TEAM-8

PROJECT FINAL REPORT

CS20BTECH11029	LANKA PRASANNA	PROJECT MANAGER
CS20BTECH11049	SIDDABOENA JEEVAN SAMMESWAR	SYSTEM ARCHITECT
CS20BTECH11032	MEGH SHAH	LANGUAGE GURU
CS20BTECH11044	SANDEEP L	SYSTEM INTEGRATOR
CS20BTECH11025	KETHAVATH PRANEETH NAYAK	SYSTEM INTEGRATOR
ES20BTECH11025	SAI SIVA ROHITH TIRUMALASETTI	SYSTEM ARCHITECT
CS20BTECH11008	BATHINI ASHWITHA	TESTER

Contents

1. Introduction

LiTeC 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. In the acronym LiTeC, Li stands for LISP, Te stands for TEX, and C stands for C programming language. This language is the intersection of procedural and object-oriented programming paradigms.

Variables, data types, expressions, data structures (arrays and matrices), statements, functions, blocks, loops, and standard libraries(proposed) are there in LiTeC.

There is a function called TeX, which takes input in the Tex programming language but can also include functions or expressions of LiTeC. This enables us to solve complex problems in LiTeC and call those functions directly in TeX. This feature can be furthur extended by embedding LaTeX compiler with LiTeC so that output of TeX function results in pdf.

We use prefix expressions inspired by LISP. Since there is no concept of precedence and associativity, these expressions make our language highly efficient.

Our program has subfiles that enable multiple programmers to simultaneously work on the same file. Each subfile has block of code which can be inherited by subfiles following it. Object oriented concepts encapsulation and inheritance can be seen here.

2. Language tutorial

Hello World using LiTeC

print(“Hello World”);

Program Structure

The program starts executing from the first line of code. The further execution of the code is sequential except when it encounters jump statements, loops, functions etc.

Compile and Execute LiTeC Program

To compile and execute a LiTeC program, follow these steps:

```
yacc -d parser.y
lex lex.l
gcc symbol_table.c ast.c semantic.c lex.yy.c y.tab.c -o parser
```

Data types in LiTeC

bool, int, double and char are the available data types in LiTeC.

Operators in LiTeC

All the basic arithmetic operators and relational operators are supported by LiTeC.

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

Semicolons and Blocks in LiTeC

In LiTeC, every statement must end with a semicolon that indicates the end of a logical entity.

For example,

```
a : (+ b c);
k : fact(a);
```

A block of statements represents a set of statements that are logically connected. Such a block of statements must be enclosed within curly braces.

For example,

```
{
        a : 2;
        b : 3;
        c : (+ a  b);
        print(“%i”, c);
}
```

LiTeC Identifiers

An identifier in LiTeC is used to identify a variable, function, structures and subfiles. An identifier can be any combination of letters, digits and underscores but it must start with a letter only. The keyword ‘declare’ followers by the data type is used to declare an identifier.

For example,

```
declare int a : 2;
```

Comments in LiTeC

Comments are written to understand what a particular statement or block of code does and the flow of the code. They are ignored by the compiler.

For example,

```
a : (+ b c); /* Addition of two numbers */
```

Enumerated comments help us to uniquely identify a comment and navigate to it easily.

For example,

```
#1 /* Multiplication of two numbers */
p : (* q r);
```

Decision Making in LiTeC

The decision making statements in LiTeC are:

1. if

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

2. if - else

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

3. if - else if - else

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

```
            statements;
        }
```

Loops in LiTeC

Since the for, while and do while loops can be written in terms of the other two, there is only one loop in LiTeC. The syntax is:

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

Functions in LiTeC

A function in LiTeC acts as a map between a set of zero or more inputs and one output. The syntax for function declaration is:

```
return_type functionName(argument1, argument2, ...)
{
        statements;
}
```

Data Structures in LiTeC

1. Arrays: An array in LiTeC can store integers, floating point numbers and characters. The datatype of all elements of an array must be the same. To declare an array we need to follow these steps below:

   ```
   declare int A[10];
   declare char B[20];
   ```

   To initialize the elements in an array:

```
        loop(declare int i : 0; (< i 10); i : (+ i 1))
        {
                A[i] : i;
        }
```

How to include TeX code?

To include TeX code in LiTeC, we have to write them within the TeX {} block. To include a variable, a statement, function or perform any operation within this block, we must encode them within [: :]

For example,

```
TeX
{
        "\use{packagename}

        \begin{document}

        \section{sectionname}",

        [: texFunction(parameters list) :],

        "\end{document}"
}
```

Working with Subfiles in LiTeC

The following example illustrates the way to work with subfiles in LiTeC:

```
{[
        MainFile

        ([
        declare int x : 10;
        declare char a : "b";

        ])

        int main()

        {

        declare int u : (+ x 10);
        declare int v : 15;
        return 0;

        }

]}

{[

        SubFile1

        ([

        declare int y : 20;
        declare char c : "x";

        ])


        int main()

        {
```

```
        declare int m : (/ x y);
        declare char b : a;
        return 0;

        }

    ]}

    {[

        SubFile2

        ([

        declare int z : 30;

        ])

        int main()

        {

        declare int m : (% x (* y z));
        declare char b : c;
        c : a;
        return 0;

        }

    ]}
```

Example code:

```
{[
    MainFile
    ([
        int fact(int n)
        {
            if(n < 2)
            {
                return 1;
            }
            n : (- n 1);
            return (* (fact(n)) (+ n 1));
        }
    ])


    declare int A[3][3];
    declare int B[3][3];
    declare int C[3][3];
    declare int i : 0 ;
declare int j : 0 ;

int main()

{

    loop( i:0 ; i < 3 ; i: (+ i 1) )

    {

        loop(j:0; j < 3 ; j: (+ j 1) )

        {

            A[i][j] : (* i j );

            B[i][j] : (+ i j );

        }

    }

    TeX{

       "/document{article}",

       "/autor{Name}",

       "/title{TITLE}",

       "/begin{document}",

       "/section{MatrixAddition}",

        [: ( + A B ) :],

        "/end{document}"
        }

        return 0;

    }

]}
```

Expected output :

A tex output file was generated by compiler output.tex which is passed to tex compiler for further compilation.

The snap of output.tex is :

```
/document{article}

/autor{Name}

/title{TITLE}

/begin{document}

/section{MatrixAddition}

  [[0,0,0],       [[0,1,2],

   [0,1,2],   +    [1,2,3],      =   [[0,1,2],[1,3,5],[2,5,8]]

   [0,2,4]]        [2,3,4]  ]


 /end{document}
```

3. Project plan

Week 0:

- Discussed various ideas for language and finalized one.
- Came up with basic features of our language and decided tool for lexical analysis and parsing.

Week1:

- Discussed primitive features and syntax of our language.
- Worked on unique features to implement, and we finalized our language.
- Submitted language specification document via GitHub for assignment.

Week 2:

- Explored and learnt about ANTLR
- Started writing grammar for LiTeC

Week 3, 4:

- We started working on lexical analysis and parsing.
- We completed and submitted lexical analysis deliverables.

Week 5:

- Started writing YACC grammar
- Made some minor changes and added missed parts in lexer.

Week 6:

- Finished writing YACC grammar.
- Completed parser and resolved all shift-reduce conflicts.
- Submitted code, ppt and videos.

Week 7:

- Added changelog, line-count, and reorganized github repo.
- Learnt about semantic analysis and it’s implementation.

Week 8:

- Decided data structure and started implementing symbol table.
- Learnt about attribute grammar and ast generation.

Week 9:

- Learnt about the implementation of semantic actions in Yacc
- Working on ast generation

Week 10:

- Done with symbol table implementation
- Working on semantic actions

Week 11,12,13:

- Included symbol table in parser and tested.
- Started working on ast implementation.

Week 14,15:

- Type checking was implemented.
- Ast was partially completed.

Week 16,17:

- Ast and symbol table are completely done.

Week 18:

- Essential error checks in semantic analysis were added.
- Started learning about code generation.

4. Language evolution

Our language has gone through many changes since we started. Our test cases used in Lexer and Semantic analyzer differ which clearly shows that this language evolved during the different phases.

We removed keywords like invariant. Our data types no longer consist of a number of bytes. i.e int_64 is changed to int. Function definitions changed to C style rather than ones similar to python.

Our concepts of subfiles and TeX are more clear. LiTeC code is readable and understandable compared to the ones which we initially planned.

5. Compiler architecture

Lexer

First phase is lexical analysis. In this phase, we converted the sequence of characters into tokens. We used the lex tool to generate a lexical analyzer. We removed comments and whitespaces. A line number is added. Our lexer takes LiTeC source code and returns all valid tokens in standard output in this phase.

Parser

Second phase is syntax analysis. In this phase, we checked syntactical structure by building a parse tree. In this phase, tokens are taken as input to output the parse tree. We used the YACC tool as a parser generator. We wrote grammar containing a set of rules to recognize strings of our language. Our parser outputs syntax errors if there are unrecognized or missed tokens and prints parsing completed when everything is syntactically correct.

Semantic

Third phase is semantic analysis. Here we checked semantic correctness using the syntax tree and symbol table. Semantic checks included in our code are errors on re-declaration of variables. Type checking for expressions and function return types. Type checking of function arguments, and errors on undeclared variables.

6. Development environment

Github:

We used Git, a version control system, and Github to push our work whenever a phase is completed.

VSCode

We used the VSCode editor for writing code. VSCode interfaces are user-friendly. It comes up with IntelliSense, useful command line options. We also used VSCode for live share.

7. Conclusions: Lessons we learned

It was a great experience to work on it in a group project. We understood all phases of compiler design clearly. We learned the concepts behind language design and compilers with hands-on experience. Now we can design and implement our own simple domain-specific language.

We learned how to handle pressure and the importance of time, especially during deadlines. Weekly progress reports helped make us more responsible and productive. We learned teamwork divides tasks but multiplies the success. We learned how to figure out things on our own. We also learned that starting is a significant step and it's ok to move slowly until we don't stop.

8. Sources

https://www.lysator.liu.se/c/ANSI-C-grammar-l.html

https://www.lysator.liu.se/c/ANSI-C-grammar-y.html