Design Document for DevilsCode

Team 5

Iyer, Hari Singh, Pankaj Yang, Tingyeu Yang, Mingfei

1. Language Name: DevilsCode

2. Language Design:

Data Type	int (integer), bool (boolean)
Binary Arithmetic Operations	+, -, *, /
Assignment	=
Comparison	==, <, >, <=, >=
Condition	if (expr) { code_block } [Optional] else { code_block }
Iteration	while (expr) { code_block }
Reserved Keywords	int, bool, if, else, while, true, false, print
Symbols	{, }, (,), ;, +, -, *, /, =, ==, <, <=, >, >=, whitespace, newline
Identifiers	Should begin as a non-digit character

Table.1 Basic Feature & Attributes for DevilsCode

```
main ()                         //main() indicates start of program code
{                               //Scope of program begins
        int f = 1;              //Assignment operations
        int i = 1;
        while (i <= 5)          //Loop begins
        {
                f = f * i;      //Calculations
                i = i + 1;
        }
        if (i > 5)              //Conditional statement
        {
                print f;
                                //writing output to the prolog program,
                                //which will in turn print the output to the console
        }
}                               //End of program scope
```

An example of DevilsCode: generate the factorial of a number

3. Grammar Rules

Used characters

<digit> ::= '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
<non_digit> ::= 'a' | 'b' | … | 'z'
<number> ::= <number> <digit> | <digit>
<cpr_symbol> ::= '<' | '<=' | '>' | '>=' | '=='
<data_type> ::= int | bool
<bool_value> ::= true | false

Statement

<stmt> ::= <expr> ';'
| <dec_stmt> ';'
| <while_stmt>
| <if_stmt>
| '{' <stmts> '}'

<stmts> ::= <stmts> <stmt> | ε

Expression

<expr> ::= <id>
| <assign_expr>
| <math_expr>
| <cpr_expr>

Declaration

<dec_stmt> ::= <data_type> <id> | <data_type> <assign_expr>

Identifier

<id> ::= <non_digit> | <id> <non_digit> | <id> <digit>

Assignment expression

<assign_expr> ::= <id> ‘=’ <expr>

Mathematical expression

<math_expr>. ::= <math_expr> '+' <math_term>
| <math_expr> '-' <math_term>
| <math_term>

<math_term> ::= <math_term> '*' <math_factor>
| <math_term> '/' <math_factor>
| <math_factor>

<math_factor> ::= <number> | '(' <math_expr> ')'

while-loop statement

<while_stmt> ::= while '(' <expr> ')' <stmt>

if statement

<if_stmt> ::= if '(' <expr> ')' <stmt>
| if '(' <expr> ')' <stmt> else <stmt>

Comparison expression

<cpr_expr> ::= <cpr_term> <cpr_symbol> <cpr_term>
<cpr_term> ::= <id> | <number> | '(' <math_expr> ')'

4. Compiler Architecture:

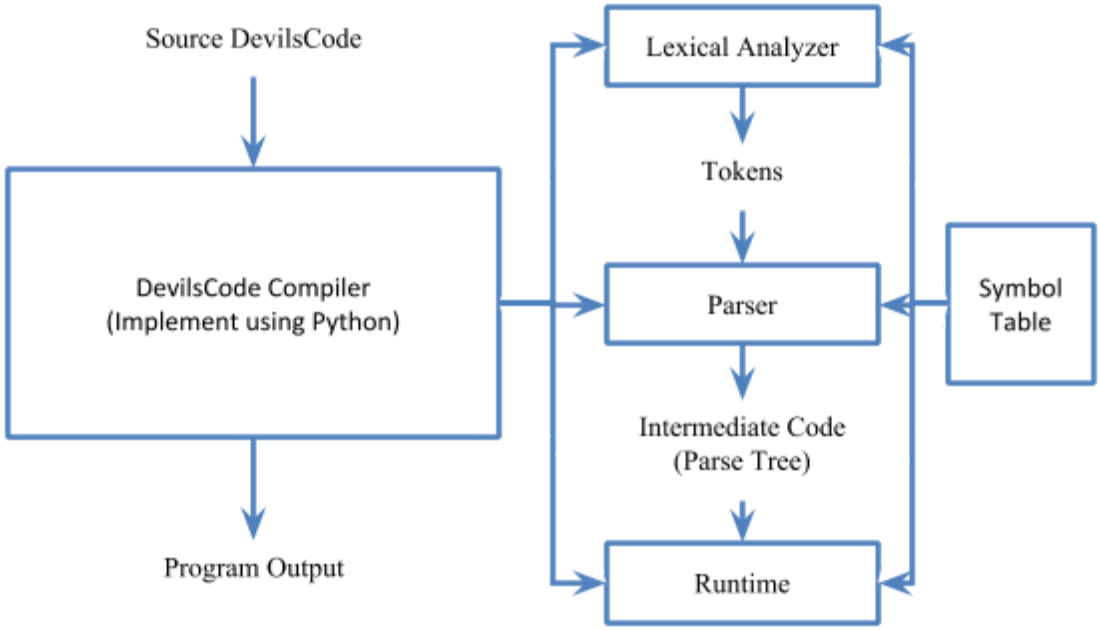

Fig.1 Architecture of the Compiler

5. Workflow for Program Execution:

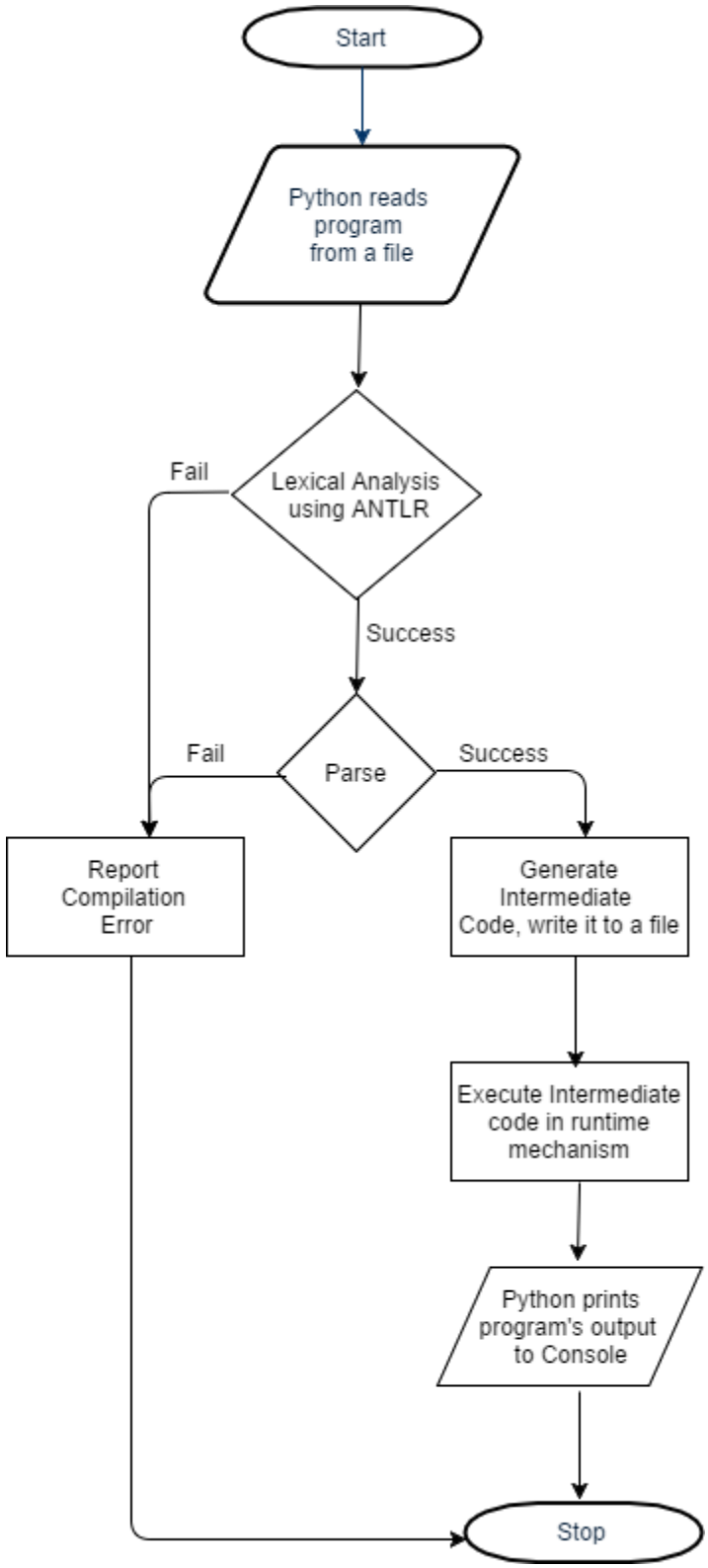

Fig.2 Workflow

6. ANTLR Integration with Python:

In the use case requirement, ANTLR would leverage the following two implementation aspects:

1. Lexical Analysis of the DevilsCode program.
2. Parse Tree Generation.

The prestored grammar of DevilsCode that has been discussed further in this file will be fed to ANTLR. This would enable it to have a context of how the input program should be parsed as. This grammar file is turned into two parts by ANTLR, namely Lexer and Parser.

The Lexer reads the DevilsCode program as an input stream from a file, and tokenizes it for further processing. This is fed into the Parser, which in turn relates the tokens to the Grammar that ANTLR has been made to assume true for the language. The Parser generates a parse tree out of the input, which will be an intermediate code to be sent to the runtime in order to generate the desired output. The runtime would be implemented in Python, with the input as the above-mentioned intermediate code. It would access the instructions based on the format of data structure in which the code is stored. The output after execution in the runtime environment is echoed back to Python, which will in turn print it out to the console.

7. Data Structures to be used:

Trees: Trees will be used by the parser to create parse trees. The parser will parse the source code and generate tokens which will be stored as a parse trees. Each expression would be stored in Inorder tree format of the tokens. Interpreter would use these parse trees to provide semantics to the expressions and store the outputs.

Dictionary: The symbol table will be implemented with a dictionary. The parser and Interpreter use dictionaries to transform source code to intermediate code. Variables will be stored as keys in the symbol table to provide the data values they store.

Lists: Separate Lists will be used to store the keywords, identifiers and mathematical symbols. In fact the whole program will be a list of statements. Lists will be useful in implementing and accesing Grammar during parsing and interpretation.