COMP2212 - PROGRAMMING LANGUAGE CONCEPTS REPORT

Jordan Jones - jj1g18

Kritagya Gurung - kg4g18

May 8, 2020

Introduction to JORKRI

Style of programming

Our language is a fusion of the Haskell programming style as well as the Imperative approach. We did this because in our opinion, it blends the simplicity of Haskell and the clarity of Imperative, although we have an emphasis on brackets which may be unnecessary and too verbose as a consequence of clarity. This can be seen in Figure 1a and Figure 1b.

```
j = 0 !
while (j < length(Stream) ) then {
stream1 ++ Stream index j !
stream2 ++ Stream index (j + 1) !
j = j + 2 !
}
```

(a) A blend of Haskell and Imperative style

```
1  List digits !
2  Int pointer = 0 !
3  while (pointer < 10) then {
4      digits ++ pointer !
5      pointer = pointer + 1 !
6  }
7  OUTPUT ( digits ) !
```

(b) Prints numbers 0-9

Figure 1: Examples of Programs

Main features of the language

- We have Lists, Integers (positive and negative) and Boolean conditions (True and False).
- We have Variables which can be assigned as a List of Integers or just an Integer.
- We have some innate functions and if/while statements integrated into the language already.
- We have a end of line (EOL) character "!" used after every expression.

Generic information of JORKRI

Innate Functions

Assigning variables, indexing to retrieve an element, concatenation of an element to a list, add, subtract, multiply, modulus, div (to closest integer), exponents, less than, more than, equal to, not, OUTPUT, if statement, while statement, logical and (&&) , logical or (||), length of a List.

Typing

Weakly and in theory, Statically typed. Initially, you have to explicitly declare "List" to make a list and have to declare Integers as "Int" (but since the language only dealt with integers, it is no longer required to declare the type of the variable). But, as we never implemented a Type checker, we cannot claim to be Statically typed but perhaps Dynamically typed.

Error handling

We do have some error messages, such as when you are trying to find variables that do not exist, however in our short-sightedness, we have not updated the error messages to be more helpful (for example "uh oh list not here"). Additionally, we have forgotten to reap the benefits of using the Posn wrapper. However, errors without a message (e.g. Parsing errors) will display where the error was caused but not the reason why it was caused.

Scoping
As there is no way to define functions, all variables and lists are in the global environment. This means that they can be accessed anywhere and everywhere. This should be fine for smaller programs but could be an issue in larger ones.
Type checking
There is no explicit type checking and no error messages when types are mismatched, e.g. our variables are always a list of Integers or just an Integer, but can still have a Boolean variable assigned to it which which cannot be accessed. There is some implicit type checking as a result of how the grammar was constructed since Boolean expressions (referred to as "Conditions") have their own grammar rule set.

Lexer

Examples of lexemes

```
for    { tok (\p s -> TokenFor p) }
while  { tok (\p s -> TokenWhile p) }
if     { tok (\p s -> TokenIf p) }
then   { tok (\p s -> TokenThen p) }
else   { tok (\p s -> TokenElse p) }
\:     { tok (\p s -> TokenHasType p) }
not    { tok (\p s -> TokenNot p) }
```

(a) Lexemes for keywords

```
\+     { tok (\p s -> TokenPlus p) }
\-     { tok (\p s -> TokenSubtract p) }
\*     { tok (\p s -> TokenMultiply p) }
\/     { tok (\p s -> TokenDivide p) }
\^     { tok (\p s -> TokenPower p)}
div    { tok (\p s -> TokenDiv p)}
mod    { tok (\p s -> TokenMod p)}
```

(b) Some lexemes for operators

Figure 2: Examples of lexemes

Posn wrapper
We used this wrapper as it provided vital information to where the error was, however due to our ignorance, we never fully made use of this wonderful feature in our own custom error messages. These positional error messages can only be seen wherever there is a lexer or parse error.
Justifications for Lexer design
We have many lexemes that are not used for the final language but still exist since we weren't sure if we needed them for later functionality. For example, in Figure 2a, `else` is not used anywhere in the program since you could just use the complementary if statement (it's syntactic sugar).

Grammar

The rule-set
There is Program, Line, Condition, Exp, Factor and (Deprecated) Type rule sets. You can see the rest Grammar rules in Figure 3.

- Program rule-set is responsible for recursively looping through the program
- Line rule-set is responsible for variable manipulation and key statements such as if or OUTPUT
- Condition is responsible for the Boolean expressions for the if and while statements
- Exp is responsible for Variable operations such as addition or multiplication
- Type was responsible to declare types of variables however, the program only uses Integers thus was quickly dropped
  We have 5 Unused terminals and 44 shift/reduce conflicts.

Justification for rule-set
Having it all under one Grammar type "Exp" would cause a lot of shift/reduce conflicts and perhaps some reduce/reduce conflicts, therefore we decided to break it up into multiple grammar rule-sets. Additionally, it is easier to follow the control flow as each rule-set has an appropriate name e.g Condition always results in a Boolean.

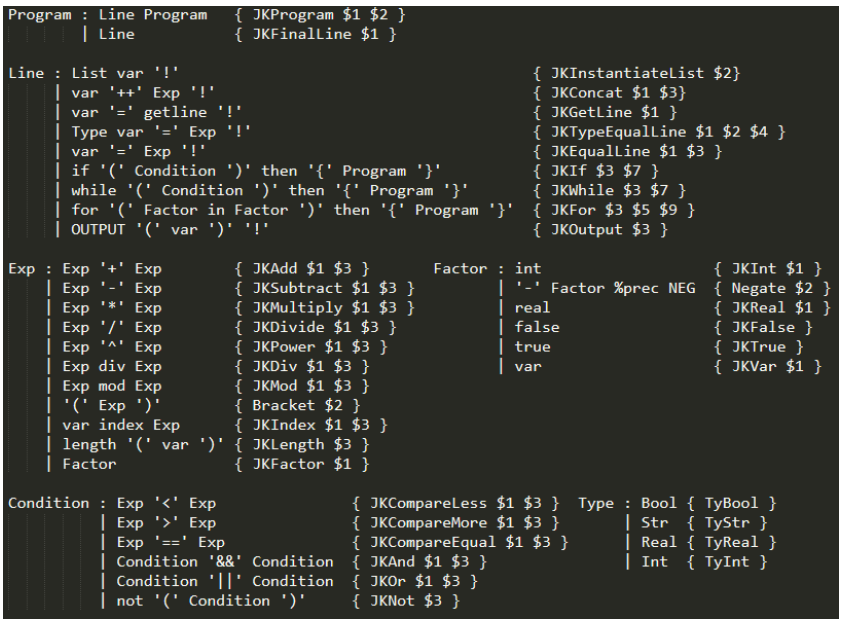

Figure 3: Grammar rules in our Language

```
type Environment = [ (String, Int) ]
type ListEnvironment = [ (String, [Int]) ]

type State = (Program, Environment, ListEnvironment)
type CState = (Condition, Environment, ListEnvironment)
type EState = (Exp, Environment, ListEnvironment)
type FState = (Factor, Environment, ListEnvironment)
```

Figure 4: Environments and States in our Evaluator

Example and Program execution

We will be focusing on the program seen in Figure 1b and the Environment and states can be seen in Figure 4. The Environment stores the string (variable name) and the associated value stored to it, while the ListEnvironment stores the string (name of list) and the associated list of values. Environments and expressions are stored in states within the program, named State, CState, EState and FState for, line, condition, expression and factor respectively. They store the expression to be evaluated and the current environments in the program. These Environments are in the state so that they can be passed on and updated as the program executes and allows the program to access these stored values.

- The first line `List digits !` evaluates to adding a new list to the list environment and is automatically set as the empty list.
- Next, `pointer = 0 !` (could also be written as `Int pointer = 0 !` for syntactic sugar) evaluates to adding a new variable to the environment and in this case, pointer is assigned to have a value of 0.
- After that, `while (pointer < 10) then {lines 4,5,6}` is the next line that needs evaluating. The parser recognises that this follows the rule `while (condition) then { Program }` and then proceeds to evaluate the Condition term (which evaluates to a Boolean) and decides whether to execute the Program (referring to code within the while loop). Since pointer's value is set to 0 and 0 < 10, then the code within the while loop is executed.
- The line `digits ++ pointer !` appends the value of pointer (0 at the moment) to the list called digits.
- Following that, `pointer = pointer + 1 !` then increments the value of pointer by one.
- The program within the while will keep on being executed until the condition is no longer met (and thus should add numbers from 0 to 9 to the digits list).
- Finally, `OUTPUT ( digits ) !` will evaluate to printing all the elements of the digits list one at a time, beginning from the start of the List to stdout.