CSE 302: Compilers — Lab 1

2020-09-10

1 INTRODUCTION

This lab is primarily intended to get you set up with your development libraries and associated tools. The goal of this lab will be to write a simple code interpreter for straight line code, i.e., code that contains no control structures. The source language, `BX0`, is explained in the next section. We provide a skeleton implementation (in Python) that impelements a tiny subset of the functionality of the interpreter

This lab will be assessed. It is worth 5% of your final grade. Your interpreter is due in 1 week, i.e., on or before `2020-09-17 23:59:59` (Paris time). Each student must work individually for lab 1.

Before you begin

1. Set up your version control system. We recommend using `git` and private repositories on Github. Give read-only access to `@chaudhuri` before the due date.
2. Quickly scan section 4 to find out the requirements for running your interpreter. We will be grumpy if you don't follow these requirements carefully, since we expect to do lots of automated testing as part of the grading process.

2 SOURCE LANGUAGE: `BX0`

The source language, `BX0`, is a purely calculational language. There are no control structures such as loops, conditionals, or functions, nor is there any way to produce any output except by means of the `print` statement. This language works with data of a single type, signed 64 bit integers in 2's complement representation. This can represent all integers in the range $[-2^{63}, 2^{63})$.

`BX0` source consists of the tokens listed in figure 1, and its grammar is shown in figure 2. A ⟨program⟩ consists of a sequence of ⟨statement⟩s, each terminated with a semi-colon. There are two kinds of statements: computation statements (called ⟨assign⟩s) and ⟨print⟩ statements. An ⟨assign⟩ statement computes the value of an ⟨expr⟩ (rhs) and stores the result in a variable (lhs). A ⟨print⟩ statement can be used to print out the value of a single ⟨expr⟩ on a line by itself on the standard output.

An ⟨expr⟩ is made up of `NUMBER`s and variables (`IDENT`s) combined with the usual arithmetic operators (`+`, `-`, `*`, `/`, `%`) and bitwise operators (`&`, `|`, `^`, `~`, `<<`, `>>`). The bitwise operators work on the underlying bit pattern of the word, so two's complement identities such as `x + ~x == -1` will be true. Note that the right shift operator, `>>`, stands for *arithmetic* right shift in 2's complement arithmetic, meaning that the new bits to the left are copies of the sign bit, not `0`s. `BX0` (currently) does not have any Booleans, and therefore does not support (in)equations or Boolean operators. These will be added in the future when the language gets support for conditionals.

The table in figure 3 lists the operators, their arity, and their precedence values. Note that higher precedence binds tighter, so `1 + 2 * 3` would implicitly stand for `1 + (2 * 3)` and not `(1 + 2) * 3`.

```
IDENT  ::= [A-Za-z_][A-Za-z0-9_]*                          (excepting keywords)
NUMBER ::= [0-9]+                                          (numerical value must be ∈ [0, 2^63).)
PRINT  ::= 'print'
PLUS ::= '+'     MINUS ::= '-'     STAR ::= '*'     SLASH ::= '/'     PERCENT ::= '%'
AMP  ::= '&'     BAR   ::= '|'     CARET ::= '^'    TILDE ::= '~'     LTLT    ::= '<<'
GTGT ::= '>>'    LPAREN ::= '('    RPAREN ::= ')'   EQ    ::= '='     SEMICOLON ::= ';'
```

Figure 1: Tokens of the BX0 language. Each token is defined as a literal string of characters (delimited by single quotes) or in terms of the regular expression that matches that token (with restrictions or side conditions written in parentheses).

⟨program⟩ ::= ϵ | ⟨stmt⟩ ⟨program⟩

⟨stmt⟩ ::= ⟨assign⟩ | ⟨print⟩
⟨assign⟩ ::= IDENT EQ ⟨expr⟩ SEMICOLON
⟨print⟩ ::= PRINT LPAREN ⟨expr⟩ RPAREN SEMICOLON

⟨expr⟩ ::= IDENT | NUMBER
| ⟨expr⟩ PLUS ⟨expr⟩ | ⟨expr⟩ MINUS ⟨expr⟩
| ⟨expr⟩ STAR ⟨expr⟩ | ⟨expr⟩ SLASH ⟨expr⟩ | ⟨expr⟩ PERCENT ⟨expr⟩
| ⟨expr⟩ AMP ⟨expr⟩ | ⟨expr⟩ BAR ⟨expr⟩ | ⟨expr⟩ CARET ⟨expr⟩
| ⟨expr⟩ LTLT ⟨expr⟩ | ⟨expr⟩ GTGT ⟨expr⟩
| MINUS ⟨expr⟩ | TILDE ⟨expr⟩
| LPAREN ⟨expr⟩ RPAREN

Figure 2: The grammar of the BX0 language in BNF.

operator	description	arity	precedence
|	bitwise or	binary	10
^	bitwise xor	binary	20
&	bitwise and	binary	30
<<, >>	bitwise shifts	binary	40
+, -	addition, subtraction	binary	50
*, /, %	multiplication, division, modulus	binary	60
-	integer negation	unary	70
~	bitwise complement	unary	80

Figure 3: BX0 operator arities and precedence values. A higher precedence value binds tighter.

Here are two example BX0 programs.

```
x = 10;
print(x);
y = 2 * x;
print(x + y);
z = y / 2;
print(z + z * y);
w = z - x - y;
print(w);
print(- w - w);
print(- w - - w);
print(- - w);
print(w * - w);
print(x + y * 2);
print((x + y) * 2);
```

Simple tests

```
x = 0;
y = 0;
print(x);
z = x;
x = y;
y = y + z;
print(x);
z = x;
x = y;
y = y + z;
print(x);
z = x;
x = y;
y = y + z;
print(x);
```

Fibonacci

These two programs do not cover every kind of construct in BX0. Write your own tests as well.

3 WHAT YOU ARE GIVEN

You have been given the start of an implementation of the lexer and parser for BX0, using the code that you saw during lecture 1. Here are the relevant files:

file	purpose
`ply/`	The PLY library
`scanner.py`	Lexical scanner built using `ply.lex`
`parser.py`	LR parser built using `ply.yacc`
`bx0_interpreter.py`	Dummy interpreter
`README.md`	A text file where you can include any relevant notes about your interpreter

PLY documentation: `https://ply.readthedocs.io/en/latest/ply.html`

4 DELIVERABLES

Your tasks are as follows:

1. Extend the lexical scanner to handle all the tokens of BX0.
2. Extend the parser to handle all the ⟨expr⟩ forms.
3. Extend the parser with the grammar of ⟨stmt⟩s, and then change the start symbol to return a sequence of ⟨stmt⟩s (i.e., a ⟨program⟩).
4. Modify the interpreter to execute the program returned by the parser.

SOFTWARE VERSION The sample code has been tested to work with Python version 3.6+. You may choose to use a higher version of Python, such as 3.8.5, if you want.

If you write an interpreter from scratch in a different programming language, make sure to include a `Makefile`. Also indicate in the `README.md` file precisely what tools and libraries are required to build your interpreter. Your interpreter must be able to run in a recent Linux distribution (think: Debian 8.0+).

Running the interpreter The interpreter will be run on a single `BX0` source file given in the command line. For example, for the source file `example.bx`:

```
$ ./bx0_interpreter.py example.bx
```

Running this command should display all the numbers corresponding to the arguments of the `print()` statements, one by one. If you have other diagnostic messages in your interpreter, set off the `print()` output in some way, such as prefixing the output with a string such as `">>>"`.

Runtime errors The only runtime error that can happen in your `BX0` programs is if a variable is read from before it is written to, i.e., a read from an uninitialized variable. When this happens, print out an error message pointing out the offending variable and possibly also the line of the source file where the error occurred.

Testng The `scanner.py` and `parser.py` sample code you are provided also has simple test suites built in. You can run these test suites by running these files individually.

```
$ python3 parser.py
WARNING: Token 'EQ' defined, but not used
WARNING: Token 'PRINT' defined, but not used
WARNING: Token 'SEMICOLON' defined, but not used
WARNING: There are 3 unused tokens
....
----------------------------------------------------------------------
Ran 4 tests in 0.000s

OK
```

As you extend these modules, make sure to also expand the test suites so that the new constructs are also adequately tested. If you want you may choose to promote these test suites into toplevel visible submodules, so that the tests can be run on all the files at once with `python3 -m unittest`.

You are also encouraged to write your own test `BX0` programs and then routinely test your compiler against your programs. Place every test program you write in the directory `test/` at the top level of your code. Each test program, say `test.bx`, should be accompanied with the expected output, `test.expected`.