LiTeC

White Paper

CS20BTECH11008	BATHINI ASHWITHA
CS20BTECH11025	KETHAVATH PRANEETH NAYAK
CS20BTECH11029	LANKA PRASANNA
CS20BTECH11032	MEGH SHAH
CS20BTECH11044	SANDEEP L
CS20BTECH11047	SIDDA BOENA JEEVAN SAMMESWAR
ES20BTECH11025	SAI SIVA ROHITH TIRUMALASETTI

INDEX

LiTeC Introduction

LiTeC (.ltc {lowercase L, T, C}) is a high-level imperative, general-purpose programming language. Our language is not focused on one primary goal; instead, we are trying to solve different problems. The name LiTeC is derived from: 'Li' referring to LISP and its list processing; 'Te' referring to TeX and its type-setting system; and last but certainly not least, 'C' referring to the C language, the primary influence on LiTeC.

Language Design

LiTeC includes the following essential parts: data types, variables, expressions, statements, blocks, functions, decision control structures, and standard libraries. Besides these, the uniqueness of the language arises from the following key features:

1. **Subfiles**
   We will introduce tree-like subfiles where multiple programmers can work concurrently in the same program. The sub-files are arranged in a tree data structure and its hierarchy management incorporates the principles of inheritance and encapsulation.

2. **Programming Paradigm**
   LiTeC is closer to the intersection of procedural and object-oriented paradigms.

3. **TeX output**
   In-built functions that take the input text with directions to produce .tex files that are to be used by the LaTeX compiler for generating .pdf files, essentially embedding TeX into LiTeC.

4. **Enumerated comments**
   Expanded further in the Comments section

5. **Safer Address usage by index-bound checks**
   Every indice call on an array, matrix or struct is checked to be within the bounds of its declaration.

6. **In-depth error messages for common errors**
   Expanded further in the Error Handling section

Uses

The myriad features of LiTeC ensure its robustness as it's useful in versatile situations. Using prefix notations is easy to implement. Sub-files and enumerated comments help in readability and accessibility. TeX compatibility eases the generation of TeX files.

LiTeC language specification

Keywords

Keywords are predefined words that are reserved and understood by the compiler.

Keyword	Description
declare	A declaration specifies a unique name for the entity, along with information about its type and other characteristics.
receive	Used to receive code from other subfiles and libraries 'as is' into the current Main (root subfile) or any other subfile.
if	The statements inside the 'if' block are executed only if the condition is true.
else if	Statements in any 'else if' branch are executed only if the corresponding condition evaluates to true, otherwise, the statements are skipped.
else	The 'else' keyword is used to specify a block of code to be executed, if the 'if' and 'else if' conditions are false.
break	The 'break' keyword is used to terminate the loop.
continue	The 'continue' keyword skips an iteration and continues with the next iteration in a loop.
invariant	The keyword 'invariant' is used when we want the value of the variable to be the same throughout the program.

loop	The keyword ‘loop’ is used to repeat the execution of a block of statements, while the given condition holds.
return	Termination of a function and the required, resulting value is returned to the variable that called the function.
void	Void doesn’t return any value when the function return type is void.
bool_1	Mentioned in the Primitive Data Types section.
char_8	
int_64	
double_128	

Comments

Comments are used for readability and explaining code. They are ignored by the compiler.

Comment	Description
/* ... */	Multi-line comments
#n	Enumerated Comment Placeholder (ECP) used to recognize a comment by a unique number n, where n is an integer greater than 0.
#n /* ... */	Enumerated Comment Expansion (ECE) for detailed description of this comment. Every ECP must have only one ECE tagged to it.

Primitive Data Types

Data types that are inbuilt in the programming language are called primitive data types.

Data type	Description	Format Specifiers	Default size (bits)
bool_1	It can only two values ‘0’ or ‘1’	%b	1
char_8	It holds a single character, a letter or number	%c	8
int_64	Data type to store signed integer values	%i	64
double_128	It stores numbers with floating point values	%d	128

While declaring variables and functions, we should mention the number of bits the specific data type holds. For ‘int_64’ and ‘bool_1’ the declarations are as follows:

```
declare int_64 a : 5;
declare bool_1 isCorrect : 1;
```

Operators

LiTeC supports the following operators:

Operators	Symbol	Description
Arithmetic Operators	+	Addition
	-	Subtraction
	*	Multiplication
	/	Division
	%	Modulo
Relational Operators	=	Checks Equality
	<	Checks Lesser Than
	>	Checks Greater Than

Logical Operators	^	And Operator
	|	Or Operator
	~	Not Operator
Assignment Operator	:	Is Assigned
Address of	&	Returns address

Character set

Set of all valid characters supported by LiTeC.

Type of character	Description
Alphabets	Lower case a-z
	Upper case A-Z
Digits	0-9
Special characters	` ~ @ ! $ # ^ * % & ( ) [ ] { } < > + = _ – | / \ ; : ‘ “ , . ?

Punctuators

Punctuators are tokes having semantic meaning.

Punctuator	Symbol	Description
Semicolon	;	Indicates End of Statement
Brackets	( )	The only parentheses used for operations and grouping expressions
	[ ]	Used to access indices of arrays and structs
	{ }	Used to create a block (of compound statements)

	:[ ]:	This combination is used for including LiTeC code within the TeX block
	{[ ]}	This combination is used for separating subfiles
	([ ])	This combination is used inside subfiles for public inheritance
	#TeX { }	Block for including tex
Ellipsis	...	Variable length argument list
Single quotes	' '	Character constant
Double quotes	“ ”	String literal
Comma	,	Argument list separator

Expressions

LiTeC expressions use prefix notation inspired from LISP. All operators are prefixed to operands they work on.

Example:
d : (* (+ a b) c);

The above example evaluates to **d = (a+b)*c** in infix notation.

Whitespace

Character	Description
‘ ’	Space
\n	New line character

\t	Tab space

Identifiers

LiTeC identifiers are names given to variables, functions, structures, and subfiles. We introduced a new keyword, “declare” in declarations ensuring uniqueness. A valid identifier can be any combination of letters and numbers beginning with a letter. In some cases, we have a combination of two identifiers separated by “_” for more informative declarations.

Declarations

Variable Declarations

Every declaration must be followed by initialization.
syntax:
declare data_type identifier : initialised_value;
example:
declare int_16 a : 11;

Function Declarations

Syntax:

```
declare libraryname_functionname(arguments) -> return_type
{
        statements;
}
```

```
declare standardlibrary1_geq(double_128 x, double_128 y) -> bool
{
    declare bool_1 b : !(< x y);
    return b;
}

declare FunctionName(args,args…)->return_type
{
    statements
}

```

Note: Function declarations and definitions cannot be separate in LiTeC.

Data Structures

Array

Declaring a character array A of size n.

```
declare array char_8 A[11];
loop (declare int_64 i : 0; (< i 11); i : (+ i 1) )
{
        A[i] : i;
}

```

Matrix

Declaring a matrix K of size n x m.

```
declare matrix double_128 K[11][24];
loop (declare int_64 i : 0; (< i 11); i : (+ i 1) )
{
loop (declare int_64 j : 0; (< j 24); i : (+ j 1) )
{
    A[i][j] : ( * i j );
}
}

```

Struct

Example struct (S) declaration of 1 int and 2 doubles:

```
declare struct S[int_64][double_128][double_128];
S[1] : 5;
S[2] : 2.5;
S[3] : 1.25;

```

Standard Library

Input and output

print("string", %format); (implemented using fwrite)
scan("string", &%format);

Additional Libraries

- ➢ Set Theory
- ➢ Probability and Statistics
- ➢ Trigonometry
- ➢ Vectors and Matrices

Program Control Statements

Decision Control

The following are representative code examples of the 'if', 'if-else' and 'if-else if-else' decision control statements.

1. 'if' condition

```
if(condition)
{
    statements
}
```

2. 'if-else' condition

```
if(condition)
{
        statements
}
else
{
        statements
}
```

3. ‘if-else if-else’ condition

```
if(condition)
{
        statements
}
else if(condition)
{
        statements
}
else
{
        statements
}
```

Loop control

LiTec offers a single loop structure, which is equivalently as powerful as a ‘while’, ‘do-while’ or a ‘for’ loop as standardly present in most languages.

```
loop( initialization; condition; incrementation)
{
      statements
}
```

Embedding TeX

TeX code blocks are integrated into LiTeC in the block #TeX { } and variables, operations, statements and functions are embedded into the TeX code using the parentheses syntax :[ ]:

Note: Errors in .TeX files compilations aren't handled by the LiTeC compiler.

```
#TeX
{
    \use{packagename}
    \begin{document}

\section{sectionName}
        TexFunction:
[: ( functionStatements + operations)*  :]

\Begin{enumerate}
\Item  sum is 5+4 = [< (+ 5 4)  >]
\Item Factorial of 10 is  [< factorial(10) >]
…
\Item
\end{enumerate}

}
```

Subfiles

In order to internalize the concept of inheritance and encapsulation, in terms of certain regions of the program having different read and write accesses for other sections, subfiles have been introduced.
The entire program is a tree of subfiles, with the mandatory Main file at the top of the program, and all other subfiles being its direct or indirect children nodes.
Every subfile has it’s name declared at the beginning, followed by the names of all the subfiles that are received by it, through public inheritance.
Every subfile must have a main_function() declared for that subfile, which will be executed whenever that subfile is run. main_function() are always in the private section of a subfile, and hence, cannot be inherited.

```
{[
    Main
    ([
        /*code to be used by Main and all sub-files (by public inheritance)*/
])

/*private to the Main sub-file*/

declare Main_mainfunction()->int
{
    return 1;
}
]}

{[
    SubFile1                /*sub-file name*/
receive Parent_Subfile1, Parent_Subfile2, …
([
/*code to be used by SubFile1 and it's descendents (by public inheritance)*/
    ])

/*private to the SubFile1*/

declare SubFile1_mainfunction()->int
{
    declare int_64 a:0;
    return a;
}
]}

```

Error handling

The following compile time errors would be dealt with appropriate error message and return value

Error type	Return value	Error message
Lexical	1	“Incorrect keywords / operators / identifiers, program exited with Lexical error”
Syntactical	2	“Missing punctuators, program exited with Syntactical error”

Semantical	3	“Type mismatch during assignment, program exited with semantical error”
Length	4	“Loop iteration limit crossed, program exited with length error”

Sample section of a LiTeC program

```
declare factorial(int_64 x)->int_64
{
    int_64 p : 1;
    if (x=1 | x=0)
    {
        return 1;
    }
    for (int_64 i : 1; ( < i ( + x 1 ) ) ; ( + i 1) )
{
    p = (* p i);
}
return p;
}

#TeX
{
    \use{packagename}
    \begin{document}

\section{sectionName}

\Begin{enumerate}
\Item  sum is 5+4 = [< (+ 5 4)  >]
\Item Factorial of 10 is  [< factorial(10) >]
…
\Item
\end{enumerate}
\end{document}

}
```

Post compiling:

Generated *Output.tex* file contains following data

```
\use{packagename}
    \begin{document}

\section{sectionName}

\Begin{enumerate}
\Item  sum is 5+4 = 9                /* [: (+ 5 4 :] is replaced by 9  */
\Item Factorial of 10 is 3628800  /* [: factorial(10) is replaced by 3628800  */
…
\Item
\end{enumerate}
\end{document}
```