Sabancı University
Faculty of Engineering and Natural Sciences

CS305 Programming Languages

Homework 3

Due: April 27, 2023 - Thursday @ 23:55

1 Introduction

In this homework, you will implement a tool that includes a simple type checker and a very simple optimizer for the JISP language. Detailed information about this programming language was given in the second homework document. You can check it for more information. The tool will first check if a given JISP program has any syntax error, grammatically. If there are no syntax errors, the tool will perform some type checks for simple cases and simplify some special cases of expressions. Read the rest of the document for more information.

2 Parser and Scanner

The scanner and the parser which you can use to implement this homework are provided to you. The type checks and the simplification of expressions will require you to implement an attribute grammar. Of course, you can write your own versions of the scanner and parser from scratch. However, you can also start from the scanner/parser files provided, modify them as you wish. No matter which approach you take (start from scratch, or modify the given codes), you must make sure that a JISP program must be parsed successfully if and only if it is a syntactically valid JISP program.

3 Constant Expressions

We are familiar with the type rules of programming languages, such as "a string value cannot be divided into a real number", or "a boolean value cannot be added to a real number", etc. Actual implementations of the programming languages perform very complicated checks for this purpose. In this homework, you will write a type checker that will perform a simplified version of such checks, where you will only consider expressions having constants.

One reason we restrict ourselves to constant–only expressions is that the type of expressions can be inferred easily. For example, if you add two constant integers, you get an integer. However, if the expression has a variable in it, in this case, you need to

know the type of the variable, which may not be statically possible for a language like JISP, which is dynamically typed.

Another good thing about constant–only expressions is that, we can compute the value of these expressions statically as well. For example a JISP expression like `["+", 1, ["*", 2, 3]]` is equivalent to `["+", 1, 6]]`, which in turn is equivalent to number `7`.

Let us now explain all this in a formal way.

- We call an expression *a constant expression*:

  1. if it is an integer literal: In this case, the value of the constant expression is the integer literal, and the type of the constant expression is integer. Examples are: `7`, `-3`, `0`, etc.
  2. if it is a real literal: In this case, the value of the constant expression is the real literal, and the type of the constant expression is real. Examples are: `7.1`, `-03.0520`, `0.5`, etc.
  3. if it is a string literal: In this case, the value of the constant expression is the string literal, and the type of the constant expression is string. Examples are: `'Hello World!'`, `'CS305'`, etc.
  4. if it is addition, subtraction, multiplication, or division of two operands which are constant expressions, where these constant expressions have type either integer or real: Note that the operands do not have to be literals. Examples are: `["*", 2, 3]` (in this example, both operands of the multiplication are literals), `["+", 1, ["/", 4, 2.0]]` (in this example the first operand of the addition is a literal, and the second operand is a constant expression which is not a literal), etc.

     – When the type of both operands are integers, then the type of the constant expression is also an integer.
     – When the type of at least one operand is real, the type of the constant expression is real.

     So, type of `["*", 2, 3]` is integer, whereas the type of `["/", 4, 2.0]` and hence the type of `["+", 1, ["/", 4, 2.0]]` are both real.

     In this case, the value of the constant expression is the operator (i.e. addition, subtraction, multiplication, or division) applied to the values of the operands. For example, the value of `["*", 2, 3]` is `6` and the value of `["+", 1, ["/", 4, 2.0]]` is `3.0`. Note that when both operands of the division are integers, we consider an integer division. In integer division, we divide the first operand by the second and round the result down to the nearest whole number. Therefore, the value of `["/", 29, 10]` is `2`.
  5. if it is addition of two operands which are constant expressions of type string: In this case, the type of the constant expression is string and the value of the constant expression is the concatenation of the values of the operands.

Note that the operands do not have to be string literals. Examples are: `["+", 'CS', '305']]`, `["+", 'Hello', ["+", ' ', 'World']]`. The value of the example `["+", 'CS', '305']]` is `'CS305'`, and the value of the example `["+", 'Hello', ["+", ' ', 'World']]` is `'Hello World'`.

6. if it is multiplication of two operands where the first operand is a constant expression of type non-negative integer and the second operand is a constant expression of type string: Note that the operands do not have to be literals. Examples are: `["*", 2, '20']`, `["*", ["+", 1, 1], ["+", '2', '0']]`, etc. For both of these examples, the value is the string `'2020'`.

We would like to emphasize here the fact that the definition of a constant expression is recursive. For example, `6` is a constant expression of type integer (item 1 above), and `7.5` is a constant expression of type real (item 2 above).

Starting from these, we can conclude that `["+", 6, 7.5]` is a constant expression as well (item 4 above). The value of `["+", 6, 7.5]` is `13.5` and the type is real. If we continue in this way, we can also conclude that `["*", 6, ["+", 6, 7.5]]` is also a constant expression (again item 4 above), with value `81.0` and type real. So, the definition works recursively. We can build larger constant expressions from given constant expressions.

Furthermore, you can assume that division by 0 will not be performed. You don't need to check for this condition or produce an error.

4 Type Check

Suppose that there is an expression of the form

```
[OPERATOR, OPERAND1, OPERAND2]
```

where

- `OPERATOR` is one of the operators `"+"`, `"-"`, `"*"`, `"/"`
- `OPERAND1` and `OPERAND2` are expressions

Suppose that `OPERAND1` and `OPERAND2` are constant expressions (according to the possibly recursive use of rules 1–6 given above). However we cannot infer that `[OPERATOR, OPERAND1, OPERAND2]` is a constant expression. In this case, we say that there is a type mismatch error in `[OPERATOR, OPERAND1, OPERAND2]`.

Note that `OPERAND1` and `OPERAND2` don't need to be literals. As long as they are some expressions that can be identified as constant expressions of a certain type, you will need to perform this check.

Whenever your tool detects such a type mismatch error for an expression of the form `[OPERATOR, OPERAND1, OPERAND2]`, it will print the following error.

```
Type mismatch on LINENO
```

where `LINENO` is the line number where the `OPERATOR` is located.

For example, if we have the following program:

```
1   [
2       [ "Print" , 'This is an example JISP program']
3
4       [ "Print" , 'Below are all invalid expressions']
5
6       ["+", 12, '1']
7       ["-", 'string', ["-", 13.5, 2.0] ]
8       ["/", 12.0, 'x']
9       ["*", 'z', ["/", 13.5, 2.0] ]
10      ["+", 'Hello ', 5]
11      ["*", 1.0, ["+", 'Test ', 'String!']]
12  ]
13
```

Then below lines should be printed.

```
Type mismatch on 6
Type mismatch on 7
Type mismatch on 8
Type mismatch on 9
Type mismatch on 10
Type mismatch on 11
```

On line 6, an integer is being added to a string. On line 7, we have subtraction of two real numbers, `["-", 13.5, 2.0]` and there is no problem here. This is a constant expression with value `11.5` of type real (see next section for the values and types of constant expressions). However, we then see a problem for `["-", 'string', ["-", 13.5,2.0] ]`, since this is a subtraction of a string and a real number. On line 8, there is a division of a real number with a string. So on, so forth...

Please note that for nested expressions, you want to print error statements only for the specific expression in which the type mismatch occurs, rather than for all its parent expressions. For example, for the code below:

```
1               [
2                   ["+",
3                       ["+" , 2,
4                           ["+", 'hello', 8]],
5                               ["+", 'world', 5]]
6               ]
7
```

The output would be:

```
Type mismatch on 4
Type mismatch on 5
```

5 Expression Reduction

Let us call an expression as *top–level expression* if it is not a subexpression, i.e. it is not a part of another enclosing expression. For example, when we consider `["+", 1, ["*", 2, 3]]`, in this example the expression `["*", 2, 3]` is not a top–level expression, since it is a part of this enclosing expression.

Your tool will also compute and print the value of top–level constant expressions which are not literals. Suppose that there is a top–level constant expression of the form

```
[OPERATOR, OPERAND1, OPERAND2]
```

In this case, the value of this top–level constant expression will be reported as:

```
Result of expression on LINENO is (RESULT)
```

where `LINENO` is the line number where the `OPERATOR` is located, and `RESULT` is the value the constant expression.

If the value to be printed is a real number, then you have to round the number up to the first decimal point.

For instance, if we have the following program:

```
1     [
2         [ "Print" , 'This is an example JISP program']
3
4         ["Set", "HelloWorld",
5         ["Function", [],
6           [
```

```
            ["Return", ["+", 'Hello ', 'World!']]
          ]
        ]
        ]

        ["+", 12.0, 13]
        ["Set", "X", ["-", 12.1, ["+", 13.5, 2.0] ] ]
        ["/", 12.0, 4]
        ["/", 12, 7]
        ["+", 1, ["/", 12.0, 7] ]
        ["*", 1, ["/", 13.5, 2.0] ]
        ["+", 'Test ', 'String!']
        ["+", ["Get", "X"], ["Get", "X"] ]
        ["*", 2, ["+", 'A', 'B']]
    ]

```

Then the output of this program must be the same as the following lines:

```
Result of expression on 7 is (Hello World!)
Result of expression on 12 is (25.0)
Result of expression on 13 is (-3.4)
Result of expression on 14 is (3.0)
Result of expression on 15 is (1)
Result of expression on 16 is (2.7)
Result of expression on 17 is (6.8)
Result of expression on 18 is (Test String!)
Result of expression on 20 is (ABAB)
```

Note that, although the below expressions are valid expressions in JISP and although there are some constant expressions in these examples, the value of these constant expressions will not be printed out since they are not top–level constant expressions.

```
["+", ["Get", "X"], ["+", 1, 1 ] ]
["/", ["Get", "Y"], ["+", 1, ["Get", "X"] ] ]
["*", 2, ["+", 'a', ["Get", "ID"] ] ]
["/", 2.5, ["+", 'a', ["Get", "ID"] ] ]
```

Also note that, even though the last line is possibly an erroneous statement, your tool will not report any error for this line since `["+", 'a', ["Get", "ID"] ]` is not a constant expression.

6 Example Programs and Ouputs

1. If the program is not grammatically correct then like the second homework you have to print ***ERROR***. For example, if we have the below program:

```
[
    ['Invalid Program']
]
```

   Then the output should be:

```
ERROR
```

2. If a program is grammatically correct then the output should display all the semantic errors and also the result of all of top–level constant expressions.

   Note that you will have to only print the result of constant expressions, and if the expression is valid but is not constant, you don't have to print the result of that expression. So, for example, if we have the following program:

```
[
    ["+", 1, 3]
    ["*", 'B', ["+", 'C', 'A']]
    ["+", 1, ["+", 1, ["Get", "X"] ] ]
    ["++", "X"]
    ["Set", "X", ["*", 1, ["+", 0, 1] ] ]
    ["+", ["Get", "X"], ["+", 1, 2] ]
    ["+", 'A', 'B']
    ["*", 3, 'T']
    ["If", [">",
            ["+", 1, 2],
            ["+", ["Get", "X"], 1]
           ],
           [["+", 2, 3 ]]
    ]
]

```

   Then the output to the above program must be as follows:

```
Result of expression on 2 is (4)
Type mismatch on 3
Result of expression on 6 is (1)
```

```
Result of expression on 8 is (AB)
Result of expression on 9 is (TTT)
Result of expression on 11 is (3)
Result of expression on 14 is (5)
```

- The expression on line 2 is a constant expression thus the result of that expression is printed.
- The expression on line 3 is not a valid expression, so the error message is printed out.
- The expression on line 4 is a valid expression, but it is not a constant expression, so nothing is printed out for that expression.
- The statement on line 5 is an increment statement. And thus, nothing is done for that.
- We have a constant expression inside the set statement on line 6. Thus the result of that expression is printed out.
- The expression on line 7 is a valid expression, but it is not a constant expression, so nothing is printed out for that expression.
- We have a valid constant expression on lines 8, 9, 11, and 14. Thus the results of those expressions are printed out.
- The expression on line 12 is a valid expression, but it is not a constant expression, so nothing is printed out for that expression.

3. You can assume that multiple constant expressions and/or type mismatches don't occur in a single line. Thus, it does not matter in what order you choose to print the output for expressions that appear on the same line. For example, we will not test your program with an input such as:

```
[
    ["+", 1, 3] ["-", 5, 2]
]
```

4. One can have an expression whose value is too large to be stored in the underlying C datatype, causing an overflow. However, our assumption is that we will work with small enough examples that stay within the bounds without causing overflows. For example, you do not need to consider inputs like

```
[
["*", 999999, 'CS305 is the best course ever!!!']
]
```

even though we all know that CS305 is the best course ever...

7 How to Submit

Submit your Bison file named as `username-hw3.y`, and flex file named as `username-hw3.flx` where `username` is your SU-Net username. You may use additional files, such as a header file. Please also upload those files.

We will compile your files by using the following commands:

```
flex username-hw3.flx
bison -d username-hw3.y
gcc -o username-hw3 lex.yy.c username-hw3.tab.c -lfl
```

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

```
username-hw3 < test17.JISP
```

8 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 7. 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. (check sections 4 and 5 for more details).**
- **Important**: 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 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.
- Do not copy-paste JISP program fragments from this document as your test cases. Copy/paste from PDF can create some unrecognizable characters. Instead, all JISP code fragments that appear in this document are provided as a text file for you to use.

- 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 implement the homework by yourself**.
- Start working on the homework immediately.
- 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.