexbuf }

and singlecom = parse
 ['\n' '\r']    {token lexbuf}
| _  {singlecom lexbuf}

and stringparse= parse
 ';'   {token lexbuf}
| ['A''C'-'I''K'-'N''P'-'T''U'-'W''Y''a''c'-'i''k'-'n''p'-'t''u'-
'w''y'] as char    {raise(Failure("SCAN ERROR : amino acid " ^
Char.escaped char ^ "is not a sequence"))}
| _ as char {raise (Failure("SCAN ERROR : illegal character in
sequence " ^ Char.escaped char))}
```

Parser.mly

```
%{
open Ast
%}

%token ASSIGN
%token OR AND NOT NEG
%token LT LEQ EQ NEQ GEQ GT
%token PLUS MINUS TIMES DIVIDE MODULO EXPONENTIAL STRCAT
%token COMPLEMENT TRANSCRIBE  TRANSLATE  TRANSLATETWO
%token BEGIN END IF ELSEIF ELSE THEN FOR WHILE CONTINUE BREAK SIZEOF
FREAD READ CAST
%token NUC INT DOUBLE AA BOOL CHAR VOID STRING DNA RNA
%token CODON SEQUENCE PEPTIDE
%token TRUE FALSE
```

```
%token LPAREN RPAREN LBRACK RBRACK
%token SEMI COMMA COLON PERIOD
%token INCLUDE
%token RETURN EOF
%token LOCAL VARIABLE

%token <int> INT_LIT
%token <string> ID
%token <string> SEQUENCE_LIT
%token <char> CHAR_LIT
%token <char> NUC_LIT
%token <char> AA_LIT
%token <string> DNA_LIT
%token <string> RNA_LIT
%token <string> PEP_LIT
%token <float> DOUBLE_LIT
%token <string> STRING_LIT


%right ASSIGN
%left  OR
%left  AND
%left  LT       LEQ        EQ    NEQ  GEQ   GT
%left  PLUS     MINUS
%left  TIMES    DIVIDE     MODULO
%left  TRANSLATETWO
%left  TRANSLATE
%left  TRANSCRIBE
%left  EXPONENTIAL
%right NOT      COMPLEMENT   NEG
%left  LPAREN   RPAREN


%start program
%type <Ast.program> program

%%

program: decls EOF {$1}

decls:  {{pstmts = []; funds = [];}}
     |   decls stmt {{pstmts = $1.pstmts @ [$2] ; funcs =
$1.funcs;}}

     |   decls func_decl {{pstmts = $1.pstmts; funcs = $1.funcs @
[$2] }}

func_decl:
     typ ID LPAREN formals_opt RPAREN stmt_list END
     {{ typ = $1; fname = $2; formals = $4; stmts = List.rev $6;}}

formals_opt:
     /* nothing */ {[]}
     |   formal_list { List.rev $1 }

formal_list:
     typ ID {[($1, $2)]}
     |formal_list COMMA typ ID { ($3, $4) :: $1 }



typ:
     INT   {Int}
     |   BOOL   {Bool}
     |   VOID   {Void}
     |   CHAR   {Char}
     |   DOUBLE  {Double}
     |   AA     {Aa}
     |   NUC    {Nuc}
     |   CODON   {Codon}
     |   SEQUENCE {Seq}
     |   DNA    {DNA}
     |   RNA    {RNA}
     |   PEPTIDE {Pep}
     |   STRING  {Str}

stmt_list:
     /* nothing */  {[]}
     |   stmt_list stmt { $2 :: $1 }

stmt:
     expr SEMI   { Expr $1 }
     |   RETURN expr_opt SEMI   {Return $2 }
     |   BEGIN stmt_list END   {Block(List.rev $2)}
     |   FOR expr_opt SEMI expr SEMI expr_opt THEN stmt_list END {
For($2, $4, $6, Block(List.rev $8)) }
     |   WHILE expr THEN stmt_list END  { While($2, Block(List.rev
$4)) }
     |   IF expr THEN stmt_list bstmt END{ If($2, Block(List.rev
$4), $5) }
     |   typ ID ASSIGN expr SEMI{ VDecl($1, $2, $4)}
     |   typ LBRACK expr RBRACK ID SEMI { ArrayDecl($1,$3, $5)} /*
CHANGE LATER TO FORCE ASSIGNMENT */

bstmt:
     /* nothing */   {Nobranching}
     |   ELSEIF expr THEN stmt_list bstmt { Elseif($2,
Block(List.rev $4), $5) }
     |   ELSE stmt_list   { Else(Block(List.rev $2))}

expr:
     TRUE  { Litbool(true) }
     |   FALSE   { Litbool(false) }
     |   ID     { Id($1) }
     |   INT_LIT {Litint($1)}
     |   DOUBLE_LIT  { Litdouble($1) }
     |   SEQUENCE_LIT  { Sequence($1) }
     |   ID LBRACK expr RBRACK {ArrayAcc($1, $3)}
     |   DNA_LIT  { Litdna($1) }
     |   RNA_LIT  { Litrna($1) }
     |   PEP_LIT  { Litpep($1) }
     |   CHAR_LIT     { Litchar($1) }
     |   NUC_LIT      { Litnuc($1) }
     |   AA_LIT       { Litaa($1)}
     |   STRING_LIT  { Stringlit($1)}
     |   expr PLUS expr  {Binop($1,Add,$3)}
     |   expr MINUS expr {Binop($1,Sub,$3)}
     |   expr TIMES expr {Binop($1,Mult,$3)}
     |   expr DIVIDE expr{Binop($1,Div,$3)}
     |   expr MODULO expr{Binop($1,Mod,$3)}
     |   expr EXPONENTIAL expr {Binop($3,Expon,$1)}
     |   expr AND expr {Binop($1,And,$3)}
     |   expr OR expr{Binop($1,Or,$3)}
     |   expr EQ expr  {Binop($1,Equal,$3)}
     |   expr NEQ expr {Binop($1,Neq,$3)}
     |   expr LT expr {Binop($1,Less,$3)}
     |   expr LEQ expr{Binop($1,Leq,$3)}
     |   expr GT expr {Binop($1,Greater,$3)}
     |   expr GEQ expr{Binop($1,Geq,$3)}
     |   MINUS expr %prec NEG {Lunop(Neg, $2)}
     |   NOT expr {Lunop(Not, $2)}
     |   COMPLEMENT expr {Lunop(Comp, $2)}
     |   expr TRANSCRIBE {Runop($1, Transcb)}
     |   expr TRANSLATE  {Runop($1, Transltt)}
     |   expr TRANSLATETWO   {Runop($1, Transltwo)}
     |   ID ASSIGN expr  {Assign($1, $3)}
     |   ID LBRACK expr RBRACK ASSIGN expr {ArrayAssign($1,$3,$6)}
     |   LPAREN expr RPAREN  {$2}
     |   ID LPAREN actuals_opt RPAREN {Call($1, $3)}
     |   ID PERIOD SIZEOF {SizeOf($1)}
     |   FREAD LPAREN STRING_LIT RPAREN {Fread($3)}
     |   READ LPAREN STRING_LIT RPAREN {Read($3)}
     |   CAST LT typ GT LPAREN expr RPAREN {Cast($3, $6)}

expr_opt:
     /* nothing */   {Noexpr}
     |   expr   { $1 }

actuals_opt:
     /* nothing */ { [] }
     |   actuals_list { List.rev $1 }

actuals_list:
     expr { [$1] }
     |   actuals_list COMMA expr { $3 :: $1 }
```

Ast.ml

```

type uop = Neg | Not | Comp | Transcb | Translt | Transltwo

type op = Add | Sub | Mult | Div | Mod | And | Or | Equal |
           Neq | Less | Leq | Greater | Geq | Expon

type typ = Int | Bool | Void | Char | Double | Aa | Nuc | Codon | Seq
| Str | DNA | RNA | Pep | ArrayInt | ArrayDouble | ArrayStr |
ArrayBool | ArrayChar | ArrayAa | ArrayNuc | ArrayCodon | ArraySeq |
ArrayDNA | ArrayRNA | ArrayPep
```

```
type ending = End

type expr =
            Litint of int       (*added into Codegen*)
           | Litbool of bool          (*added into Codegen*)
           | Litchar of char          (*added into Codegen*)
           | Litnuc of char
           | Litaa of char
           | Id of string             (*added into Codegen*)
           | Litdna of string
           | Litrna of string
           | Litpep of string
           | LitCodon of string
           | Sequence of string
           | Stringlit of string
           | Litdouble of float
           | ArrayAcc of string * expr
           | Binop of expr * op * expr (*added into Codegen*)
           | Lunop of uop * expr            (*added into Codegen*)
           | Runop of expr * uop            (*added into Codegen*)
           | Assign of string * expr  (*added into Codegen*)
           | ArrayAssign of string * expr * expr      (* added to
the functions below *)
           | Call of string * expr list(*added into Codegen*)
           | SizeOf of string                                     (*
adds to the functions below *)
           | Fread of string                                      (*
adds to the functions below *)
           | Read of string                                  (* adds
to the functions below *)
           | Cast of typ * expr
           | Noexpr                        (*added into Codegen*)


type stmt =
            Block of stmt list
           | Return of expr
           | Expr of expr
           | If of expr * stmt * stmt
           | For of expr * expr * expr * stmt
           | While of expr * stmt
           | Elseif of expr * stmt * stmt
           | Else of stmt
           | VDecl of typ * string * expr
           | ArrayDecl of typ * expr * string    (* adds to the
functions below *)
           | Nobranching

type bind = typ * string

type func_decl = {
      typ        : typ;
      fname : string;
      formals    : bind list;
      stmts: stmt list;
}

type program =  {
      pstmts: stmt list;
      funcs: func_decl list;
}

(*printing functions*)
(*functions : Printing AST*)

(*print expr*)
let string_of_op = function
       Add       -> "+"
      | Sub           -> "-"
      | Mult          -> "*"
      | Div           -> "/"
      | Mod           -> "%"
      | And           -> "&"
      | Or            -> "|"
      | Equal    -> "=="
      | Neq           -> "!="
      | Less          -> "<"
      | Leq           -> "<="
      | Greater  -> ">"
      | Geq      -> ">="
      | Expon    -> "^"


let string_of_typ = function
       Int            ->"int"
      | Bool          ->"bool"
      | Void          ->"nul"
      | Char          ->"char"
      | Double   ->"double"
      | Aa       ->"aa"
      | Nuc           ->"nuc"
      | Codon    ->"codon"
      | Seq           ->"seq"
      | DNA      ->"DNA"
      | RNA      ->"RNA"
      | Pep      ->"Peptide"
      | Str       ->"str"
      | ArrayInt ->"Array of Int"
      | ArrayDouble   ->"Array of Double"
      | ArrayStr ->"Array of Str"
      | ArrayBool     ->"Array of Bool"
      | ArrayChar     ->"Array of Char"
      | ArrayAa  ->"Array of  Aa"
      | ArrayNuc ->"Array of  Nuc"
      | ArrayCodon    ->"Array of  Codon"
      | ArraySeq ->"Array of  Seq"
      | ArrayDNA ->"Array of  DNA"
      | ArrayRNA ->"Array of  RNA"
      | ArrayPep ->"Array of Pep"

let string_of_uop = function
```

```
       Neg       -> "-"
      | Comp           -> "@"
      | Transcb  -> "->"
      | Translt  -> "+>"
      | Transltwo-> "%>"
      | Not      -> "!"

let rec string_of_expr = function
       Litint(i) ->
           string_of_int i
      | Id(str)->
           str
      | Sequence(str)->
           str
      | Litdna(str) ->
           str
      | Litrna(str) ->
           str
      | Litpep(str) ->
           str
      | LitCodon(str)->
           str
      | Stringlit(str)->
           str
      | Litbool(true)->
           "true"
      | Litbool(false)->
           "false"
      | Litchar(ch)->
           String.make 1 ch
      | Litnuc(ch)->
           String.make 1 ch
      | Litaa(ch)->
           String.make 1 ch
      | Litdouble(flt)->
           string_of_float flt
      | Binop(exp1,op,exp2)->
           string_of_expr exp1 ^ string_of_op op ^
           string_of_expr exp2
      | Lunop(uop,exp)->
           string_of_uop uop ^ string_of_expr exp
      | Runop(exp,uop)->
           string_of_expr exp ^ string_of_uop uop
      | Assign(str,exp)->
           str ^ "=" ^ string_of_expr exp
      | ArrayAcc(str,exp)->
           str ^ "[" ^ string_of_expr exp ^ "]"
      | ArrayAssign(str,exp1,exp2)->
           str ^ "[" ^ string_of_expr exp1 ^ "]" ^ "=" ^
string_of_expr exp2
      | SizeOf(str)->
           str ^ ".length"
      | Cast(typ, expr)->
           string_of_typ typ ^ " " ^ string_of_expr expr
      | Fread(str)->
           "FASTA read " ^ str
      | Read(str)->
```

```
            "Read " ^ str
      | Call(str,l_expr)->
            str ^ "(" ^
            String.concat "," (List.map string_of_expr l_expr) ^
            ")"
      | Noexpr->
            ""


(*print stmt & stmt list*)
let rec string_of_stmt n stmt=
      let blk=String.make (n*2) ' ' in
      match stmt with
      |Block(l_stmt)->
            blk ^ "{\n" ^ string_of_stmt_list n l_stmt ^ blk ^ "}\n"

      |If(cond,l_stmt,next_stmt)->
            blk ^ "if->\n" ^
            blk ^ "  cond= (" ^ string_of_expr cond ^ ")\n" ^
            blk ^ "  body=\n" ^ string_of_stmt (n+2) l_stmt  ^
            string_of_stmt n next_stmt

      |For(strt,cond,step,l_stmt)->
            blk ^ "for->\n" ^
            blk ^ "  init= " ^ string_of_expr strt ^ "\n" ^
            blk ^ "  cond= " ^ string_of_expr cond ^ "\n" ^
            blk ^ "  step= " ^ string_of_expr step ^ "\n" ^
            blk ^ "  body=\n" ^ string_of_stmt (n+2) l_stmt

      |While(cond,l_stmt)->
            blk ^ "while->\n" ^
            blk ^ "  cond= (" ^ string_of_expr cond ^ ")\n" ^
            blk ^ "  body=\n" ^ string_of_stmt (n+2) l_stmt

      |Return(exp)->
            blk ^ "return->\n" ^
            blk ^ "  value= " ^ string_of_expr exp ^ "\n"

      |Expr(exp)->
            blk ^ "expr->\n" ^
            blk ^ "  value= " ^ string_of_expr exp ^ "\n"

      |Elseif(cond,l_stmt,b_stmt)->
            blk ^ "elseif->\n" ^
            blk ^ "  cond= (" ^ string_of_expr cond ^ ")\n" ^
            blk ^ "  body=\n" ^ string_of_stmt (n+2) l_stmt ^
            string_of_stmt n b_stmt

      |Else(l_stmt)->
            blk ^ "else->\n" ^
            blk ^ "  body=\n" ^ string_of_stmt (n+2) l_stmt

      |VDecl(typ,str,exp)->
            blk ^ "vari->\n" ^
            blk ^ "  " ^ string_of_typ typ ^ " " ^ str ^ " = " ^
string_of_expr exp ^ "\n"
```

```
     |ArrayDecl(typ,exp,str)->
          blk ^ "vari->\n" ^
          blk ^ "  " ^ string_of_typ typ ^ "[" ^ string_of_expr exp
^ "]" ^ str ^ "\n"
     |Nobranching->
          "\n"

and string_of_stmt_list n l_stmt=
     String.concat "" (List.map (string_of_stmt n) l_stmt)

let rec string_of_bind (typ,str)=
     string_of_typ typ ^ " " ^ str
and string_of_bind_list l_bid=
     "( "^String.concat ", " (List.map string_of_bind l_bid) ^ " )"

let string_of_func func_decl=
     let f_name=func_decl.fname in
     let f_typ=func_decl.typ in
     let f_form=func_decl.formals in
     let f_body=func_decl.stmts in
          string_of_typ f_typ ^ " function:" ^ f_name ^
string_of_bind_list f_form ^
          "\nbody->\n" ^ string_of_stmt_list 2 f_body ^ "\n"


let string_of_program program=
     let l_stm=program.pstmts in
     let l_fun=program.funcs in
          "----- Stmt List -----\n" ^ String.concat "" (List.map
(string_of_stmt 0) l_stm) ^ "\n" ^
          "----- Func List -----\n" ^ String.concat "" (List.map
string_of_func l_fun) ^ "\n"
```

Semant.ml

```
(* Semantic checking for #DNA compiler *)

open Ast

module StringMap = Map.Make(String)
(* Semantic checking of a program.
  Returns void if successful,throws an exception if something is
wrong.
  Check each global statement, then check each function
*)
(* Top-level functions - global checking functions *)


let globals_list = ref StringMap.empty;;
let locals_list = ref StringMap.empty;;
let count = ref true;;
let v_types_list = ["int"; "bool"; "char"; "double"; "aa"; "nuc";
"codon"; "seq"; "DNA"; "RNA"; "Peptide"; "str"];;
let types_map = List.fold_left (fun m (t) -> StringMap.add t true m)
StringMap.empty v_types_list;;
let cast_types_list = [ "seq"; "DNA"; "RNA"; "Peptide"; "str"];;
let cast_types_map = List.fold_left (fun m (t) -> StringMap.add t
true m) StringMap.empty cast_types_list;;



(* Raise an exception if a given binding is to a void type *)
let check_not_void exceptf = function
      (Void, n) -> raise (Failure (exceptf n))
      | _ -> ()
;;

let check_v_type t = try ignore(StringMap.find (string_of_typ t)
types_map)
      with Not_found -> raise (Failure ("unrecognized type " ^
(string_of_typ t)))
;;

let check_cast_type t = try ignore(StringMap.find (string_of_typ t)
cast_types_map)
      with Not_found -> raise (Failure ("invalid type " ^
(string_of_typ t) ^ " for cast."))
;;

let map_array_type t =
      match t with
      | Int -> ArrayInt
      | Double -> ArrayDouble
      | Str -> ArrayStr
      | Bool -> ArrayBool
      | Char -> ArrayChar
      | Aa -> ArrayAa
      | Nuc -> ArrayNuc
      | Codon -> ArrayCodon
      | Seq -> ArraySeq
      | DNA -> ArrayDNA
      | RNA -> ArrayRNA
      | Pep -> ArrayPep
      | _ -> ArrayInt
;;

let array_type_unfold t =
      match t with
      | ArrayInt -> Int
      | ArrayDouble -> Double
      | ArrayStr -> Str
      | ArrayBool -> Bool
      | ArrayChar -> Char
      | ArrayAa -> Aa
      | ArrayNuc -> Nuc
      | ArrayCodon -> Codon
      | ArraySeq -> Seq
      | ArrayDNA -> DNA
      | ArrayRNA -> RNA
      | ArrayPep -> Pep
      | Str -> Char
      | DNA -> Char
      | RNA -> Char
      | Pep -> Char
      | _ -> Int
;;

(* check for duplicates *)
let report_duplicate exceptf list =
      let rec helper = function
            n1 :: n2 :: __ when n1 = n2 -> raise (Failure (exceptf
n1))
            | _:: t -> helper t
            | [] -> ()
      in helper (List.sort compare list)
;;

(* Raise an exception of the given rvalue type cannot be assigned to
the given lvalue type *)

let check_assign lvaluet rvaluet err =
      match lvaluet with
       DNA -> if rvaluet == DNA || rvaluet == Seq then lvaluet else
raise err
      | RNA -> if rvaluet == RNA || rvaluet == Seq then lvaluet else
raise err
      | Char -> if rvaluet == Char || rvaluet == Aa || rvaluet == Nuc
then lvaluet else raise err
      | Aa -> if rvaluet == Aa || rvaluet == Nuc then lvaluet else
raise err
      | Pep -> if rvaluet == Pep then lvaluet else raise err
      | _ -> if lvaluet == rvaluet then lvaluet else raise err

(* function checking starts from here *)
(*  check user defined functions conflict with built-in functions
      function over loading to change this later  *)
let check_UDF_conflict funcs =
      if List.mem "print" (List.map (fun fd -> fd.fname) funcs)
      then raise (Failure ("function print may not be defined")) else
();
      report_duplicate (fun n -> "duplicate function " ^ n) (List.map
(fun fd -> fd.fname) funcs)
;;

let built_in_decls = StringMap.add "readFASTA"
    { typ = DNA; fname = "readFASTA"; formals = [(Str, "x")];
      stmts = [] } (StringMap.singleton "read"
    { typ = Str; fname = "read"; formals = [(Str, "x")];
      stmts = [] })
;;

let function_decl s function_decls = try StringMap.find s
function_decls
      with Not_found -> raise (Failure ("unrecognized function " ^
s))
;;
```

```
let check_stmt func function_decls =

     let type_of_identifier s =

          try StringMap.find s !(locals_list)
               with Not_found -> raise (Failure ("undeclared
identifier " ^ s))
     in

     let rec expr = function
          Litint _ -> Int
          | Litbool _ -> Bool
          | Litchar _ -> Char
          | Litnuc _ -> Nuc
          | Litaa _ -> Aa
          | Id s -> type_of_identifier s (* there is an issue for
order of initialization *)
          | Litdna _ -> DNA
          | Litrna _ -> RNA
          | Litpep _ -> Pep
          | LitCodon _ -> Str
          | Sequence _ -> Seq (* is this correct? *)
          | Stringlit _ -> Str (* is this correct?  *)
          | Litdouble _ -> Double (*is this correct? *)
          | ArrayAcc(s,e) -> let check_int_expr3 e =
                    if expr e != Int then raise (Failure ("expected
Integer expression in " ^ string_of_expr e))
                    else ()
               in check_int_expr3 e;
               array_type_unfold (type_of_identifier s)

(*        | Strcat(e1, e2) as ex -> let lt = expr e1
               and rt = expr e2 in
               check_assign lt rt (Failure ("illegal concatenation
" ^ string_of_typ lt ^
               " = " ^ string_of_typ rt ^ " in " ^
               string_of_expr ex))        *)
          | Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr
e2 in
          (match op with
               Add | Sub | Mult | Div | Expon when t1 = Int && t2 =
Int -> Int
               | Add | Sub | Mult | Div | Expon when t1 = Double &&
t2 = Double -> Double
               | Add when t1 =  Str && t2 = Str -> Str
               | Add when t1 =  Codon && t2 = Codon -> RNA
               | Add when t1 =  DNA && t2 = DNA -> DNA
               | Add when t1 =  RNA && t2 = RNA -> RNA
               | Add when t1 =  Seq && t2 = Seq -> Seq
               | Add when t1 =  DNA && t2 = Seq -> DNA
               | Add when t1 =  Seq && t2 = DNA -> DNA
               | Add when t1 =  RNA && t2 = Seq -> RNA
               | Add when t1 =  Seq && t2 = RNA -> RNA
               | Mod when t1 = Int && t2 = Int -> Int
               | Equal | Neq when t1 = t2 -> Bool
               | Less | Leq | Greater | Geq when t1 = Int && t2 =
Int -> Bool
                | And | Or when t1 = Bool && t2 = Bool -> Bool
                | _ -> raise (Failure ("illegal binary operator " ^
                     string_of_typ t1 ^ " " ^ string_of_op op ^ " "
^
                     string_of_typ t2 ^ " in " ^ string_of_expr e))
           )
           | Lunop(op, e) as ex -> let t = expr e in
           (match op with
                Neg when t = Int -> Int
                | Comp when t = Seq || t = DNA -> DNA
                | Not when t = Bool -> Bool
                | _ -> raise (Failure ("illegal left unary operator
" ^ string_of_uop op ^
                     string_of_typ t ^ " in " ^ string_of_expr ex)))
           | Runop(e, op) as ex -> let t = expr e in
           (match op with
                 Transcb when t = Seq || t = DNA -> RNA
                | Translt when t = Seq || t = RNA -> Pep
                | Transltwo when t = Seq || t = DNA -> Pep
                | _ -> raise (Failure ("illegal right unary operator
" ^ string_of_uop op ^
                     string_of_typ t ^ " in " ^ string_of_expr ex)))
           | Noexpr -> Void
           | Assign(var, e) as ex -> let lt = type_of_identifier var
           and rt = expr e in
           check_assign lt rt (Failure ("illegal assignment " ^
string_of_typ lt ^
                " = " ^ string_of_typ rt ^ " in " ^
           string_of_expr ex))
           | ArrayAssign(s,e1,e2) as ex -> let check_int_expr2 e =
                     if expr e != Int then raise (Failure ("expected
Integer expression in " ^ string_of_expr e))
                     else ()
                in check_int_expr2 e1;
                let lt = array_type_unfold (type_of_identifier s)
                and rt = expr e2 in
                check_assign lt rt (Failure ("illegal assignment " ^
string_of_typ lt ^
                     " = " ^ string_of_typ rt ^ " in " ^
                string_of_expr ex))
           | SizeOf(s) -> ignore(type_of_identifier s); Int
           | Cast(t,e) -> ignore(check_cast_type t);
                               ignore(check_cast_type (expr e));
                               t
           | Fread(s) -> DNA
           | Read(s) -> Str
           | Call(fname, actuals) as call -> if fname = "print" then
(
                     if List.length actuals != 1 then
                          raise (Failure ("expecting " ^
string_of_int 1 ^ " argument in print function call."))
                     else
                          List.iter (fun e -> ignore (expr e))
actuals;
                          Void
                )
```

```
                 else( let fd = function_decl fname function_decls in
                      if List.length actuals != List.length
fd.formals then
                           raise (Failure ("expecting " ^
string_of_int
                           (List.length fd.formals) ^ " arguments in
" ^ string_of_expr call))
                      else
                           List.iter2 (fun (ft, _) e -> let et =
expr e in
                           ignore (check_assign ft et
                                (Failure ("illegal actual argument
found " ^ string_of_typ et ^
                                " expected " ^ string_of_typ ft ^ "
in " ^ string_of_expr e))))
                                fd.formals actuals;
                                fd.typ     )

           in   (* end of expression checking *)
           (* check whether an expression is Boolean *)
           let check_bool_expr e =
                 if expr e != Bool then raise (Failure ("expected
Boolean expression in " ^ string_of_expr e))
                 else ()
           in
           (* check whether an expression is Integer *)
           let check_int_expr e =
                 if expr e != Int then raise (Failure ("expected
Integer expression in " ^ string_of_expr e))
                 else ()
           in

           (* stmt checking *)
           let rec stmt = function
                 Block sl -> let rec check_block = function
                      [Return _ as s] -> stmt s
                      | Return _ :: _ -> raise (Failure "nothing may
follow a return")
                      | Block sl :: ss -> check_block (sl @ ss)
                      | s :: ss -> stmt s ; check_block ss
                      | [] -> ()
                 in check_block sl
                 | Expr e -> ignore (expr e)
                 | Return e -> let t = expr e in if t = func.typ then
() else
                 raise (Failure ("return gives " ^ string_of_typ t ^
" expected " ^
                      string_of_typ func.typ ^ " in " ^
string_of_expr e))
                 | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt
b2
                 | For(e1, e2, e3, st) -> ignore (expr e1);
check_bool_expr e2;
                 ignore (expr e3); stmt st
                 | While(p, s) -> check_bool_expr p; stmt s
                 | Elseif (p, s1, s2) -> check_bool_expr p; stmt s1;
stmt s2
```

```
                | Else (st) -> stmt st
                | VDecl(t, s, e) ->
                        ignore(check_v_type t);
                        (locals_list) := StringMap.add s t
!(locals_list);
                        ignore(let lt = t and rt = expr e in
check_assign lt rt (Failure ("illegal assignment " ^ string_of_typ lt
^
                              " = " ^ string_of_typ rt ^ " in " ^ s ^"
= "^
                        string_of_expr e)))
                | ArrayDecl(t, e, s) -> ignore(check_v_type t);
                (locals_list) := StringMap.add s (map_array_type t)
!(locals_list);
                check_int_expr e
                | Nobranching -> ()

        in

        stmt (Block func.stmts)
;;

let check_func func function_decls =

     List.iter (check_not_void (fun n -> "illegal void formal " ^ n
^ " in " ^ func.fname)) func.formals;

     report_duplicate (fun n -> "duplicate formal " ^ n ^ " in " ^
func.fname) (List.map snd func.formals);

     (locals_list) := List.fold_left (fun m (t, n) -> StringMap.add
n t m) StringMap.empty func.formals;
(*
     let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t
m) StringMap.empty func.formals
     in
*)
     check_stmt func function_decls;
     if !(count) then (
          (globals_list) := !(locals_list);
          (count) := false;     )
     else
          (locals_list) := StringMap.empty
;;


let check prog =
     check_UDF_conflict prog.funcs;
     let func_decls = List.fold_left (fun m fd -> StringMap.add
fd.fname fd m) built_in_decls prog.funcs
     in
          let dummy_func = {
               typ = Void;
               fname = "dummy_func";
               formals = [];
               stmts = prog.pstmts
               } in
```

```
           check_stmt dummy_func func_decls;
           List.iter (fun n -> check_func n func_decls) prog.funcs
```

Linker.ml

```
open Llvm
open Llvm.MemoryBuffer
open Llvm_bitreader

let c_lib_path="lib/c_lib.bc"

let link_bc modu filename =
      let llctx = Llvm.global_context () in
   let llmem = Llvm.MemoryBuffer.of_file filename in
   let llm = Llvm_bitreader.parse_bitcode llctx llmem in
      ignore(Llvm_linker.link_modules' modu llm)
```

C_lib.c

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>

#define EPS 0.000001

char codon[64] = {'K', 'D', 'K', 'D', 'T', 'T', 'T', 'T', 'R', 'S',
'R', 'S', 'I', 'I', 'M', 'I', 'Q', 'H', 'Q', 'H', 'P', 'P', 'P', 'P',
'R', 'R', 'R', 'R', 'L', 'L', 'L', 'L', 'E', 'D', 'E', 'D', 'A', 'A',
'A', 'A', 'G', 'G', 'G', 'G', 'V', 'V', 'V', 'V', 'B', 'Y', 'B', 'Y',
'S', 'S', 'S', 'S', 'B', 'C', 'W', 'C', 'L', 'F', 'L', 'F' };

int test(int a,int b)
{
   printf("Hello I'm in C\n");
   return a+b;
}

char* complement(char* str){
   if(!str)
      return str;
      int i = 0;
      while(str[i] != 0) i++;
      int length = i;
      char* retstr = malloc(length);

      char curr;
      for(int i = 0; i<length; i++){
      curr = str[i];
      if(curr == 'A' || curr == 'a')
           retstr[i] = 'T';
      else if(curr == 'T'|| curr == 't')
           retstr[i] = 'A';
      else if(curr == 'G' || curr == 'g')
           retstr[i] = 'C';
      else if(curr == 'C' || curr == 'c')
           retstr[i] = 'G';
       else
            retstr[i] = 'G';
      }
      retstr[length] = '\0';
      return retstr;
}

char* transcribe(char* str){
   if(!str)
      return str;
      int i = 0;
      while(str[i] != '\0'){
       i++;
   }
      int length = i;
   char* retstr = malloc(length);
      char curr;
      for(i = 0; i<length; i++){
      curr = str[i];
       if(curr == 'A' || curr == 'a')
           retstr[i] = 'U';
      else if(curr == 'T'|| curr == 't')
           retstr[i] = 'A';
      else if(curr == 'G' || curr == 'g')
           retstr[i] = 'C';
      else if(curr == 'C' || curr == 'c')
           retstr[i] = 'G';

      }

      retstr[length] = '\0';
      return retstr;
}

int char_num(char a){
      int ret = 0;
      if(a == 'A' || a == 'a')
      ret = 0;
      else if(a == 'C' || a == 'c')
      ret = 1;
      else if(a == 'G' || a == 'g')
      ret = 2;
      else if(a == 'U' || a == 'u')
      ret = 3;
      return ret;
}
int codon_number(char a, char b, char c){
   int ret = char_num(a) * 16 + char_num(b) * 4 + char_num(c);
   //if(ret == 0)
   //    printf("HEHEHEHEHEHH->%c%c%c\n", a, b, c);
      return ret;
}

char* translate(char* str){
   if(!str)
      return str;
      int i = 0;
      while(str[i] != 0) i++;
      int length = i;
      int start[3];
      int ending[3];
      for(i=0; i<3; i++){
      start[i] = -1;
      ending[i] = -1;
      }
      int temp=0;
      for(i = 0; i<length-2; i++){
      temp = codon_number(str[i], str[i+1], str[i+2]);
      if(temp == 14 || temp == 46){
           if(start[i%3] == -1)
                start[i%3] = i;
      }
      if(temp == 48 || temp == 50 || temp == 56){
           if(start[i%3] != -1){
                if(ending[i%3] == -1 && i>start[i%3])
                ending[i%3] = i;
           }
      }
      }
      temp = -1;
      for(i = 0; i<3; i++){
      if(start[i] != -1 && ending[i] != -1){
            if(temp != -1 && start[i]<start[temp])
                 temp = i;
            if(temp == -1)
                 temp = i;
      }
      }
      char* retdef = "No possible translation available.";
      if(temp == -1)
      return retdef;
      int b = ending[temp];
      int a = start[temp];
      int ret_size = (b-a)/3 + 1;
      char* retstr = malloc(ret_size);
      int index;
      for(int i = a; i<b; i=i+3){
      temp = codon_number(str[i], str[i+1], str[i+2]);
      index = (i-a)/3;
      retstr[index] = codon[temp];
      }
   retstr[ret_size-1] = '\0';
   int length2 = ret_size - 1;
   char retstr2 [length2];
      int curr;
      for(int i = 0; i<length2; i++){
```

```
      curr = retstr[i];
      retstr2[i] = curr;
      }
      retstr2[length2] = '\0';
      return retstr;
}

char* translate2(char* str){
   char* str1 = transcribe(str);
   char* str2 = translate(str1);
   return str2;
}

char* concat(char * input1, char * input2)
{
      int i= 0;
      while(input1[i] != 0) i++;
      int j = 0;
      while(input2[j] != 0)j++;
      int length = i + j;
      char retstr [length];
      for (int k = 0; k < i; k++)
      {
      retstr[k] = input1[k];
      }
      for (int l =0; l <i; l++)
      {
      retstr[i + l] = input2[l];
      }
      retstr[length] = '\0';
      return retstr;
}
int strlength(char * input)
{
      int i =0;
      while (input[i] != 0) i++;
      return i;
}
char* readFASTAFile(char * string) //currently only supports single
sequence FASTA of char length < 10000
{
      //Reference for code help:
http://stackoverflow.com/questions/4823177/reading-a-file-character-
by-character-in-c
      FILE *file = fopen(string, "r");
      char code [10000]; // NOTE: Maybe decrease size???
      size_t n = 0;
      int c;
      int start = -1;
      if (file == NULL)
      return NULL; //could not open file
      while ((c = fgetc(file)) != EOF) // NOTE: Should we make
capital or make lowercase?
      {
      if (((char) c) == '\n')
      {
            start = 1;
            continue;
      }
      if ((start >= 0) && ((((char) c) == 'a') || (((char) c) == 't')
|| (((char) c) == 'g') || (((char) c) == 'c') || (((char) c) == 'A')
|| (((char) c) == 'T') || (((char) c) == 'G') ||(((char) c) == 'C'))
)
      {
      code[n++] = (char) c;
      }
      }

      // don't forget to terminate with the null character
      code[n] = '\0';
      return code;
}
char* readFile(char * string) //currently only supports single
sequence FASTA of char length < 10000
{
      //Reference for code help:
http://stackoverflow.com/questions/4823177/reading-a-file-character-
by-character-in-c
      FILE *file = fopen(string, "r");
      char code [10000]; // NOTE: Maybe decrease size???
      size_t n = 0;
      int c;
      int start = -1;
      if (file == NULL)
      return NULL; //could not open file
      while ((c = fgetc(file)) != EOF) // NOTE: Should we make
capital or make lowercase?
      {

      code[n++] = (char) c;
      }

      // don't forget to terminate with the null character
      code[n] = '\0';
      return code;
}

int mod(int a,int b)
{
      return a%b;
}

int exp_ii(int a,int b)
{
      return pow(a,b);
}

double exp_di(double a,int b)
{
      return pow(a,b);
}

double exp_id(int a,double b)
{
     return pow(a,b);
}

double exp_dd(double a,double b)
{
     return pow(a,b);
}


int double2int(double d)
{
     return (int)d;
}

double int2double(int i)
{
     return (double)i;
}

int print_tf(bool b)
{
     if(b){
     return printf("true\n");
     }
     else{
     return printf("false\n");
     }
}
char getChar (char ** input, int index)
{
     if (input == NULL)
     return -1;
     if (input[0][0] == NULL)
     return -1;
     int i = 0;//length of input
     while (input[0][i] != 0) i++;
     if (index >= i)
     {
     return -1;// check for type casting
     }
     char temp =input[0][index];
     return temp;
}
char* formatPep(char * input)
{
     int i= 0;
     while(input[i] != 0) i++;
     char result [i];
     int j = 0;
     for (int k=0; k<i; k++)
     {
     if (input[k] != '-')
     {
          result[j] = input[k];
          j++;
     }
     }
```

```
      while (j <= i)
      {
      result[j] = 0;
      j++;
      }
      return result;

}

int printPep(char * input)
{
      int i= 0;
      while(input[i] != 0) i++;
      for (int j=0; j<(i-1); j++)
      {
      char temp = (char) input[j];
      printf("%c", temp);
      printf("-");
      }
      printf("%c", input[i-1]);
      return 1;

}
bool testValid(char * input, char type)
{
      int size = 0;
      while(input[size] != 0) size++;
      if (type == 'd')// dna type checking
      {
      for (int i =0; i< size; i++)
      {
            if ((input[i] != 'a') && (input[i] != 't') && (input[i] !=
'g')  && (input[i] != 'c')  && (input[i] != 'A')  && (input[i] !=
'T')  && (input[i] != 'G')
            && (input[i] != 'C'))
            {
                  return 0;
            }
      }
      }
      if (type == 'r')// rna type checking
      {
      for (int i =0; i< size; i++)
      {
            if ((input[i] != 'a') && (input[i] != 'u') && (input[i] !=
'g')  && (input[i] != 'c')  && (input[i] != 'A')  && (input[i] !=
'U')  && (input[i] != 'G')
            && (input[i] != 'C'))
                  return 0;
      }
      }
      if (type == 'p')// pep type checking
      {
      for (int i =0; i< size; i++)
      {
             for (int j = 0; j<64; j++){
                   if(input[i] == codon[j])
```

```
                        break;
                   return 0;
              }
      }
      }
      return 1;
}
int release_memory(char* a){
      free(a);
      return 1;
}
```

runDNAs.sh

```
#!/bin/bash
# Run DNA#
# If the terminal shows "can't find command", plz input "chmod +x
runDNAs.sh" in terminal first.

LLI="lli"
#LLI="/usr/local/opt/llvm/bin/lli"

DNAS="./DNAs.native"
#DNAs="_build/DNAs.native"

DEFAULT_PATH="."

Run() {
      eval $*
}

Usage() {
      echo "Usage: testall.sh [options] [.dnas files]"
      echo "-a   Print AST of file"
      echo "-c   Compile file"
      echo "-r   Run file"
      echo "-h   Print this help"
      exit 1
}


GenerateAst(){
      basename=`echo $1 | sed 's/.*\\///
                           s/.dnas//'`
      echo "# generating AST of ${basename}.dnas ..."
      Run "$DNAS -a" "<" $1 ">" "${DEFAULT_PATH}/${basename}.ast"
      echo "     ${basename}.ast ... Done"
}

GenerateLlvm (){
      basename=`echo $1 | sed 's/.*\\///
                           s/.dnas//'`
      echo "# generating Llvm-IR of ${basename}.dnas ..."
      Run "$DNAS -l" "<" $1 ">" "${DEFAULT_PATH}/${basename}.ll"
      echo "     ${basename}.ll ... Done"
}

CompileFile(){
      basename=`echo $1 | sed 's/.*\\///
                           s/.dnas//'`
      echo "# compiling ${basename}.dnas"
      Run "$DNAS" "<" $1 ">" "${DEFAULT_PATH}/${basename}.ll"
      echo "     ${basename}.ll ... Done"
      Run "$LLI" "${DEFAULT_PATH}/${basename}.ll" ">"
"${DEFAULT_PATH}/${basename}.out"
      echo "     ${basename}.out ... Done"

}

RunProgram(){
      basename=`echo $1 | sed 's/.*\\///
                           s/.dnas//'`

      Run "$DNAS" "<" $1 ">" "${DEFAULT_PATH}/${basename}.ll"

      echo "# Executing ${basename}.dnas ..."
      Run "$LLI" "${DEFAULT_PATH}/${basename}.ll"

}

#entrance point of shell
MODE="Compile"

while getopts acrlh c; do
      case $c in
   a) # Ast
      MODE="Ast"
      ;;
   c) # Compile
       MODE="Compile"
       ;;
   r) # Run
       MODE="Run"
       ;;
      l) #LLVM-IR
      MODE="Llvm"
      ;;
   h) # Help
      Usage
      ;;
      esac
done

shift `expr $OPTIND - 1`

if [ $# -ge 1 ]
then
      files="$@"
else
      files="*.dnas"
```

```
fi

for file in $files
do
      case $MODE in
   Ast)
      GenerateAst $file
      ;;
      Llvm)
      GenerateLlvm $file
      ;;
   Compile)
      CompileFile $file
      ;;
   Run)
      RunProgram $file
      ;;
      esac
done
```

test_DNAs.sh

```
#!/bin/sh

#regression testing suit for DNA#

# path to the llvm interpreter
LLI="lli"

# path to DNA# compiler
DNAC="./DNAs.native"

# set the time limit
ulimit -t 30

globallog=test_DNAs.log
rm -f $globallog
error=0
globalerror=0

keep=0
passed=0
failed=0

Usage() {
      echo "Usage:test_DNAs.sh [options] [.dnas files]"
      echo "-k Keep intermediate files"
      echo "-h print this help"
      exit 1
}

SignalError() {
      if [ $error -eq 0 ] ; then
      echo "Failed"
      failed=$((failed+1))
      error = 1
      fi
      echo "  $1"
}

# Compare <outfile> <reffile> <difffile>
# Compares the outfile with reffile.  Differences, if any, written to
difffile
Compare() {
      generatedfiles="$generatedfiles $3"
      echo diff -b $1 $2 ">" $3 1>&2
      diff -b "$1" "$2" > "$3" 2>&1 || {
   SignalError "$1 differs"
   echo "FAILED $1 differs from $2" 1>&2
      }
}

# Run <args>
# Report the command, run it, and report any errors
Run() {
      echo $* 1>&2
      eval $* || {
   SignalError "$1 failed on $*"
   return 1
      }
}

# RunFail <args>
# Report the command, run it, and expect an error
RunFail() {
      echo $* 1>&2
      eval $* && {
   SignalError "failed: $* did not report an error"
   return 1
      }
      return 0
}

Check() {
      error=0
      basename=`echo $1 | sed 's/.*\\///
                              s/.dnas//'`
      reffile=`echo $1 | sed 's/.dnas$//'`
      basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

      echo -n "$basename..."

      echo 1>&2
      echo "###### Testing $basename" 1>&2

      generatedfiles=""

      generatedfiles="$generatedfiles ${basename}.ll ${basename}.out"
&&
      Run "$DNAC" "<" $1 ">" "${basename}.ll" &&
      Run "$LLI" "${basename}.ll" ">" "${basename}.out" &&
      Compare ${basename}.out ${reffile}.out ${basename}.diff

      # Report the status and clean up the generated files

      if [ $error -eq 0 ] ; then
   if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
   fi
   echo "OK"
   echo "###### SUCCESS" 1>&2
   passed=$((passed+1))
      else
   echo "###### FAILED" 1>&2
   globalerror=$error
      fi
}

CheckFail() {
      error=0
      basename=`echo $1 | sed 's/.*\\///
                             s/.dnas//'`
      reffile=`echo $1 | sed 's/.dnas$//'`
      basedir="`echo $1 | sed 's/\/[^\/]*$//'`/."

      echo -n "$basename..."

      echo 1>&2
      echo "###### Testing $basename" 1>&2

      generatedfiles=""

      generatedfiles="$generatedfiles ${basename}.err
${basename}.diff" &&
      RunFail "$DNAC" "<" $1 "2>" "${basename}.err" ">>" $globallog
&&
      Compare ${basename}.err ${reffile}.err ${basename}.diff

      # Report the status and clean up the generated files

      if [ $error -eq 0 ] ; then
   if [ $keep -eq 0 ] ; then
      rm -f $generatedfiles
   fi
   echo "OK"
   echo "###### SUCCESS" 1>&2
   failed=$((failed+1))

      else
   echo "###### FAILED" 1>&2
   globalerror=$error

      fi
}

while getopts kdpsh c; do
      case $c in
   k) # Keep intermediate files
```

```
      keep=1
      ;;
   h) # Help
      Usage
      ;;
      esac
done

shift `expr $OPTIND - 1`

LLIFail() {
 echo "Could not find the LLVM interpreter \"$LLI\"."
 echo "Check your LLVM installation and/or modify the LLI variable in
testall.sh"
 exit 1
}

which "$LLI" >> $globallog || LLIFail


if [ $# -ge 1 ]
then
      files=$@
else
      files="dnas_tests/test-*.dnas dnas_tests/fail-*.dnas"
fi

for file in $files
do
      case $file in
   *test-*)
      Check $file 2>> $globallog
      ;;
   *fail-*)
      CheckFail $file 2>> $globallog
      ;;
   *)
      echo "unknown file type $file"
      globalerror=1
      ;;
      esac
done
echo "# of test cases run: " $passed
echo "# of test cases failed: " $failed

exit $globalerror
```

Makefile

```
DNAS_SRC=src/
DNAS_LIB=lib/
TARGET=DNAs
LIBS=-I,/usr/lib/ocaml/
```

```
FLAGS= -use-ocamlfind -pkgs
llvm,llvm.analysis,llvm.bitwriter,llvm.bitreader,llvm.linker,llvm.target
-cflags -w,+a-4
OCAMLBUILD=ocamlbuild
OPAM=opam config env


.PHONY: master.all native byte depend

all: clib native
   @ echo "Finished!"

clib:
   @ echo "## Building Extern C Lib"
   @ clang -c -emit-llvm $(DNAS_LIB)/c_lib.c -lm
   @ mv c_lib.bc $(DNAS_LIB)/c_lib.bc

native:
   @ echo "## Building DNAs.native"
   @ eval `opam config env`
   @ $(OCAMLBUILD) $(FLAGS) $(DNAS_SRC)/$(TARGET).native

byte:
   @ cd src
   @ $(OCAMLBUILD) $(FLAGS) $(TARGET).byte
   @ cd ..

depend:
   @ echo "Not needed."


# "make clean" removes all generated files
.PHONY : clean

clean :
   ocamlbuild -clean
   rm -rf test_DNAs.log *.diff DNAs scanner.ml parser.ml parser.mli
   rm -rf *.cmx *.cmi *.cmo *.cmx *.o *.ll
```

README

```
The DNA# compiler

Coded in OCaml, DNA# is a biology computational language. It has
major common data types plus nucleotde, amino acid, DNA, RNA, and
peptide types. Key functions include transcribe and translate
functions, which mimic the biological process. It is compiled into
LLVM IR.

It needs the OCaml llvm library, which is most easily installed
through opam.

Install LLVM and its development libraries, the m4 macro
preprocessor, and opam, then use opam to install llvm.
```

```
The version of the OCaml llvm library should match the version of the
LLVM system installed on your system.

It also requires clang compiler for the c built-in functions inside
the compiler.
( to install clang: sudo apt install clang )

To run your dna files using the following command:
make  (to build the compiler first)
./runDNAs.sh -r filename.dnas

<\t>-a   Print AST of file
<\t>-c   Compile file
<\t>-r   Run file
<\t>-h   Print this help

If the terminal shows "can't find command",feed "chmod +x runDNAs.sh"
into the terminal first.

To run all the test files, run the following command:
./test_DNAs.sh

Clean up all the built files/dependencies -> make clean
```

10 TESTS

10.1 Tests to Pass

Test 1 Declaring bool:

```
bool hi = true;
print(hi);
bool hello = false;
print (hello);
```

Test 2 Declaring Peptide:

```
Pep sample = #A-T-G-C;
RNA test  = #;
Pep final = test+>;
print(final);
```

Test 3 While and For Loop:

```
int i =0;
int j =0;
int b = 7;
int c = 6;
while (i +1)<c then
 print(i);
 i=i+1;
 for j=0; j<7; j=j+1 then
      print(j);
 end
end
```

Test 4 Declaring Char:

```
DNA hi = #atgc;
char temp = hi[5];
print(temp);
```

Test 5 Declaring String:

```
string hello = "krieger";
char temp = hello[2];
print(temp);
```

Test 6 Printing Bool:

```
bool hi = true;
print(hi);
bool hello = false;
print (hello);
```

Test 7 Indexing String:

```
string hello = "krieger";
char temp = hello[1];
print(temp);
```

Test 8 Translate 1:

```
Pep sample = #ATGC;
RNA test  = #;
Pep final = test+>;
print(final);
```

Test 9 Casting:

```
DNA hi = #atgc;
```

```
string boo = cast<string>(hi);
print(boo);
```

Test 10 Add Integers:

```
int adder (int a, int b, int c)
c = b +a;
return c;
end

print(adder(6,7,0));
```

Test 11 Assigning Char:

```
char a = 'B';
```

Test 12 Declaring Function:

```
int sample(int a, int b)
 int a=4;
 end
```

Test 13 Testing For Loop:

```
int i =0 ;
int b = 4;
int c = 6;
for i = 0; i<b*c; i=i+1 then
 print(i);
 end
/*Fatal error: exception Failure("expected Boolean expression in
b+1")*/
```

Test 14 Testing For Loop:

```
int i =0 ;
int b = 4;
int c = 6;
for i = 0; i<b*c; i=i+1 then
 print(i);
 end
/*Fatal error: exception Failure("expected Boolean expression in
b+1")*/
```

Test 15 Indexing DNA:

```
DNA hello = #atgc;
char temp = hello[2];
print(temp);
```

Test 16 Declaring DNA and RNA:

```
DNA sampA=#ATGC;
RNA sampB=#AUGC;
```

Test 17 Indexing DNA:

```
DNA hi = #atgc;
char temp = hi[5];
print(temp);
```

Test 18 Comments:

```
int i=0;
int j=0;
int k=0;
int cnt=0;

while i<10 then
   // print("# i loop#");
   i=i+1;
   int k=i;
   for j=i;j<10;j=j+1 then
       int k=i*j;
       cnt=cnt+1;
       // print(k);
   end
   // print("# i loop scope # Value of k:");
   // print(k);
end
/*
print("# global scope # Value of k:");
   print(k);

print("# global scope # Value of cnt:");
   print(cnt);
*/
```

Test 19 DNA Declaration:

```
DNA [5] myarray;
myarray[0] = #AGTC;
DNA test = myarray[0];
print(myarray);
print(test);
```

Test 20 Declaring Peptide:

```
Pep sample = #A-T-G-C;
print(sample);
```

Test 21 Arithmetic and Exponential:

```
int [4] array;
array[3] = 3;
print(array[3]);
print(array.length);
print("hello "+"world!");
print(1 + 2);
print(2.5/0.5);
print(2^4);
print(2.5^2.0);
```

Test 22 DNA Declaration:

```
DNA hello = #atgc;
```

Test 23 DNA# Data Types:

```
Pep oriana = #A-T-G-C;
int hi = 42;
double hello = 42.0;
bool isTrue = true;
string sample = "hello";
char samp = sample[0];
DNA min = #atgc;
RNA stan = #gcgc;
print(hi);
print(isTrue);
print(sample);
print(samp);
print(min);
print(hello);
print(stan);
print(oriana);
```

Test 24 Declaring String:

```
string hi = "hello";
print(false);
print(hi);
```

Test 25 Declaring Functions :

```
int f1()

end
```

Test 26 Array of Strings:

```
string [8] David;
David[0] = "Lose ";
David[1] = "yourself";
David[2] = "to ";
David[3] = "dance";
int i = 0;
for i=0;i<4;i=i+1 then
   print(David[i]);
end
```

Test 27 Declaring DNA and Pep:

```
/*  %%;  parsing need to catch an error spit out the wrong input. */
DNA r1 = #tatgcct;
DNA r1 = #agg;
Pep p1 = #a-k-w;
int i = 1;
//print(d1);
print(r1);
print(p1);
```

Test 28 Seq declaration:

```
Seq hi = #agcgcaagcca;
print(hi);
```

Test 29 Declaring String:

```
string hi = "hello";
print(hi);
print("hi");
hi = "boo";
print(hi);
```

Test 30 For Loop:

```
int i=1;
for i=1;i<2;i=i+1 then
   print("Hello World !");
```

```
end


while i>1 then
   i=i-1;
   print("What is next?");
end
```

Test 31 Printing String:

```
int i=1;
for i=1;i<2;i=i+1 then
   print("Hello World !");
end


while i>1 then
   i=i-1;
   print("What is next?");
end
```

Test 32 Declare nuc:

```
int i=1;
i = i+1;
//print(i);

int func(int A, int B)
     int c = A + B;
     c = A+1;
     return c;
end

for i =1; i<10; i=i+1 then

end

nuc a = 't';
//char a = 't';
// aa a = 't';
print(a);
```

Test 33 Print Number:

```
print(2);
```

Test 34 Casting:

```
string sample = "incorrect";
```

```
DNA test = cast<DNA>(sample);
```

Test 35 Declare DNA:

```
DNA hi = #atgc;
int l1 = 1;
string boo = cast<string>(hi);
//DNA d1 = cast<DNA>(l1);
//string boo = cast<int>(hi);
//string boo = cast<DNA>(hi);
print(hi);
```

Test 36 Index String:

```
string hello = "krieger";
char temp = hello[2];
print(temp);
```

Test 37 Index String:

```
string hello = "krieger";
char temp = hello[1];
print(temp);
```

10.2 Tests to Fail

Test 38:

```
int print(int a, int b, double c)
 a = b +1;
end
```

Test 39:

```
string sample = "incorrect";
DNA test = cast<DNA>(sample);
```

Test 40:

```
void a = Void;
```

Test 41:

```
int i =0 ;
i = true;
/*Fatal error: exception Failure("illegal assignment int = bool in
i=true")*/
```

Test 42:

```
int i =0 ;
int b = 4;
int c = 6;
while i +1 then
 print(i);
 end
/*Fatal error: exception Failure("expected Boolean expression in
i+1")*/
```

Test 43:

```
int i =0 ;
int b = 4;
int c = 6;
for i = 0; i+1; i=i+1 then
 print(i);
 end
/*Fatal error: exception Failure("expected Boolean expression in
i+1")*/
```

Test 44:

```
int b = 4;
int c = 6;
if b +1 then
 b = b +1;
else
 c = b +c;
end
/*Fatal error: exception Failure("expected Boolean expression in
b+1")*/
```

Test 45:

```
int [5] myarray;
myarray [5.6] = 5;

/*Fatal error: exception Failure("expected Integer expression in
5.6")*/
```

Test 46:

```
int [5.5] myarray;
/*Fatal error: exception Failure("expected Integer expression in 5.5")
*/
```

Test 47:

```
double sample(int a, double b, double c)
 b = 5.0;
 c = 2.5;
 double z = b + c;
 return z;
 a= 5;
 end
 /*Fatal error: exception Failure("nothing may follow a return")*/
```

Test 48:

```
int sample(int a, double b, double c)
 b = 5.0;
 c = 2.5;
 double z = b + c;
 return z;
 end
 /*Fatal error: exception Failure("return gives double expected int in
z")
*/
```

Test 49:

```
 int i = 42;
 i = 10;
 int b = true;
 b = false;
 i = false; /* Fail: assigning a bool to an integer */
```

Test 50:

```
string hi = "abc";
string hello = "def";
string sample = "atgc";
DNA another= sample->;
print(sample.length);
```

Test 51:

```
string hi = "abc";
string hello = "def";
string sample = "atgc";
DNA another= sample->;
print(sample.length);
```

Test 52:

```
int sample(int a, int a, double c)
 a = b +1;
end
```

Test 53:

```
int sample(int a, int a, double c)
 a = b +1;
end
```

Test 54:

```
int print(int a, int b, double c)
 a = b +1;
end
```

Test 55:

```
RNA r1 =
#AAAAAAAAAAAAAAAGGCAGAUUCCCCCUAGACCCGCCCGCACCAUGGUCAGGCAUGCCCCUCCUCAUC
GCUGGGCACAGCCCAGAGGGUAUAAACAGUGCUGGAGGCUGGCGGGGCAGGCCAGCUGAGUCCUGAGCAG
CAGCCCAGCGCAGCCACCGAGACACCAUGAGAGCCCUCACACUCCUCGCCCUAUUGGCCCUGGCCGCACU
UUGCAUCGCUGGCCAGGCAGGUGAGUGCCCCCACCUCCCCUCAGGCCGCAUUGCAGUGGGGGCUGAGAGG
AGGAAGCACCAUGGCCCACCUCUUCUCACCCCUUUGGCUGGCAGUCCCUUUGCAGUCUAACCACCUUGUU
GCAGGCUCAAUCCAUUUGCCCCAGCUCUGCCCUUGCAGAGGGAGAGGAGGGAAGAGCAAGCUGCCCGAGA
CGCAGGGGAAGGAGGAUGAGGGCCCUGGGGAUGAGCUGGGGUGAACCAGGCUCCCUUUCCUUUGCAGGUG
CGAAGCCCAGCGGUGCAGAGUCCAGCAAAGGUGCAGGUAUGAGGAUGGACCUGAUGGGUUCCUGGACCCU
CCCCUCUCACCCUGGUCCCUCAGUCUCAUUCCCCCACUCCUGCCACCUCCUGUCUGGCCAUCAGGAAGGC
CAGCCUGCUCCCCACCUGAUCCUCCCAAACCCAGAGCCACCUGAUGCCUGCCCCUCUGCUCCACAGCCUU
UGUGUCCAAGCAGGAGGGCAGCGAGGUAGUGAAGAGACCCAGGCGCUACCUGUAUCAAUGGCUGGGGUGA
GAGAAAAGGCAGAGCUGGGCCAAGGCCCUGCCUCUCCGGGAUGGUCUGUGGGGGAGCUGCAGCAGGGAGU
GGCCUCUCUGGGUUGUGGUGGGGGUACAGGCAGCCUGCCCUGGUGGGCACCCUGGAGCCCCAUGUGUAGG
GAGAGGAGGGAUGGGCAUUUUGCACGGGGGCUGAUGCCACCACGUCGGGUGUCUCAGAGCCCCAGUCCCC
UACCCGGAUCCCCUGGAGCCCAGGAGGGAGGUGUGUGAGCUCAAUCCGGACUGUGACGAGUUGGCUGACC
ACAUCGGCUUUCAGGAGGCCUAUCGGCGCUUCUACGGCCCGGUCUAGGGUGUCGCUCUGCUGGCCUGGCC
GGCAACCCCAGUUCUGCUCCUCUCCAGGCACCCUUCUUUCCUCUUCCCCUUGCCCUUGCCCUGACCUCCC
AGCCCUAUGGAUGUGGGGUCCCCAUCAUCCCAGCUGCUCCCAAAUAAACUCCAGAAG;
DNA d1 = #CACGTGTTTGTATAAATT;
print(@d1);
print('\n');
r1 = d1->;
print('\n');
print('\n');
print(r1);
print('\n');
print(d1%>);
print('\n');
print(r1+>);

RNA r3= #GUGUUUGUAUAA;
print("here");
print(r3+>);
```

Test 56:

```
DNA test = #AA;
test = myarray[0];
print(myarray);
print(test);
```

Test 57:

```
DNA [8] David;
David[0] = "atgc";
print(David[0]);
```

Test 58:

```
DNA hi = "atgc";
print_int(true);
```

Test 59:

```
DNA min = #atgc;
RNA stan = #gcgc;
print(@min);
print(@stan);
```