LUCAS Language Final Report
Team 3

JATIN TARACHANDANI
CS20BTECH11021

JULAKUNTLA MADHURI
CS20BTECH11023

PRASHANTH SRIRAM S
CS20BTECH11039

S GOUTHAM SAI
CS20BTECH11042

SHANTANU PANDEY
CS20BTECH11046

ARAVINDA KUMAR REDDY T
CS20BTECH11053

ANIRUDH SRINIVASAN
CS20BTECH11059

SRIVATSAN T
CS20BTECH11062

22nd November 2022

Contents

1 Introduction

1.1 About the Language

1. "Lucas - Programming made simple" is a compiled, statically typed, object based language with calculus features and support for arbitrary precision integers as its USP.

2. It draws inspiration from some of the popular languages like C++, Java, Go, COOL, LaTeX and many more.

1.2 Why Lucas?

1. The need for our language is because of the unique combination of features that we support. We have designed this language for mathematicians, academicians, scientists, business analysts and for anyone who might need calculus support. The idea behind our language is to keep all relevant features bundled so that it's easy for the programmer to get the computational details that he needs, all at one place instead of depending on multiple languages in a technological stack.

2. We are planning to start with a support for calculus functionalities that can evaluate accurately the differentials at a point and integrals over an interval with certain level of user-defined precision level. We'll add upon relevant packages for differential equations and anymore related functionalities in our next release based on the feedback from the user community.

1.3 Design Goals

1. We want to make life easier for the programmer, so our focus is on bringing a balance between productivity and performance. Based on the use cases we could imagine of, we decided to go with an object based language that supports data encapsulation features.

2. The idea of having begin and end with labels for any block of code comes from LaTeX, which majority of the users, (mostly mathematicians) are already familiar with. And also, dangling if-else problem can be taken care of, with the help of proper labelling.

3. Also, we support multiple return types that can returned from a function. The programmer doesn't need to encapsulate the return values into a single structure before returning it from the function.

4. Arbitrary precision integers are represented as the new non-primitive datatype **bigint** that the language supports.

5. We support 32-bit int data type as well, which can be used if the programmer has the domain knowledge that the variables are going to be within the range supported for normal integers.

6. And necessary exceptions will be thrown, whenever the compiler feels that there might be a chance of overflows or if int might not be suitable for the operation.

2 Language Tutorial

Lucas syntax is very easy to understand and intuitive as well. The syntax is very similar to the C++ programming language but we have refined it at a lot of small points to help the programmer avoid trivial yet annoying coding errors.

Let's have a look at the world's famous "Hello World" program.

2.1 Hello World!

```
1        begin function main()->(int)
2            printf("Hello World!");
3            return 0;
4
5        end function main
```

Listing 1: The Hello World Program

In Lucas, we define or start a function by simply writing `beign function` followed by the name of the function and by its return type. And we end the function by `end function` followed by the name of the function. In the above example, we can observe how we have defined main function which is followed by its return type and we have ended the main function by `end function main`.

2.2 Strings in Lucas

Lucas uses `charseq` which is a shortened version of "character sequence" which works similar as of `string` in other programming languages.

```
1        begin function main()->(int)
2            charseq ch_seq="Welcome to Lucas";//Charseq is like
    string in cpp
3            printf(ch_seq+"\n");
4            return 0;
5        end function main
```

Listing 2: A basic program

2.3 Looping Constructs

```
1        begin function main()->(int)
2            bigint fact=1;
3            int num=50;
4            begin for(int i=1; i<=num; i++)
5                fact*=i;
```

```
            end for

            print("Factorial(50)="+fact+"\n");//we cannot store the value of 50! in int data type but we can store it in bigint data type

            return 0;

        end function main
```

Listing 3: An application of bigint

In the above program, we highlighted the use case of `bigint`. 50! is a huge number that cannot be stored in `int`, and here comes the role of `bigint` to store that value.
There is one more thing to observe and that is the initializing of `for` loop. In Lucas, we mark the start and the end of the iterative statements in a similar fashion as we do for functions. We use `begin` to mark the start and we use `end` to mark the end. This simple change in how we initialize our functions or iterative statements make a huge impact in debugging large nested loops and functions as we mark and tell the compiler about the exact loop or function we want to end.

2.4 Object-based Programming:

```
        begin class cls
        public:
            begin function fun(int a, int b)->(int, int, int, int)
                int add=a+b;
                int sub=a-b;
                int mul=a*b;
                int div=a/b;

                return (add, sub, mul, div);
            end function fun
        end class cls

        begin function main()->(int)    //This is our main function
            int num1, num2;
            print("Enter two integers to get their sum, difference, product and ratio:\n");
            read(num1, num2);//taking input
            int sum, dif, prod, ratio;
            cls c;
            (sum, dif, prod, frac)=c.fun(num1, num2);

            print("sum="+sum+"::difference="+dif+"::product="+prod+"::ratio="+ratio+"\n");//printing the output

            return 0;
        end function main
```

Listing 4: Returning Multiple Values

The above program is just a representation of how our functions return multiple values and syntax of programs dealing with such functions. In the above program we are just returning the values of results of the 4 arithmetic operations when performed on the given 2 variables.

2.5 Calculus Features:

```
1         begin function main()->(int)//This is our main function
2
3             expr e;
4             print("Enter the variable:\n");
5             read(e.var);
6             print("Enter the expression:\n");
7             read(e);
8             double x_val;
9             print("Enter the value at which the value of expression
     is required:\n");
10            read(x_val);
11            e_val=e.eval(x_val);
12            expr Int=e.intg();
13            expr Dif=e.diff();
14            double bounds[2];//lower bound=bounds[0], and upper
    bound=bounds[1]
15            print("Enter the lower and upper bounds:\n");
16            read(bounds[0], bounds[1]);
17            double Def_Int=e.intg(bounds[0], bounds[1]);
18            double x_of_point;
19            print("Enter the x-coordinate at which the slope is
    required:\n");
20            read(x_of_point);
21            double D_val=e.diff(x_of_point);
22            print("Value of expression="+e_val+"\n");
23            print("Expression after integration:"+Int+"\n");
24            print("Expression after differentiation:"+Dif+"\n");
25            print("Value of Definite Integral="+Def_Int+"\n");
26            print("Slope at given point="+D_val+"\n");
27            return 0;
28        end function main
```

Listing 5: Some functions we provide

The above program represents the core purpose of Lucas-"supporting calculus". The `intg()` and `diff()` functions just returns the expressions, when no arguments were provided and they will return values when they were provided with arguments. The function `eval()`, just evaluates the expression at that particular value.

3 Language Reference Manual

The entire language specification document can be found in our Github Repository.
You may find the link here: Lucas - Language Specification Document

4 Project Plan

4.1 Overview

1. We sought to produce a compiler for our custom-defined DSL, Lucas, which would focus on the ease of use of the calculus features, while seamlessly integrating them into the normal flow of code.

2. Our vision for the project's components was to complete the normal parts of writing a compiler for a programming language and then look into the possibility of writing backend parts to handle the calculus features taking examples from other modern calculus modules for programming languages.

4.2 Lexing and Parsing

1. The overall plan for the project was to finish the lexer and parser with help from existing grammars written for the C language, carefully study those and then implement our own features like the mathematical expression format.

2. We didn't believe that we would have to implement specific features for bigint in the grammar; we instead thought that we could integrate it into the existing structure defined for primitive types and write backend code for it later on.

4.3 Semantic Analysis

1. For semantic analysis, a substantial amount of time was spent on whether we could generate code from the parse tree or not; this idea did not lead anywhere, and resulted in us deciding to hand write the classes for the abstract syntax tree on which to do semantic analysis and code generation.

2. Once we had the classes for the AST ready, the intention was to build custom visitor classes that would walk the parse tree generated by the parser and then either build or perform semantic actions. We would then focus on using this visitor design pattern in order to call LLVM APIs for code generation in LLVM IR.

3. Given more time, we would have invested in working out these details and then eventually optimizing this process, possibly reworking the grammar and AST classes as well in order to accommodate optimizations for this process.

4.4 Testing

1. Exhaustive test cases have been written following some of the standard testing frameworks that are available for the programming languages

2. The original plan was to add to add an exhaustive list of test cases for the features supported exclusively by the Lucas language as well.

3. More details about this is given elaborately in section 8.

5 Language Evolution

5.1 Our Idea

As part of the course, when asked to design a language of our own, we had very few ideas in mind and didn't know what to make. After looking at a few of the sample projects of our seniors, we had this idea to write a language that supports calculus functions on its own. However, we had to make our language stand out, we had to make it different. We came up with some very unique features in our language such as:

5.1.1 Calculus Functions

1. The original plan was to encapsulate all the calculus functionalities that a mathematician might require in his or her computations. The main calculus functionalities that we were looking at included - Differentiation and Integration.

2. So, the idea was to work with simpler polynomial functions which are infinitely differentiable and integrating them over the variable is also pretty simple. The original idea that we had for doing differentiation and integration on other complicated functions was to build upon the idea of Taylor Series Expansions around a point where we are evaluating the differential or integral, and then approximate any function using the polynomial obtained. Once this is done, it is fairly simple to evaluate differentials and integrals of the polynomials, obtained after approximation.

3. The idea was to store the coefficients of the polynomial in a vector. The only catch here is that: the level of precision needed has to be user defined, and the problem that it comes with is, we have to find a way to store the coefficients of the infinite polynomial in an infinite vector and then use appropriate number of degree to get the precision as specified by the user.

5.1.2 Arbitrary Precision Integers

1. Our attempt was to emulate what java does with its own BigInteger datatype, which is store an array of 32 bit integers instead of single digits. operations will then be done on an individual basis for the corresponding elements in the array, with appropriate carry overs implemented as additions into the next element of the operation.

2. Our plan was to begin with this at the implementation level and see the practical limits of such a solution. We will not impose an explicit limit, but if memory issues occur a runtime error or a special inf value can be returned.

5.1.3 Use of begin and end keywords for Scope identification

1. We had planned to use begin and end keywords to track the scope of functions and variables within the code. This was inspired from the famous typesetting system, LaTeX.

2. We had approved of this notion at the start in order to counter the dangling if-else problems we had found in C.

3. Having a visual representation of the start and end points of functions and if-else constructs would help the programmer debug the code easier and make it easier to work with.

5.2 Problems we Faced

As with every group project anywhere in the world, we had run into trouble along the way at various points and had to deal with them in our own ways. We were heavily inspired by the C language but things got slightly difficult for us as we added our own flair to the language. Some of the main issues we faced were:

1. We were heavily inspired by the C grammar and had derived expressions and declaration skeletons based on the C grammar. However, this was too complicated and had to be revised as some of rules were hard to work with during the semantic analysis phase.

2. Although the grammar had been inspired by C, the grammar had been revamped in order to meet our needs. Complicated rules were dropped and redundant rules removed. Keywords were changed and access specifiers modified.

3. We had to revise the choice of keywords and compiler specific methods which were used as part of the GNU GCC compiler as we wanted the Lucas compiler to have its own set of compiler options.

4. Coming up with a grammar for the math expressions which wouldn't hinder the functioning of our compiler while also being disctinct from any other major grammar rule was tricky.

5. After testing the code that we wrote, we could identify some preliminary errors and had updated the grammar to rectify the errors we found.

6. Understanding the dependencies of the nodes and the hierarchy of the AST Nodes was a difficult part to handle and was difficult to implement.

6 Compiler Architecture

Our plan is to basically construct a 2-pass compiler, in which the front end generates the IR from the provided source code input and the back end will generate the target code from the generated IR. ANTLR(version-4) is the main tool we used to construct the compiler.

Like most of the compilers our compiler also consist of several phases like Lexing, Parsing, Semantic Analysis, Intermediate Code Generation. But our Lexer and Parser are a single grammar file which consist of Grammar rules. This is due to the fact that we used Antlr.

6.1 Why ANTLR?

ANTLR makes the work easier for the compiler developers. Following are the advantages of using ANTLR:

- ANTLR will take Grammar as input and generates Lexer/Parser for us.
- Also Antlr can generate Lexers and Parsers in various languages (e.g: C++, C#, Java, Go, PHP, Swift).
- We will have to only deal with the errors in the Grammar, whereas using other tools will also requires us to take care of errors we make in the source language in which we are generating Lexer and Parser.

We used Java as the target language(the language in which the lexer and parser are to be generated) which is the default target language in Antlr.

6.2 Lexing and Parsing:

- Lexing and Parsing is easy if used Antlr.
- We wrote the Grammar of Lucas, and used ANTLR to generate the Lexer and Parser in Java.
- ANTLR uses LL(k) parsing to generate parser.
- Using ANTLR's grun test rig(-gui command) we generated the parse tree representation of some Lucas source codes.

6.3 Semantic Analysis:

- We have written AST classes, to make the tree structure more abstract than the parse tree the ANTLR tool generated.
- We followed the rule "one class for every non-terminal and a subclass for each production rule", while writing the classes. This was to give us a guideline, which we planned to revise and edit later based on ASTs created from test programs.

- We planned to use visitor pattern in the classes we created. The plan was to have separate visitors for separate tasks like building the AST, checking types etc.
- Visitor pattern will allow us to access classes using the visit method. This allows us to have more generality when walking an AST, with the actual Visitor objects taking care of the intended purpose of the tree walk.

6.4 Code Generation:

- We intended to generate LLVM IR from the AST, using certain API calls. This, to our understanding, could be done by having a tree walker walk the AST (making use of the Visitor pattern) and then at each node, calling appropriate API functions to generate IR.
- To implement the backend portion of our calculus features, since the original idea for the structure and function for the calculus features was inspired by SymPy, we thought that we could possibly have computed the expressions there and used an interface to the main Java code.
- We intended to implement our custom bigint datatype using the same way Java's famous BigInteger datatype is implemented. This means storing the integers as arrays of 32 bit integers, with each element of the array being like a "32 bit digit" of the number represented. Operations would then be carried out element wise, with sufficient carryovers. We did not want to impose a specific limit on the size, but planned to return a runtime error or a special value corresponding to infinity if the number became too large. This case seemed unlikely enough that we deemed a limit on the type's size against its spirit.

7 Development Environment

1. Since we used ANTLR4 to generate our lexer/parser, we used the ANTLR4 grammar syntax extension for VSCode to help us write our lexer/parser.
2. We made use of org.antlr.v4.Tool in the antlr-4.11.1-complete.jar to generate the source code of our lexer/parser in java from the antlr grammar we have written.
3. We used openjdk 11.0.17 to compile the generated java classes to generate our lexer/parser
4. We then made use of org.antlr.v4.gui.TestRig in the same jar file to run our lexer/parser on input text to then generate the parse tree on the input text.
5. To test the lexer alone, we used `-tokens` flag while running the antlr TestRig on the input text

6. We made a makefile to automate steps 2,3 and 4 to build the lexer/parser and run it on the given input text, and also to automate running tests on our lexer/parser
7. ANTLR4 generates a concrete syntax tree from our grammar. We intended to write java classes for each non-terminals and production rules in our language grammar.
8. After writing these classes, we intended to define visitors outside of these classes which accesses these classes making use of the visit method in these classes, to generate the ASTs and run semantic checks via AST walks. .

7.1 Branches of the repository

There are 4 branches in our repository

1. **Parser-Grammar**: contains working code of our lexer/parser
2. **main**: same as the Parser-Grammar branch
3. **Test-Cases**: contained parser-grammar + a few test cases (now merged to semantic-analysis)
4. **semantic-analysis**: The most recent branch that is being currently worked on, contains parser-grammar + test cases + few classes for the semantic analysis.

7.2 Directory Structure of the semantic-analysis branch

1. *Folder* **Lexer**:
   (a) *Folder* **documentation:** Contains the presentation and demo of our lexer
   (b) *Folder* **testcases:** Contains a few test cases for the lexer
   (c) *File* **LucasLexer.g4:** the antlr grammar file for lexer
   (d) *File* **README:** Readme file
   (e) *File* **makefile:** the makefile for building the lexer and running it on the testcases in the testcases directory
2. *Folder* **Parser**:
   (a) *Folder* **documentation:** Contains the presentation and demo of our Parser and the parse trees generated for the sample inputs
   (b) *Folder* **testcases:** Contains a few test cases for the parser
   (c) *File* **LucasLexer.g4:** the antlr grammar file for parser
   (d) *File* **makefile:** the makefile for building the parser and running it on the testcases in the testcases directory and generating the parse trees for them.

(e) *File* **test.txt** Sample test file for our parser

3. *Folder* **Semantic:**
   (a) *Folder* **classdefs:** contains the java class definitions for all the non-terminals and production rules in our grammar.
   (b) *File* **LucasLexer.g4:** the antlr grammar file for the lexer/parser
   (c) *File* **makefile:** the makefile for building the lexer/parser and running it on the testcases in the testcases directory, Intended to be used to do semantic analysis on the testcases.
   (d) *File* **test.txt** Sample test case file
4. *Folder* **Test**
5. *File* **.gitignore** The gitignore file
6. *File* **README.md** The readme file for the repository
7. *File* **Team-3_language_specification.pdf** The language specification of LUCAS
8. *File* **antlr-4.11.1-complete.jar** The .jar file by antlr used by the makefiles to generate the lexer/parser and also to generate the parse trees on the test cases.

7.3 Makefile

The makefile in the Lexer folder, does lexical analysis only, while the makefile in the Parser folder, does both lexing and parsing on the test cases. All the makefiles make use of the *antlr-4.11.1-complete.jar* in the central directory.

```
antlrlib := ../antlr-4.11.1-complete.jar
TEST_DIR := testcases
antlr4 := java -Xmx500M -cp "$(antlrlib):$$CLASSPATH" org.antlr.v4.
    Tool
grun := java -Xmx500M -cp "$(antlrlib):$$CLASSPATH" org.antlr.v4.
    gui.TestRig
```

The above commands allows us to make use of the *org.antlr.v4.Tool* to generate the lexer and parser, *org.antlr.v4.gui.TestRig* to test our lexer/parser on test cases and generating the parse trees.

```
$(antlr4) LucasLexer.g4
javac -classpath "$(antlrlib):$$CLASSPATH" LucasLexer*.java
```

The above commands runs the *org.antlr.v4.Tool* on our grammar file, to generate the java files, which are then compiled to generate our lexer/parser.

```
 $(grun) LucasLexer tokens -tokens < $(TEST_DIR)/test1.txt
```

The above command runs the *org.antlr.v4.gui.TestRig* with our generated lexer to lexically analyse the *test1.txt*, while,

```
$(grun) LucasGrammar compilationUnit $(TEST_DIR)/Test_case1_parser.
    txt -gui
```

the above command runs the *org.antlr.v4.gui.TestRig* with our generated lexer/parser to parse the *Test_case1_parser.txt* with the compilationUnit as the starting non-terminal.

8 Test Plans and Test Suits

- We have written test cases accompanying the lexer and parser to test various aspects of the language grammar. These test cases are present in the Test directory and can be run using the provided make file.
- The tester module's *tester* begins by taking the *antlr-4.11.1-complete.jar* file and generating the lexer and parser using the given rules and grammar file. It then iterates through the sub directory, TestCases, performs lexical analysis on the .luc files present in the directory and generates the parse trees for them.
- Make commands are provided if the user wants to test a specific piece of code. The user may name the file to be tested as testsample.luc and run the make command testsample to test the file and generate a parse tree structure. Note that -gui compile option may be used to get the visualization of the generated parse tree.
- The built-in test cases try to cover all aspects of the language. These include data types including arrays and classes, statements like assignment, selection and looping, pointer arithmetic, operators and language specific features like multiple return types, labels and calculus functionalities.
- Note that the test cases have a naming convention as TS followed by the test case number and a brief description regarding what aspect of the language is tested. The set contains both positive and negative test cases and were written in the black-box fashion as in language manual was strictly followed to write the test cases without help from the underlying grammar.
- Given below is a simple test case `TS2-main_return0.luc` which is an empty main block with a return statement and the parse tree generated for the same.

```
begin function main()->(int)
    return 0;
end function main
```

Listing 6: TS2-main_return0.luc

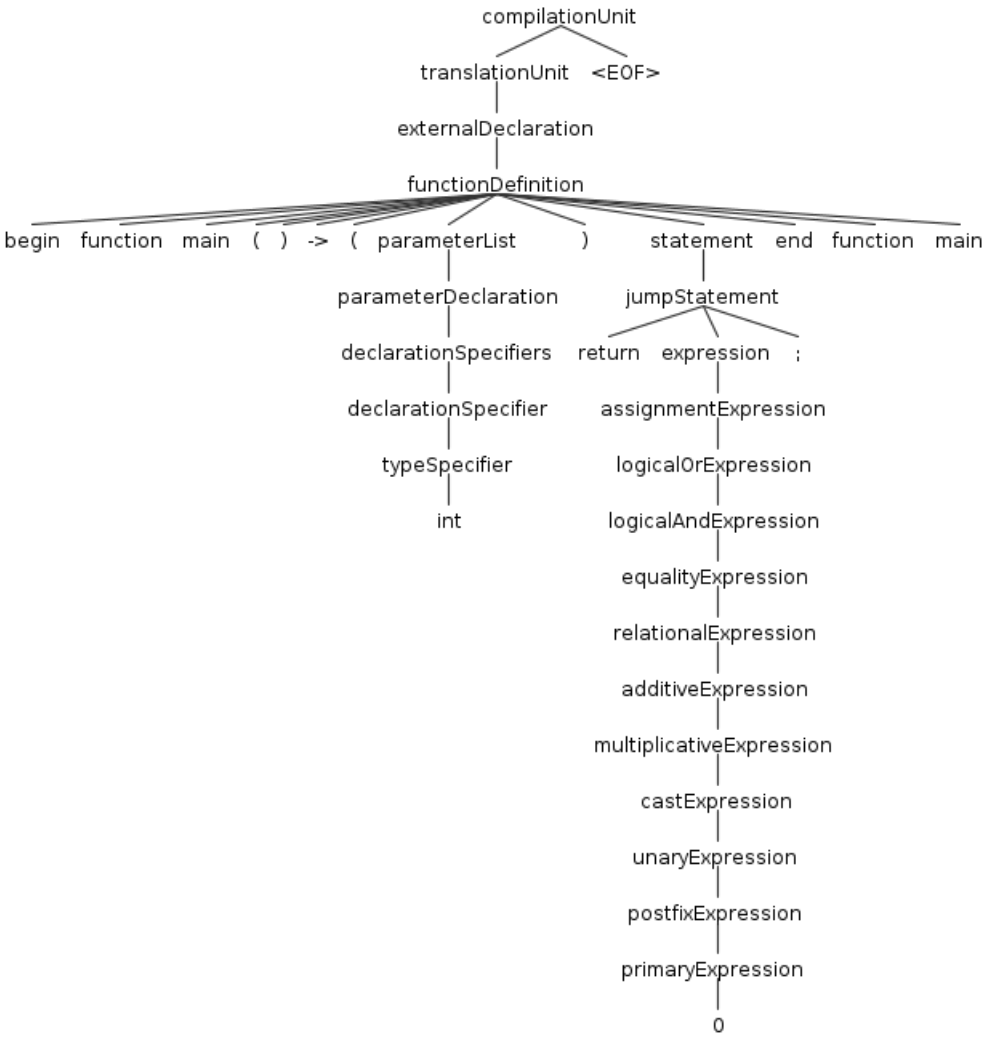

Figure 1: Parse tree for TS2-main_return0.luc

9 Conclusions containing lessons learned

1. Gained a lot more clarity about lexical analysis by creating a lexer for LUCAS.
2. Since, Antlr is a LL(1^*) parser, and not a LALR parser like yacc, we faced a few problems with indirect left recursion (ANTLR can handle direct left recursion, but not indirect left recursion), we had to change the grammar slightly to accommodate this. So, we gained a better understanding of LL and LR parsing, fixing left recursions, etc.
3. Learned about making makefiles and how to use them to build and automate the testing
4. Gained understanding about Concrete and Abstract Syntax trees, especially their differences with respect to amount of information contained in each tree's individual nodes.
5. We gained knowledge of common semantic analysis paradigms, like AST construction using visitor patterns, type checking using tree walks, etc.

10 Appendix

10.1 References

1. COOL Manual
2. Go Manual
3. Sympy - Symbolic computing in Python
4. Open ACC Dialect for MLIR
5. ANTLR Documentation
6. Sample ANTLR Grammars for known programming languages
7. Creating ASTs with ANTLR4
8. Exhaustive List of Test Cases for C Language