Prof. Jingke Li (lij@pdx.edu), Lec: T 1000-1150; Labs: WR 1400-1550, R 1000-1150; Office Hour: TR 1200-1300; ZID: 87054734177

Homework 4

(Due Wednesday 2/3/21)

This homework continues the work of Lab 4 on compilers and interpreters. It carries a total of 10 points.

1. [3 pts] You are given the task of bootstrapping a compiler for a new programming language L. Assume that you have already implemented a minimal core of the language, L_0, which has the following features:

   - Integer literals, variables, and arithmetic ops, e.g. `1, 25, x, y, +, -, *, /`
   - Assignment of a single-operation expression to a variable, e.g. `x = -5, y = x + 1`
   - Unconditional jump to a label, e.g. `goto L`
   - Conditional jump with a single variable as condition, e.g. `if x goto L` (if `x` $\neq 0$, jump to `L`)
   - Print a single variable or integer, e.g. `print x, print 4`

   Here is a factorial program written in L_0:

   ```
       n = 10
       x = 1
   L1: x = x * n
       n = n - 1
       if n goto L1
       print x
   ```

   The next level of the language, L_1, has the following new features:

   - General arithmetic expressions (parentheses allowed), e.g. `x + (y - z) * 2`
   - General assignment and print statement (arbitrary expr allowed): `x = a + b * c, print(x + y)`
   - General if statement (arbitrary expr allowed as cond): `if (E) S1 else S2`
   - General while statement: `while (E) S` (if `E`'s value $\neq 0$, execute `S` and continue, otherwise break)
   - Compound statement: `{S1; S2; ...}`

   These new features need to be implemented in L_0. As an exercise, show how the following L_1 code fragments can be implemented in L_0 by giving an equivalent code sequence in L_0 for each of them. (*Hint:* You may introduce additional variables if needed.)

   (a) `print(x + y - z * 2)`

   (b) `if (x - y) z = 2 else z = 5; print(z * 3)`

   (c) `while (k) {x = x * k; k = k - 1}; print(x)`

   Save your answers in a file (plain text or pdf) `hw4sol.txt` or `hw4sol.pdf`. Include your name in the file.

2. [7 pts] Consider the following grammar for simple arithmetic expressions (`num` is a token representing an integer):

   ```
   exp    -> term {('+'|'-') term}
   term   -> factor {('*'|'/') factor}
   factor -> '-' factor
          |  '(' exp ')'
          |  num
   ```

Here is a sample expression written in this language: `12 + 2 * (10 - - 4 / 2) + 6`

Note that the symbol `'-'` denotes both the unary negative operator and the binary subtraction operator.

You are to write two parsers and an interpreter for this language.

(a) [3] The first parser only verifies the input, it does not generate output. If the input string is a syntactically correct program, the parser prints `"OK"`; otherwise it raises an exception with an informative error message. The file `arithex0.py` contains a partially implemented parser function, `parse(str)`. Your task is to complete the implementation.

Recall in Lab 3, you worked on a regular expression parser, `regex.py`. This new parser shares the same program structure as the regex parser. However, there are two noticeable differences:

- The regex language is "character-based", meaning that every token is a single character. The arithex language, on the other hand, has an integer token that can have multiple digits.
- The regex expressions are "compact", meaning that there is no whitespace in the input string. The arithex expressions, on the other hand, allow whitespace.

In completing the parser implementation, you need to pay attention to these two issues. Specifically, in the function `next()`, you need to add code to skip over whitespace, so that `next()` always returns the next non-space character. In the function `factor()`, in handling the `num` token, you need to add code to collect all the digits of the integer. (*Hint:* The utility functions `next()` and `advance()` can help in this case.)

Make sure you test your program not only with correct inputs, but also with incorrect ones. Here are some sample test cases, included in the testing section of `arithex0.py`:

```
if __name__ == "__main__":
    parse('12 + (4 * 2 - 5)')
    parse('12 + 2 * (10 - - 4 / 2) + 6')
    # below are error cases; should test each one separately
    parse('x')
    # parse('1=2')
    # parse('1++2')
    # parse('(1+2')
```

(b) [2] After finishing the first parser, copy the program `arithex0.py` to `arithex.py`, and work in the new program file. Modify the parser function `parse(str)` so that it will now return an AST, in the form of a nested list, representing the input program. Here is a sample output:

```
ast = parse('12 + (4 * 2 - 5)')
print(ast) => ['+', 12, ['-', ['*', 4, 2], 5]]
```

(*Hint:* This part is very similar to what you did in the last homework.)

(c) [2] The last part is to write an interpreter function, `eval(ast)`, for interpreting the AST generated by the parser. It returns the value of the expression represented by the AST. Since the AST for this language is very simple, this `eval(ast)` function is also easy to implement. The only place that you need to pay special attention is distinguishing between unary `'-'` and binary `'-'`. Include this function in the second program file, `arithex.py`, so that you can test it directly with the output from the parser function. (*Hint:* Very similar to the `eval()` function in this week's lab.)

Submission

Zip your `hw4sol.[txt|pdf]` file and the two program files `arithex0.py` and `arithex.py` into a single zip file and submit through the "Dropbox" on the D2L class website. *Important:* Keep a copy of your submission file in your folder, and do not touch it. In case there is a glitch in D2L's submission system, the time-stamp on this file will serve as a proof of your original submission time.