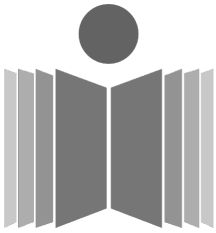

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Compilers 2

Tureasy

Language Whitepaper

By IITH8

"We hope we have made a programming language a little less mysterious for you. If you do want to make one yourself, We highly recommend it. There are a ton of implementation details to figure out but the outline here should be enough to get you going."

INDEX

1. Introduction

Motivation for Designing:

The motivation for designing this language comes from two different areas. For many years, compilers were translators of the high-level code to the assembly code or binary code. This made the efficiency of programs depend entirely on the programmer's coding skills. As AI is advancing in all the fields in recent years, compilers shouldn't be left behind. The compilers must be more than mere translators. A lot of research is going on in different parts of the world for the same. We took it as an opportunity to proudly present our new programming language "Tureasy" which tries to achieve this goal. The compiler could be smart only when there is support from the programming language design. Tureasy supports this by introducing a new concept called tags that would group the data for analysis. The main challenge faced in data analysis and model creation in AI-based problems is the requirement of large amounts of data segregated properly. The tags would help to handle this situation.

The other arena that grabbed our attention was discrete mathematics. It plays a central role in the fields of modern cryptography, social networking, digital signal and image processing, computational physics, analysis of algorithms, etc. as more and more mathematics that is done, both in academia and in industry, is discrete. This motivates us to build a programming language focused on discrete mathematics, to help computer scientists and mathematicians to work more easily and efficiently as compared to that while using a General Purpose Language (GPL).

Goal:

The goal of this programming language subject to a specified time constraint would be to implement a compiler for the programming language Tureasy which is capable of suggesting the programmer with the assistance for the code snippet that is included within the dedicated tags. We have tried to implement these tasks on a domain dedicated to matrix theory, graph theory, and set theory of discrete mathematics.

We would like to extend this to a bigger generalized domain!

2. Lexical Conventions

2.1 Comments:

Both single and multiline comments are supported in Tureasy.
All tokens after a **$** symbol on a line are considered to be part of a comment and are ignored by the compiler.

Ex. 2.1

```
$ This is a single-line comment.
matrix  M1 = [[1.0, 4, 6],  $ This is also a valid single line comment
              [0.5, 3, 0],
              [9.1, 2, 7]];
```

Multiline comments start with **$*** and end with ***$**.

Ex. 2.2

```
matrix  M1 = [[1.0, 4, 6]
              [0.5, 3, 0] $* This is a valid comment *$
              [9.1, 2, 7]];

matrix M2 = [[1.0, 4, 6],
             $* This is a valid
                multiline comment *$
             [0.5, 3, 0],
             [9.1, 2, 7]];
```

Nesting comments are not allowed, and comments cannot be inside strings.

Ex. 2.3

```
$* Comments cannot be $* nested *$ like this *$
string str1 = "Hello $* This is also part of str1 *$ world";
matrix  M1 = [[1.0, 4, 6]
              [0.5, 3, 0]
              [9.1, 2, 7]];
```

2.2 Whitespace:

Whitespace, including tabs, spaces, and comments, will be ignored, except in places where spaces are required for the separation of tokens. Tabs/indentation cannot be used to define the scope in Tureasy.
Both of the programs in Ex 2.4 and 2.5 below are equivalent:

Ex. 2.4

```
void main ()
{
    int a;
    a = input();
    if(a%2 == 0)
    {
         a=a+5;
         output(a);
    }
    else
    {
         output(a);
    }
    return;
}
```

Ex. 2.5

```
void main()
{int a;
a=input();
if(a%2 == 0)
{a=a+5;
```

```
output(a);}
else{output(a);}
return;}
```

2.3 Reserved Keywords:

The following words are reserved keywords in Tureasy and they cannot be used as regular identifiers.

if	else	switch	case	default	link
loop	int	return	break	continue	struct
new	rename	bool	void	const	
int	long	float	string	numset	strset
graph	matrix	AND	OR	const	public
private					

Apart from the ones listed above, all the data types listed in section 3 are also reserved keywords.

2.4 Identifiers:

Identifiers must start with a letter, which can be followed by a sequence of letters, digits, and underscores. Hyphens and other special characters cannot be present in an identifier. Tureasy is case-sensitive, so the identifiers foo and FOO and Foo and fOo are distinct.

Ex. 2.6

```
foo;        $ correct
fo0;        $ correct
f_o;        $ correct
_foo;       $ correct;
```

Ex. 2.7

```
0of;        $ not a proper identifier
f-o0;       $ not a proper identifier
fo^;        $ not a proper identifier
```

2.5 Punctuators:

Statements should be terminated with semicolons (;).

2.6 Tags:

Tags in Tureasy are used to identify the part of code that requires specific modifications. The nodes between the opening and closing tags are colored in the abstract syntax tree formed after the semantical analysis. They begin with **#** and end with **#!**. They should not be used between instructions, string literals, integers, etc.

3. Data Types

3.1 Primitive Data Types:

data type Name	**Description**	**Initialization**
int	64-bit signed integer value	int x = 7;
float	64-bit signed floating-point number	`int x = 7;`
string	sequence of characters	string str1 = "Testing 1..2..3... Hello world! ";
bool	stores either 0/1 values for true/false	bool a = 1; bool b = 0;
struct	user-defined, similar to a struct in C	struct ex1 { int x; float y; string str1; matrix int Q; set B; };

		struct ex1 myStruct.str1 = "Example";

3.2 Non-Primitive Data Types:

Using [ ] for declaring elements of graph, set, matrix, etc.

data type Name	Description	Initialization
numset	stores a set of numerical values	`numset A = [1, 2, 3, 8.5, -1];`
strset	stores a set of strings	`strset A = [“1”, 2”,“3”, "str","popl"];`

graph	a set of nodes and edges	`graph G = [1 : 2,3,4;`
		`2 : 1,5;`
		`3 : 1,4;`
		`4 : 1,3;`
		`5 : 2,6;`
		`6 : 5;`
		`];`

matrix	a matrix	`matrix M = [[1.0,4,6],` `[0.5,3,0]];`

4. Operators and Expressions

4.1 Precedence - Highest to Lowest:

Purpose	Symbol	Associativity	Valid Operands
Parentheses for grouping of operations	( )	left to right	int, float
Member access operator	.	Left to right	struct
Unary negation	!	right to left	all bool
Increment	++	right to left	int, float, matrix

Decrement	--	right to left	int, float, matrix
Shift operators	<<	left to right	Int, long
	>>	left to right	Int, long
Exponent	^	left to right	int, float, matrix
Modulo	%	left to right	int, float
Multiplication	*	left to right	int, float, matrix
Division	/	left to right	int, float
Addition	+	left to right	int, float, string, matrix
Subtraction	-	left to right	int, float
Union	|	left to right	numset, strset
Intersection	&	left to right	numset, strset
Set difference	~	left to right	numset, strset
Relational Operators			
	<= < >= >	left to right	all datatypes
	== !=	left to right	all datatypes
Logical Operators	AND OR	left to right	bool
Assignment operators	= *= += /= ^= %=	right to left	wherever the operator is valid

Ex. 4.1

```
int a = 5+13;                  $ evaluates to 18
float b = 15.23+5;             $ evaluates to 20.23
float c = 4-10.2;              $ evaluates to -6.2
int d = 5*6;                   $ evaluates to 30
float e = 20.0 * 0.25;         $ evaluates to 5.0
int f = 10/2;                  $ evaluates to 5
float g = 10.8/2;              $ evaluates to 5.4
int h = 13%3;                  $ evaluates to 1
float i = 10.8%2;              $ evaluates to 0.8
```

```
        int j = 2^3;                    $ evaluates to 8
        float k = 3.5^2;                $ evaluates to 12.25
```

Ex. 4.2

```
 string str1 = "Hello";
 string str2 = "World";
 string str3 = str1 + str2;   $ "Hello"+"World" results "HelloWorld"
 string str4 = str2 + str1;   $ gives "WorldHello"
```

Ex. 4.3

```
        matrix M1 = [[2, 3, 4],
                     [1, 5, 0],
                     [1, 4, 8]];

        matrix M2 = [[1.0, 4, 6],
                     [0.5, 3, 0],
                     [9.1, 2, 7]];

        matrix Mat1 = [[5, 2, 3],
                       [8, 11, 0]];

        matrix M3 = M1 + M2;      $ matrix addition, term by term
        matrix M4 = M1 - M2;      $ matrix subtraction, term by term
        matrix M5 = M1 - M2;      $ matrix subtraction, decimal truncated
        matrix M6 = M1 * M2;      $ matrix multiplication
        matrix M7 = M1 * M2;      $ matrix multiplication, truncates
fractional part
        matrix M1 ^= 2;             $ assigns M1^2 to M1
```

The assignment operator can be used on variables with the same data type. Int and Float data types are compatible with each other with respect to assignment, that is an int can be assigned to a float without getting a compilation error, and a float can be assigned to an int which leads to the decimal part being truncated in the int variable.
Apart from int and float, variables of different data types cannot be assigned to each other.

4.2 Tag Expressions:

They mark the beginning and end of tags. They use the operator **#** for beginning and end with **#!**. The data between these expressions undergo analysis during compilation.

5. Declarations

A program consists of various entities such as variables, functions, types. Each of these entities must be declared before they can be used.

Ex. 5.1

```
int fun1(int num1)
{
    return num1 + 42;
}

void main()
{
    int num1;
    num1 = fun1(2);
    string str1 = "Number is ";
    output(str1, num1);
}
```

OUTPUT:

Number is 44

5.1 Variable declaration:

All variables used in a program must be declared before using them in a statement, followed by a semicolon.

- int w = 5, x;
- bool y;
- float z;
- set A;
- string s;
- matrix M1;
- graph g;

5.2 Declaration scope:

In Tureasy, the scope is defined by using { }. The identifier introduced by a declaration is valid only within the scope where the declaration occurs. Variables declared in global scope must have unique identifiers. The same identifier cannot be used to refer to more than one entity in a given local scope. For example, in Ex. 5.1 above, the variable num1 refers to two different variables, one in main and the other in the scope of fun1(). However, the programs below in Ex 5.2 and 5.3 are not valid.

Ex. 5.2

```
int fun1(int x)
{
}

int num1;

int main()
{
    int num1; $ invalid identifier as num1 has already been used for a
global variable
    num1 = fun1(2);
    string str1 = "Number is";
    output(str1, num1);
}

int fun1(int num1)
{   $ invalid identifier as num1 has already been used for a global
variable
    return num1 + 42;
}
```

Ex. 5.3

```
int fun1(int num1) $ invalid identifier as it has already been used for a
global variable
{
    return num1 + 42;
}

int num1;

void main()
{
    int num2;
    float num2; $ invalid identifier as num2 has already been used for a
variable within the same/higher scope
    num1 = fun1(2);
    string str1 = "Number is";
    output(str1, num1);}
```

5.3 Rename:

Rename keyword is used to declare a new name that is an alias for another name, the new name is taken to be everything after the comma on the same line.

Ex. 5.4

```
struct node
{ int x;
  float y;
  string str1;
  matrix Q;
  set B;
};
rename struct node, node;  $ from this point onwards, we can use 'node'
instead of 'struct node'
node ex1;

ex1.str1 = "Example";
```

5.4 const:

The const keyword specifies that a variable's value is constant and tells the compiler to prevent the programmer from modifying it. For instance, the program below in Ex. 3.13 will not compile as the variable i should not be modified.

Ex. 5.5

```
int main()
{
   const int i = 5;
   i++;
}
```

5.5 Array declaration:

We use a one-dimensional matrix instead of a traditional array. If the elements are not initialized, then it is inferred to be a one-dimensional array. The size of the matrix can be specified by built-in functions. Default values of the matrix are of float type.

The syntax for matrix is

```
matrix variableName = [[...], [...], ...];  $ Here, ... indicates comma
separated values
```

5.6 Function declaration:

The syntax for function declaration is

```
return_type functionName(datatype_1 arg_1, datatype_2 arg_2, ...);
$ datatype_i is the datatype of arg_i
```

A function may or may not have arguments but the return type is a must (return type void can be used if none is required).

5.7 Initializers:

When an object is declared, its init-declarator may specify an initial value for the identifier being declared. The initializer is preceded by = and is either an expression or a list of initializers nested in braces.

5.7.1 Types of Initializations:

5.7.1.1 Default initialization:

When variables are declared but not initialized, they are initialized by default to zero in the case of primitive data types, an empty string for strings, and null for non-primitive data types. Non-primitive data types should be used without initialization.

Ex. 5.6

```
int a; $ Here, a = 0
set A; $ Here, A is null
matrix mtx;  $ Here, mtx is null
```

5.7.1.2 Direct initialization:

Ex. 5.7

```
int i = 3;
string s = "hello";
matrix mtx = [[2,3], [4,5]];
set A= [1,2];                    $ set
```

> For matrices, the initializer is a square-bracket enclosed list of initializers for its members. If the array has an unknown size, the number of initializers determines the size of the array, and its type becomes complete. If the array has a fixed size, the number of initializers may not exceed the number of members of the array; if there are

fewer, the trailing members are initialized to 0 or null, the default value of the type of the matrix.

5.7.1.3 Copy initialization:

It is the initialization of one variable using another variable. This kind of initialization can be used for all supported data types.

Ex. 5.8

```
int a = 3, b;
b = a;
```

6. Statements

6.1 Expression Statements:

Statements are executed for their effect and do not have values. Each expression statement includes one expression, which is usually an assignment or a function call. In Tureasy, every expression is followed by a semicolon ";" .

Syntax:

```
expression;
```

In Tureasy, they fall into several categories like:

- Compound statements
- Labeled-statements
- Selection statements
- Conditional statements
- Iteration statements
- Jump statements

6.2 Compound Statements:

A compound statement is a sequence of statements enclosed by braces as follow:

Syntax:

```
{
statement1;
statement2;
}
```

If a variable is declared in a compound statement, then the scope of this variable is limited to this statement.

6.3 Control Flow:

6.3.1 Conditionals:

If statements consist of a condition (an expression) and a series of statements. The series of statements is evaluated if the condition evaluates to True. If the condition evaluates to False, either the program continues or an optional else clause is executed.
Below is the pseudocode :

if (condition(statement)) {
Statements;
}

else if (condition(statement)){
Statements;
}

else {
Statements;
}

Statement following if/ else if (condition) must be a boolean statement, or can be evaluated to boolean.
Ex. 6.1

```
numset A = {1,2,3,4,5};
if(size(A) == 5){
      output("the size is ", size(A));
      output(A);
}
```

6.3.2 Loop Statements:

Loop statement consists of a condition and incrementer separated by**;** followed by a series of statements. The statements are repeatedly evaluated as long as the condition remains True before each repetition. The “;” is used as a separator inside the condition statement and especially there is no restriction on using it as you can infer from the two examples below. Note that assigning over the looping variable will not change the original variable in the scope.
Ex. 6.2

```
int n=0;
loop(isprime(n) AND n < 10000)
{
      n++;
      output("n is", n);
}
```

Ex. 6.3

```
int n = 0;
loop (n!=20;n++)
{
}
```

6.3.2.1 break statement:

- The break statement ends the loop immediately when it is encountered. Its syntax is: break;
- The break statement is almost always used with an if...else statement inside the loop.

Ex. 6.4:

```
 loop(i-- >= 0)
{
    x=func(i);
    if(x==2)
      x=x+2;
    else
      break;
}
```

6.3.2.2 continue statement:

- The *continue* statement skips the current iteration of the loop and continues with the next iteration. Its syntax is: *continue;*
- The *continue* statement is almost always used with the if...else statement.

Ex. 6.5:

```
 loop(i-- >= 0)
{
    x=func(i);
    if(x==2)
        continue;
    x=x+2;
```

```
}
```

7. Tags

The tags are special statements that group parts of code that have some standard implementation involved. It is used in this format **#<tag_name> code #!<tag_name>**

Some of the commonly used public tags in Tureasy are

loop	Unique var	var < (const)
graph-theory	number-theory	var in range (range)
unused	dp	innerloop
checkhere	exponentiation	divide and conquer

The tags provide programmers with tips related to

1. Parallelism: There are some constructs of Tureasy which support parallel execution. The programs including such constructs would improve performance but are hard to code. The tags analyze the data and provide suitable constructs that could replace the existing code.
2. Constraints: The correctness of algorithms can be determined by finding base rules which must be satisfied throughout it. In large programs, it becomes practically impossible to keep track of these rules. So, the programmer could make use of constraint tags and the tag ensures that the property is maintained. In case of failure, it would suggest modifications for the same.
3. Memory optimization: The tags provide us with tips that could optimize memory too. There could be instances where the programmer might allocate heap memory but never use it or might use a lot of stack memory unnecessarily.
4. Time complexity: Tureasy tags try to improve the code by understanding the code and providing us with better constructs that could help us reduce time complexity.

The tags use turzers during compilation to provide these tips. The tags are used to color the nodes in the abstract syntax tree during the semantic analysis phase. During the compilation, an abstract syntax tree is formed after the semantic analysis phase which undergoes machine-independent code improvement. The turzers are used during this phase.

Turzers are the files that contain the machine learning models for analyzing the data between tags. The abstract syntax tree is used as input for the turzers. The models within turzers are

made using trees and its traversal has a certain cost associated with it. The program is compared with these models and tips are given accordingly.

There are private turzers and private tags associated with companies. These turzers have an additional requirement of .tcnf files which are the turzer configuration files. These tags can be uniquely defined and modified by the company based on its requirements.

8. Built-ins and Standard Library Functions

We plan to implement some standard libraries/headers to provide support for the following domains:

- Matrix operations
- Graph operations
- Common math functions
- Set theory
- String operations

9. Grammar

Type: INT | FLOAT | STRING | BOOL | STRUCT | NUMSET | STRSET | GRAPH | MATRIX | VOID | LONG

Operator: PLUS | MINUS | MULTIPLY | DIVIDE | MODULO | EXPONENT | LPAREN | RPAREN | SHIFT_LEFT | SHIFT_RIGHT | TRANSPOSE | DECREMENT | INCREMENT | UNARY_NEGATION | MEMBER_ACCESS | UNION | INTERSECTION | SET_DIFFERENCE | Relational | Logical | Assignment

Relational: ‘<=’
| ‘<’
| ‘>=’
| ‘ >’
| ‘==’
| ‘!=’

Logical: ‘&’
| ‘|’

Assignment: ‘=’
| ‘+=’
| ‘-=’

```
                | '*='
                | ' /='
                |  '^='
                | '%='

Decls: /*nothing*/
       | Decls var_decl
       | Decls fun_decl

fun_decl: Type IDENTIFIER LPAREN Args RPAREN

Args: /*nothing*/
          | Type IDENTIFIER
          | Type IDENTIFIER ',' Args

var_decl: Sc_specifier Type id_list ';'

id_list: IDENTIFIER
        | id_list ',' IDENTIFIER

Sc_specifier: CONST | STATIC | RENAME | /*nothing*/

stmt_list: /*nothing */
           | stmt_list stmt

stmt:    expr ';'
          | BREAK ';'
          | CONTINUE ';'
          | RETURN expr ';'
          | RETURN ';'
          | LBRACE stmt_list RBRACE
          | IF LPAREN expr RPAREN stmt ELSE stmt
          | IF LPAREN expr RPAREN stmt
          | LOOP LPAREN expr ';' expr RPAREN stmt
          | LOOP LPAREN expr RPAREN stmt

Expression : expr Operator expr
          | NOT expr
          | ABS expr ABS
          | TRUE
          | FALSE
          | FLOAT_LITERAL
          | STRING_LITERAL
          | LITERAL
```