CAPS Documentation

Group Members: Mitchell Berbera, Junhe Chen, Lissett Laguna, Jasmine Maquindang, Jorge Verduzco

Why this language?

- We chose to write our compiler in Java because it was the language that we were most familiar with as a group. The object-oriented style that Java provides also allowed us to structure our compiler in a more organized and efficient way.
- Our target language was C, so we were going for a compiler that would go from high level to low level for the sake of making it easier on the user.
- The key features include working with type inferences, complex-syntax, and higher-order functions.

Why this language design?

- CAPS was mainly designed in a way that keeps a beginner programmer in mind.
- It relies heavily on reserved words to remind the user directly of how they are manipulating their data.
- All reserved words are written in uppercase letters to stand out and also to differentiate between elements like variables, types and loops.

Code snippets and features.

EXAMPLE CODE:

```
x IS 5
y IS 8
sum IS x + y
PRINT sum Output => 13
--------
DEFINE NUMBER addnumbers (NUMBER x, NUMBER y) {
NUMBER sum IS x + y;
RETURN sum; }

NUMBER x IS 5
NUMBER y IS 8
CALL addnumbers Output => 13
```

```
--------
WHILE (X<10) {
x++; y++;
}
--------
IF (X < 10) "foo" ELSE "bar"
```

- The first example code shows a short program that prints the sum of two numbers.
- Variables are declared by combining the variable name and the value using the reserved word “IS”.
- Reserved word “PRINT” is used to display the final value of sum, in this case 13
- The second program demonstrates the use of functions. Functions are written using the keyword “DEFINE” followed by the type in capital letters, then we write the type and variable names between two parentheses, separated by a comma. The type being used in this case is “NUMBER”.
- “RETURN” is written in all caps to show it is a reserved word. This will return the value of sum to be used elsewhere.
- The reserved word “CALL” makes a call to the function “addnumbers” that was declared above
- These reserved words are meant to keep things simple and provide a more straightforward way to keep track of what the user is working with.
- Loops are done by including the type of loop you are writing, which could be WHILE, or IF…ELSE statements
- The basic structure of this example loop is to use the word “WHILE” followed by an expression between two parentheses. Here the expression is X < 10 and it is followed by the statement that will run as long as the condition is true.
- The last example is an IF…ELSE statement that is followed by an expression between two parentheses, it is then followed by the statements without including the curly brackets. This is another way of writing loops in a more simple way.

Feature: Complex-syntax

One of the features in our language is complex-syntax.
An example of complex syntax in our parser would be:

```
public ParseResult<Stmt> parseStmt(final int p) throws ParseException{
   try {
      // try parsing a variable declaration statement
             return parseVardec(p);
         } catch(final ParseException e1){
             try {
            // if parsing a variable declaration fails, try parsing a
            return statement

           return parseReturns(p);

       }catch(final ParseException e2){
           try{
// if parsing return statement fails, try parsing a loop statement
               return parseLoop(p);
           }catch(final ParseException e3){
               try{

//if parsing a loop statement fails, try parsing an if statement
                   return parseIf(p);
               }catch(final ParseException e4){
                   try{
//if parsing an if statement fails, try parsing a sequence of statements
                       return parseStmts(p);
                   }catch(final ParseException e5){
                       try{
// if parsing a sequence of statements fails, try parsing an expression
statement
                           return parseExpStmt(p);
                       }catch(final ParseException e6){
//if all parsing attempts fail, parse a progn (a group of expressions) as
the fallback
                           return parseProgn(p);
                       }
                   }
               }
           }
```

```
        }
    }
} // parse Stmt
```

- The above nested try and catch block is an example of a complex syntax in our Java parser.
    - The code uses a cascading approach where it tries different parsing strategies sequentially, falling back to the next strategy if the previous one fails.
    - Each block tries to parse a specific type of statement such as variable declaration, return statement, loop statement, if statement, expression statement, etc.
    - The last catch block servers as a final fall back, returning the result of parsing a progn if all other attempts fail.

Known limitations.

- CAPS currently does not include deallocation of memory since there is no garbage collection involved.
- Our compiler currently only has the tokenizer and the parser available, due to time constraints in the course.

Knowing what you know now, what would you do differently?

- We used mainly intelliJ to create our tokenizer and parser. It was useful in the sense that it allowed us a lot of shortcuts when it came to all the repeated code we needed to do. It also had a simple way to write the tests and check for code coverage.
- On the other hand, a few of us were more familiar with VS Code in general, so we think that it might have been easier for us to use with our compiler.
- In regards to our target language C. We would maintain the same target language, as we found the asynchronous videos on the canvas page to be highly beneficial. Since the example language was similar to ours, written in Java and had C as the target language, it helped us get a better understanding of both the tokenizer and the parser as we were following it along.
- One thing we could have done differently was put more effort into writing out example code to better understand how we wanted our language to turn out. Due to not prioritizing example code we found that we had to go back and change our

grammar syntax a few times in order to remove any contradictions with our code versus what was actually written within our grammar syntax.

How do I compile and run your compiler?

- We have the dependencies for Maven included in the POM.xml hence we use the “mvn compile” in the command line or select “ build project” in the menu.
- We then used “mvn test” to run and check all of our tests or by using the run all tests option in intelliJ

Formal syntax definition:

var is a variable

string is a string

number is a number (integer or double)

methodname is the name of a method

```
program ::= stmt                                  // entry point

vars ::= [var (`,` var)* ]                        // list of variables

type ::= `BOOLEAN` | `NUMBER` | `STRING` |        // built-in datatypes

      `(` types `)` `RETURNS` type                // higher-order function type

types ::= [type (`,` type)*]                      // list of types

param ::= type var

params ::= [param (`,` param)*]                   // list of parameters

op ::= `+` | `-` | `*` | `/` |                    // arithmetic operators

      `&&` | `||` |                               // logical operators

       `<` | `>`                                  // comparison operators

exp ::= var | string | number |                   // variables, strings, and numbers are
expressions
```

```
    `(` exp op exp `)` |                    // arithmetic expressions

   `(` vars `)` `EXECUTES` exp |               // defines higher-order functions
(execute)

   `PRINT` exp                          // prints to terminal, returns a number (print)

    `CALL` exp `(` exps `)` |                // calls higher-order function (call)

stmt ::= var `IS` exp`;` |                    // variable declaration (vardec)

   `RETURNS` exp`;` |                       // return statement (returns)

   `WHILE` `(` exp `)` |                    // loop statement (loop)

   `IF` `(` exp `)` stmt `ELSE` `stmt`          // if-else statement (if)

   `{` stmt* `}`|                        // block statements (stmts)

   `DEFINE` type methodname `(` params `)` `{` stmt* `}`   // method declaration
(methoddef)

   `{` `PROGN` stmt* `}`

   exp`;` |                            // expression statement (expstmt)
```