Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Homework 2

Due: March 30, 2021 - Tuesday @ 23:55

1 Introduction

In this homework you will write a context free grammar and implement a simple parser for JISP language for which you designed a scanner in the previous homework. Note that, there might be differences between the syntax of JISP language given in the previous homework and this one. Therefore, take the explanations on the syntax of JISP language given in this homework.

The language that will be generated by your grammar and other requirements of the homework are explained below.

2 JISP Language

The grammar you will design needs to generate the JISP language as described below. Here is an example program written in this language. This is to give you an idea how a JISP program looks like.

```
[

    ["Print", ['This is a function to print hello world on screen!']]

    ["Set", "PrintHelloWorld",
        ["Function", [],
         [
            ["Print", ['Hello World'] ]
         ]
        ]
    ]

    ["Print", ['This is a function to add X and Y taken as parameters and
              returns it.'] ]

    ["Set", "SumFunc",
```

```
    ["Function", ["X","Y"],
      [
        ["Set", "Z", ["+" , ["Get", "X"], ["Get", "Y"]]]

        ["Return", ["Get", "Z"]]
      ]
    ]
]


["Print", ['Define a new Variable and name it which_function_to_call.']]

["Set", "which_function_to_call", 1]

["Print", ['If the which_function_to_call variable is 1 then call PrintHelloWorld
otherwise call the function sumFunc.'] ]

["If",
    ["==",["Get","which_function_to_call"],1],
    [
        ["Print", ['Condition holds']]
        ["Get", "PrintHelloWorld", []]
    ]
    [
        ["Print", ['Condition does not hold']]
        ["Get", "SumFunc", [1,2]]
    ]
]

["Print", ['Define a new variable OnePlusTwo and set it to return value of SumFunc
when called with 1 and 2.']]

["Set", "OnePlusTwo", ["Get", "SumFunc", [1,2]]]

[">", 'Hello', 2]

["++", "HelloWorld"]

["--", "which_function_to_call"]

["Set", "X", ["Get","PrintHelloWorld",[] ] ]

["Get", "SumFunc", [1,2,3,4,5] ]

["Get", "which_function_to_call", [1,2]]
```

```
["Get", "PrintHelloWorld"]
]
```

Note that the set of statements given above does not have to make sense. It may even have errors as a JISP program. For example:

- There can be a usage of an undeclared variable.
- There can be a type mismatch, for instance: `[">", 'Ali', 2]`
- A function which is declared with 2 parameters can be called with 5 parameters, or a variable which is not even a function is called with parameters, or a function is called as if it is a string or number variable.

We have some examples of such errors in the example JISP program given above. However, in this homework we will not aim to detect such erros. In this homework, we will aim to implement a parser for the basic syntax rules of the JISP language that are given in detail below.

1. A JISP program starts with a left bracket, ends with a right bracket, and in between, there is a sequence of statements. Note that an empty program (i.e. a program with only a left and a right bracket) is also considered a valid JISP program. In this case, the sequence of statements is just an empty sequence.
2. Each statement in our JISP can be a `"Set"` statement, or an `"If"` statement, or a `"Print"` statement, or an increment statement, or a decrement statement, or a return statement. In addition to these statements, an expression can also be used as a statement.
3. A `"Set"` statement is used to assign a variable a value. It is basically the assignment statement for JISP. `"Set"` statement is starts with a left bracket, followed by the keyword `"Set"` followed by a comma, followed by name of the variable, followed by a comma, followed by an expression, and followed by a right bracket. Below are some examples of setting a variable:
   - A number.
     ```
     ["Set","X", 1]
     ```
   - A string
     ```
     ["Set","X", 'Test']
     ```
   - A function
     ```
     ["Set","X", ["Function", [], []] ]
     ```

Note that, the last example is an example of assigning a variable to a function value, i.e. a new function is declared (which does not have a name), and this function is assigned as the value of a variable.

4. An `"If"` statement starts with a left bracket, and ends with a right bracket. Between the brackets, we first have the keyword `"If"`, followed by a comma, followed by a condition, followed by another comma, followed by a "then" part, and optionally followed by an "else" part. The "then" part starts with a left bracket and ends with a right bracket. Between the brackets of the "then" part there will be a sequence of statements. This sequence can be an empty sequence. If the optional "else" part exists, this part also starts with a left bracket and ends with a right bracket. Between these brackets there will be a sequence of statements, which can be an empty sequence as well.

   - An if statement with the else part;

   ```
   ["If", [">", 1, 0],
       [
           ["Set", "X", 'bigger']
       ]
       [
           ["Set", "X", 'smaller']
       ]
   ]
   ```

   - An if statement without an else part;

   ```
   ["If", [">", 1, 0],
       [
           ["Get", "callFunc", []]
       ]
   ]
   ```

5. A `"Print"` statement starts with a left bracket, and ends with a right bracket. Between these brackets we first have the keyword `"Print"`, followed by a comma, followed by a left bracket, followed by an expression, and followed by a right bracket. Some example `"Print"` statements are:

   - `["Print", [1]]`
   - `["Print", ['Hello World!']]`
   - `["Print", [["Get", "X"]]]`

6. Increment statements and decrement statements are very similar in their syntax. First of all, they both start with a left bracket and end with a right bracket, similar to the other statements we considered so far. After the left bracket, an increment statement has the increment operator `"++"`, whereas a decrement statement has

the decrement operator `"--"`. After these operators, the syntax of increment and decrement statements are the same. The operator is followed by a comma, which is followed by a variable name. Some example increment/decrement statements are:

- `["--", "X12"]`
- `["++", "_myVariable"]`

7. A condition is basically a comparison using one of these operators: `"<="`, `">="`, `"=="`, `">"`, `"<"`. A condition starts with a left bracket, followed one of these operators, followed by a comma, followed by two expressions (where there is a comma between the expressions), and finally concluded by a right bracket. For instance;

   - `["==", 1, 0]`
   - `[">", ["Get", "X"], 0]`
   - `["<=", 5, ["Get", "foo", [1, ["Get", "Y"]]]]`

8. An expression is either a number, or a string, or a `"Get"` expression, or a function declaration, or an operator application, or a condition.

9. A `"Get"` expression is used both to refer to the current value of a variable, and to call a function. For both cases, the syntax starts with a left bracket, and ends with a right bracket. Here is what we have between these brackets:

   - When the `"Get"` expression is used to refer to the current value of a variable, we first have a `"Get"` keyword which is followed by comma, and followed by the name of the variable.
   - However, when the `"Get"` expression is used to call a function, we first have `"Get"` keyword which is followed by comma, followed by the name of the function, followed left bracket, followed by a comma–separated list of expressions (which can be an empty list), followed by a right bracket.

   Here are some example `"Get"` expressions:

   - Getting the value of a variable named `X`.

     ```
     ["Get", "X"]
     ```

   - Calling a function named `sum` with two parameters.

     ```
     ["Get", "sum", [1,2]]
     ```

   - Calling a function named `display` with no parameters.

     ```
     ["Get", "display", []]
     ```

10. A function declaration is an expression which starts with a left bracket and ends with a right bracket. Between the brackets we first have the keyword `"Function"`, followed by a comma, followed by a left bracket, followed by a (possibly empty) comma–separated list of parameters, followed by a right bracket. After this, there is a comma, which is followed by a left bracket, which is followed by a (possibly empty) list of statements, which then is followed by a right bracket. A parameter in the parameter list is an identifier surrounded by a pair of double quotes.

    Here are some example function declarations:

    - `[ "Function", [], []]`
    - `[ "Function", [], [ ["Print", ['Hello World']]]]`
    - ```
      [ "Function", ["X", "Y"],
        [
          [ "Print", ['Start of the function.']]
          [ "If", [">", ["Get", "X"], ["Get", "Y"]],
            [
              [ "Print", ['X is greater than Y'] ]
            ]
            [
              [ "Print", ['X is NOT greater than Y'] ]
            ]
          ]
          ["Print", ['End of the function.']]
        ]
      ]
      ```

11. Operator application is an expression that starts with a left bracket and ends with a right bracket. Between these brackets, first an operator is given (which is one the operators `+`, `-`, `*`, `/`), which is followed by a comma, and two expressions (where there is a comma between the expressions).

    Some example operator applications are:

    - `[ "+", 1, 2]`
    - `[ "-", ["Get", "endTime"], ["Get", "startTime"]]`
    - `[ "*", ["Get", "count"], ["Get", "costPerOccurrence", [1,2] ]]`
    - `[ "/", ["Get", "WeeklyBudget"], 7]`

12. A return statement starts with a left bracket and ends with a right bracket. In between these two brackets, we first have the keyword `"Return"`. This keyword is optionally followed by a comma and an expression, if a value is returned.

    Some example return statements are:

    - `[ "Return" ]`
      ```

- `[ "Return", 1 ]`
- `[ "Return", 'found' ]`
- `[ "Return", ["Get", "X"] ]`

3 Terminal Symbols

Although you can implement your own flex scanner, we provide a flex scanner for this homework. The provided flex scanner implements the following tokens.

`tSTRING`: The scanner returns this token when it sees a `string` in the input.

`tNUM`: The scanner returns this token when it sees a `number` in the input.

`tPRINT`: The scanner returns this token when it sees `"Print"` in the input.

`tGET`: The scanner returns this token when it sees `"Get"` in the input.

`tSET`: The scanner returns this token when it sees `"Set"` in the input.

`tFUNCTION`: The scanner returns this token when it sees `"Function"` in the input.

`tRETURN`: The scanner returns this token when it sees `"Return"` in the input.

`tIDENT`: The scanner returns this token when it sees an identifier in the input.

`tEQUALITY`: The scanner returns this token when it sees "`==`" in the input.

`tIF`: The scanner returns this token when it sees `"If"` in the input.

`tGT`: The scanner returns this token when it sees "`>`" in the input.

`tLT`: The scanner returns this token when it sees "`<`" in the input.

`tGEQ`: The scanner returns this token when it sees "`>=`" in the input.

`tLEQ`: The scanner returns this token when it sees "`<=`" in the input.

`tINC`: The scanner returns this token when it sees "`++`" in the input.

`tDEC`: The scanner returns this token when it sees "`--`" in the input.

Besides these tokens, it silently consumes any white space character. Any other character which is not recognized as a lexeme of the tokens is returned to the parser.

4 Output

Your parser must print out `OK` and produce a new line, if the input is grammatically correct. Otherwise, your parser must print out `ERROR` and produce a new line.

In other words, the main part in your parser file must be as follows (and there should be no other part in your parser that produces and output):

```
int main ()
{
   if (yyparse())
   {
      // parse error
      printf("ERROR\n");
      return 1;
   }
   else
   {
      // successful parsing
      printf("OK\n");
      return 0;
   }
}
```

5 How to Submit

Submit your Bison file named as `username-hw2.y`, and flex file named as `username-hw2.flx` where `username` is your sucourse username and **do not zip your files.** We will compile your files by using the following commands:

```
flex username-hw2.flx
bison -d username-hw2.y
gcc -o username-hw2 lex.yy.c username-hw2.tab.c -lfl
```

So, make sure that these three commands are enough to produce the executable parser. If we assume that there is a text file named test17.JISP, we will try out your parser by

using the following command line:

```
username-hw2 < test17.JISP
```

If the file `test17` includes a grammatically correct JISP program then your output should be `OK` otherwise, your output should be `ERROR`.

6 Notes

- **Important**: Name your files as you are told and **don't zip them.** [-10 points otherwise]
- **Important**: Make sure you include the right file in your scanner and make sure you can compile your parser using the commands given in the section 5. If we are not able to compile your code with those commands your **grade will be zero for this homework.**
- **Important**: Since this homework is evaluated automatically make sure your output is exactly as it is supposed to be. (i.e. OK for grammatically correct and ERROR otherwise).
- No homework will be accepted if it is not submitted using SUCourse+.
- You may get help from our TA or from your friends. However, **you must write your bison file by yourself**.
- Start working on the homework immediately.
- No late submission will be accepted.
- No email submission will be accepted.
- LATE SUBMISSION POLICY:
  Late submission is allowed subject to the following conditions:
  - Your homework grade will be decided by multiplying what you get from the test cases by a "submission time factor (STF)".
  - If you submit on time (i.e. before the deadline), your STF is 1. So, you don't lose anything.
  - If you submit late, you will lose 0.01 of your STF for every 5 mins of delay.
  - We will not accept any homework later than 500 mins after the deadline.
  - SUCourse+'s timestamp will be used for STF computation.
  - If you submit multiple times, the last submission time will be used.