Final Project: ChucK Compiler

For our final project, we decided to create a compiler for our own language, ChucK. ChucK is a strongly-typed programming language similar to Java or C. The use-case for this language is teaching basic information security concepts. ChucK features easy-to-use functionalities that support Caesar Cypher, Vigenere Cypher, and Frequency Analysis. The Caesar Cypher transforms text by shifting letters based on a given shift amount. The Vigenere Cypher transforms text by shifting letters based on an amount determined by the key text. Frequency Analysis calculates the frequency of each character in a text.

We implemented various language constructs to be supported by ChucK. We implemented programs, functions, statements, expressions, factors, data types, and comments.

Programs

Programs represent a block of code that can contain any of the other components.

Functions

A program can contain one or more functions, and each function has a return type and optional parameters. Functions are pass-by-value.

Statements

Statements can be a compound statement (a list of statements), an assignment statement, an if-then-else statement, a while loop statement, a print statement, a function definition statement, a function call statement, a variable declaration statement, or an empty statement.

Expressions

Expressions have layers to incorporate operator precedence. At the top, expressions contain one or more relational expressions joined by cypher operators. The cypher operators are ‘<<’ for the built-in Vigenere Cypher and ‘>>’ for the built-in Caesar Cypher. At the next level, relational expressions contain one or more simple expressions joined by relational operators, which test for equality. At the next level, simple expressions contain one or more terms joined by additive operators, which perform addition, subtraction, and logical OR. At the next level, terms contain one or more factors joined by multiplicative operators, which perform multiplication, division, and logical DIV, MOD, and AND.

Factors

Factors can be a variable, number, character, string, function call, expression, logical NOT, or string analysis. String analysis uses the operator ‘@’ for the built-in Frequency Analysis.

Data Types

Data types are Integer, represented by ‘I’, Double, represented by ‘D’, and String, represented by ‘S’.

Comments

Comments are enclosed by the ‘%’ symbol on both sides.

Our ChucK compiler generates Jasmin object code. The following are excerpts from generated Jasmin object code for key statements in ChucK.

Program

ChucK:

```
P sample1[parameters]
```

Jasmin:

```
.method public static main([Ljava/lang/String;)V
.var 0 is args [Ljava/lang/String;
.var 1 is _start Ljava/time/Instant;
.var 2 is _end Ljava/time/Instant;
.var 3 is _elapsed J
        invokestatic    java/time/Instant/now()Ljava/time/Instant;
        astore_1
```

Function

ChucK:

```
F S deshift[I amount, S cyphertext]
```

Jasmin:

```
.method private static deshift(ILjava/lang/String;)Ljava/lang/String;
.var 0 is amount I
.var 1 is cyphertext Ljava/lang/String;
.var 2 is deshift Ljava/lang/String;
```

Variable Declaration Statement, Assignment Statement

ChucK:

```
S plain; plain = ‘the quick red fox jumped over the lazy brown dog’;
```

Jasmin:

```
        ldc     "the quick red fox jumped over the lazy brown dog"
        putstatic       sample1/plain Ljava/lang/String;
```

If-Then-Else Statement, Relational Expression, Print Statement

ChucK:

```
if[plain == result] {print[plainTextAnalysis];} else {print['bad logic!'];}
```

Jasmin:

```
        getstatic       sampleSmall/plain Ljava/lang/String;
        getstatic       sampleSmall/result Ljava/lang/String;
        invokevirtual   java/lang/String.compareTo(Ljava/lang/String;)I
        ifeq    L007
        iconst_0
        goto    L008
L007:
        iconst_1
L008:
        ifne    L005
        getstatic       java/lang/System/out Ljava/io/PrintStream;
        ldc     "bad logic!"
        invokevirtual   java/io/PrintStream/print(Ljava/lang/String;)V
        goto    L006
L005:
        getstatic       java/lang/System/out Ljava/io/PrintStream;
        getstatic       sampleSmall/plaintextanalysis Ljava/lang/String;
        invokevirtual   java/io/PrintStream/print(Ljava/lang/String;)V
L006: …
```

While Loop Statement, Relational Expression, Simple Expression, Term, Print Statement

ChucK:

```
while[i< 10] {i = i + 1; j = j * i; print[j];}
```

Jasmin:

```
L001:
        getstatic       sampleSmall/i I
        bipush  10
        if_icmplt       L003
        iconst_0
        goto    L004
L003:
        iconst_1
L004:
        ifeq    L002
        getstatic       sampleSmall/i I
        iconst_1
        iadd
        putstatic       sampleSmall/i I
        getstatic       sampleSmall/j F
        getstatic       sampleSmall/i I
        i2f
        fmul
        putstatic       sampleSmall/j F
        getstatic       java/lang/System/out Ljava/io/PrintStream;
        getstatic       sampleSmall/j F
        invokevirtual   java/io/PrintStream/print(F)V
        goto    L001
L002: …
```

Function Call Statement

ChucK:

```
S result; result = deshift[2, cypher];
```

Jasmin:

```
iconst_2
getstatic       sampleSmall/cypher Ljava/lang/String;
invokestatic    sampleSmall/deshift(ILjava/lang/String;)Ljava/lang/String;
putstatic       sampleSmall/result Ljava/lang/String;
```

Expression (Cypher Operator '<<')

ChucK:

```
S polyalphabet; polyalphabet = plain << 'polyalphabet';
```

Jasmin:

```
getstatic       sampleSmall/plain Ljava/lang/String;
ldc     "polyalphabet"
invokestatic    cypher/poly(Ljava/lang/String;Ljava/lang/String;)Ljava/lang/String;
putstatic       sampleSmall/polyalphabet Ljava/lang/String;
```

Expression (Cypher Operator '>>')

ChucK:

```
S cypher; cypher = plain >> 2;
```

Jasmin:

```
getstatic       sampleSmall/plain Ljava/lang/String;
iconst_2
invokestatic    cypher/shift(Ljava/lang/String;I)Ljava/lang/String;
putstatic       sampleSmall/cypher Ljava/lang/String;
```

Frequency Analysis

ChucK:

```
polyalphabetAnalysis = polyalphabet @;
```

Jasmin:

```
getstatic       sampleSmall/polyalphabet Ljava/lang/String;
invokestatic    cypher/analysis(Ljava/lang/String;)Ljava/lang/String;
putstatic       sampleSmall/polyalphabetanalysis Ljava/lang/String;
```

We have included sample programs titled sample.ck, sample1.ck, sample2.ck, sample3.ck, and sampleSmall.ck. Follow the following instructions to build and run our ChucK compiler:

1. Import program files into IDE of your choice (preferably Eclipse).

2. Right-click on the CK.g4 grammar file, and generate ANTLR recognizer.

3. Open a terminal and run 'javac cypher.java' to create the cypher.class file containing the library methods.

4. Create a run configuration with the main class set to Pascal.java. Set the program arguments to be:

```
-compile sample1.ck
```

5. Run the run configuration. This will print the results of the Syntax pass, Semantics pass, and Compilation pass to the console. This will also create a Jasmin file sample1.j.

6. Open a terminal and run 'java -jar jasmin.jar sample1.j' to generate the sample1.class file.

7. Still in the terminal, run 'java sample1' to execute the program.

8. Repeat steps 4-7 for all the sample programs.