CS305 – Programming Languages
Spring 2020-2021

Homework 1

Due date: March 11, 2021 @ 23:55

Implementing a Lexical Analyzer (Scanner) for JISP

1 Introduction

In this homework you will implement a scanner for ***JISP*** (which is a Lisp like toy programming language). A program written in ***JISP*** starts with a left square bracket and ends with a right square bracket. Also each expression starts with a bracket and ends with a bracket. Expressions are seperated with a comma. An example ***JISP*** program can be seen in Figure 1 on page 7.

Your scanner will be used to produce the tokens in a ***JISP*** program. In a ***JISP*** program, there may be keywords (e.g. `"Get"`, `"If"`, ...), numbers (both integer and real, both negative and positive), identifiers (programmer defined names for variables, etc.), operators and punctuation symbols. Your scanner will catch these language constructs (introduced in Sections 2, 3, 4, 5, 6) and print out the token names together with their positions (explained in Section 7) in the input file. Please see Section 8 for an example of the required output. The following sections provide extensive information on the homework. Please read the entire document carefully before starting your work on the homework.

2 Keywords

Below is the list of keywords that will be implemented. The list corresponds to the complete list of ***JISP*** keywords.

`"Get"`	`"Set"`	`"Function"`	`"Print"`	`"For"`
`"If"`	`"Return"`			

We will assume that the token name for a keyword is formed by appending upper case keyword to t(without quotations). For example, for the keywords `"Set"` and `"If"`, we will use the token names ***tSET*** and ***tIF***, respectively.

JISP is **case–sensitive**. Only the lexemes given above are used for the keywords.

3 Operators & Punctuation Symbols

Below is the list of operators and punctuation symbols to be implemented, together with the corresponding token names.

Lexeme	**Token**	**Lexeme**	**Token**
,	tCOMMA	==	tEQUALITY
+	tPLUS	-	tMINUS
*	tMUL	/	tDIV
++	tINC	−−	tDEC
[	tLBRAC	]	tRBRAC
>	tGT	<	tLT
>=	tGEQ	<=	tLEQ

Some example usage of the operators are given below:

```
["+", 1, 2]
["-", 1, 2]
["/", 1, 2]
["*", 1, 2]
["++", "VariableName"]
```

```
["--", "VariableName"]
[">", 1, 2]
```

As you can see, the expression in ***JISP*** use *prefix* notation (first the operator is given, and then the operands are given), rather than the *infix* notation we are used to in many languages.

4 Identifiers, Numbers, & Strings

You need to implement identifiers and numbers. Here are some rules that you need to pay attention to:

- An identifier consists of any combination of letters, digits and underscore character. However, it cannot start with a digit.
- The token name for an identifier is tIDENT.
- You need to implement positive and negative integers and reals. Any sequence of digits with an optional - at the beginning and with an optional . anywhere in the middle is counted as a number. Note that, the decimal point . cannot appear in the very beginning or at the very end. Examples of possible numbers are given below:

  0
  0.0
  -00.0000
  1
  -1
  01.0
  -1.0
  2.510
  -0.001

- The token name for a number is tNUM.

- Anything inside a single quotation mark is considered as a string. The empty string (an opening quote immediately followed by a closing quote) is also a string.
- The token name for a string is tSTRING.
- To define a new variable, the keyword `"Set"` is used.
- To get the value of an already defined variable the keyword `"Get"` is used.

```
["Set", "x", 123]
["Print" , ["Get", "x"]]
["Set", "stringExample", 'This is a string']
["Set", "name", 'Yusuf']
["Print" ,["+", 'Hello ',  ["Get", "x"]]]
```

- Moreover, for all identifiers, numbers and string you need to store the actual lexemes associated with the tokens. You are required to output the lexemes for the identifiers and the positions of these lexemes. For numbers you will output both the lexemes and the corresponding values along with their positions in your output. Note the lexeme and the value of a number may not be the same. For example a number with the lexeme 0012 has the value 12. See Section 8 for details on the output to be produced.

5 Conditions

In ***JISP*** one can use an `"If"` statement to write a condition. The syntax for an if condition is simple. Just like any other statement an `"If"` statement is also starting with a left square bracket and ends with a right square bracket. Inside these brackets we have a comma separated `"If"` keyword, condition to be checked, expression(s) to be executed if the condition holds, and expression(s) to be executed if the condition doesn't hold. Some examples are given below.

1. If 5 is greater than 0, then 1 otherwise 2.

```
["If", [">",5 ,0] , 1, 2]
```

2. If the value of variable X is greater than the value of variable Y, then print "X is greater than Y", otherwise print "Y is greater than X"

```
["If", [">", ["Get", "X"], ["Get", "Y"]],
       ["Print" , ['X is greater than Y']],
       ["Print" , ['X is not greater than Y']]
]
```

6 Functions

In ***JISP*** one can define a new function. The keyword `"Function"` is used to create a new function. Below code example should give you an idea on how you can define a new function in ***JISP***.

1. A function that doesn't take any parameters, and prints "Hello World".

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

2. A function that takes X and Y as parameters, computes Z=X+Y and returns it.

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

3. The function "printHelloWorld" that was defined before is called.

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

4. The function "SumFunc" that was defined before is called with parameters 1 and 2 and returned value is stored in the variable "OnePlusTwo".

```
["Set", "OnePlusTwo", ["Get", ["PrintFunctions", [1,2]]]]
```

7 Positions

You must keep track of the position information for the tokens. For each token that will be reported, the line number at which the lexeme of the token appears is considered to be the position of the token. Please see Section 8 to see how the position information is reported together with the token names.

8 Input, Output, and Example Execution

Assume that your executable scanner (which is generated by passing your flex program through flex and by passing the generated lex.yy.c through the C compiler gcc) is named as JISPscanner. Then we will test your scanner on a number of input files using the following command line:

```
JISPscanner < test17.JISP
```

As a response, your scanner should print out a separate line for each token it catches in the input file (test17.JISP given in Figure 1). The output format for a token is given below for each token separately:

Token	**Output**
string	⟨*row*⟩⟨*space*⟩`tSTRING`⟨*space*⟩(⟨*lexeme*⟩)
identifier	⟨*row*⟩⟨*space*⟩`tIDENT`⟨*space*⟩(⟨*lexeme*⟩)
number	⟨*row*⟩⟨*space*⟩`tNUM`⟨*space*⟩(⟨*lexeme*⟩)⟨*space*⟩(⟨*value*⟩)
for all other tokens	⟨*row*⟩⟨*space*⟩⟨*token_name*⟩

```
[
    ["Set", "main",
        ["Function", ["date", "birthdate"],
            ["Set", "curYear", 2021],
            ["Set", "curMonth", 'February'],
            ["Print",
                ["-",
                ["Get", "date"], ["Get", "birthdate"]
                ]
            ]
        ]
    ]
]
```

Figure 1: An example ***JISP*** program: test17.JISP

Here, ⟨*row*⟩ gives the location (line number) of the first character of the lexeme of the token and ⟨*token_name*⟩ is the token name for the current item. ⟨*lexeme*⟩ will display the lexeme of the tokens, and ⟨*space*⟩ corresponds to a single space. For example let us assume that the test file test17.JISP given in Figure 1 is processed with your scanner. In this case, the output should be in the following form:

```
1 tLBRAC
2 tLBRAC
2 tSET
2 tCOMMA
2 tIDENT (main)
2 tCOMMA
3 tLBRAC
3 tFUNCTION
```

```
3 tCOMMA
3 tLBRAC
3 tIDENT (date)
3 tCOMMA
3 tIDENT (birthdate)
3 tRBRAC
3 tCOMMA
4 tLBRAC
4 tSET
4 tCOMMA
4 tIDENT (curYear)
4 tCOMMA
4 tNUM (2021) (2021)
4 tRBRAC
4 tCOMMA
5 tLBRAC
5 tSET
5 tCOMMA
5 tIDENT (curMonth)
5 tCOMMA
5 tSTRING (February)
5 tRBRAC
5 tCOMMA
6 tLBRAC
6 tPRINT
6 tCOMMA
7 tLBRAC
7 tMINUS
7 tCOMMA
8 tLBRAC
8 tGET
8 tCOMMA
8 tIDENT (date)
8 tRBRAC
8 tCOMMA
8 tLBRAC
8 tGET
8 tCOMMA
```

```
8 tIDENT (birthdate)
8 tRBRAC
9 tRBRAC
10 tRBRAC
11 tRBRAC
12 tRBRAC
13 tRBRAC
```

Note that, the content of the test files need not be a complete or correct ***JISP*** programs. If the content of a test file is the following:

```
000.0001000
-000.00
'Hello World!'
-005.000 "6thisIsNotAvalidVariableName"
```

Then your scanner should not complain about anything and output the following information:

```
1 tNUM (000.0001000) (0.0001)
2 tNUM (-000.00) (-0.0)
3 tSTRING (Hello World!)
4 tNUM (-005.000) (-5.0)
4 tNUM (6) (6)
```

Note that we have `-005.000 "6thisIsNotAvalidVariableName"` in the input. It looks like a lexeme for an identifier. However, it is not exactly the lexeme of an identifier, since it starts with a digit. In this case, the correct output of the scanner will be to ignore the starting quote, detect 6 as a number, and ignore rest of the characters all the way to the next quote.

9 How to Submit

Submit only your flex file (**without zipping it**) on SUCourse. The name of your flex file must be:

username-hw1.flx

where username is your SuCourse username.

10 Notes

- **Important**: Name your file as you are told and **don't zip it.** **[-10 points otherwise]**
- Do not copy-paste JISP program fragments from this document as your test cases. Copy/paste from PDF can create some unrecognizable charcaters. Instead, all JISP codes fragments that appear in this document are provided as a text file for you to use.
- Make sure you print the token names exactly as it is supposed to be. You will lose points 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 flex file by yourself**.
- Start working on the homework immediately.
- If you develop your code or create your test files on your own computer (not on flow.sabanciuniv.edu), there can be incompatibilities once you transfer them to flow.sabanciuniv.edu. Since the grading will be done automatically on the flow.sabanciuniv.edu, we strongly encourage you to do your development on flow.sabanciuniv.edu, or at least test your code on flow.sabanciuniv.edu before submitting it. If you prefer not to test your implementation on flow.sabanciuniv.edu, this means you accept to take the risks of incompatibility. Even if you may have spent hours on the homework, you can easily get 0 due to such incompatibilities.

- 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.