

# CS313E: Elements of Software Design

## A Language Interpreter

Dr. Bill Young  
 Department of Computer Sciences  
 University of Texas at Austin

Last updated: November 21, 2011 at 06:44

The goals of this slideset are:

- 1 Put together many of the lessons you've learned this semester.
- 2 Give you more experience in reading some fairly complex code.
- 3 Show you an example of structuring a reasonably-sized Python program.

You will not be expected to know the specifics of the code in this slideset, but you are expected to know how to use the constructs used and to be able to understand the individual code fragments.

The complete code is linked from the syllabus webpage. *I'd suggest that you study this code and make sure you understand it even if you couldn't produce it.*

## A Simple Language

**The project:** Construct in Python an interpreter for simple assembly language programs such as:

```
# This simple subroutine takes an argument on the
# stack, squares it, and leaves the result on the
# stack
square:  dup
         mul
         ret

# The test routine.  Read in an integer, square it
# and output the result.
main:   decl n
        in n
        push n
        call square
        pop ans
        out ans
        hlt
```

## The Language

Our language consists of the following statements, though we won't use most of them in our sample programs.

Arguments in brackets are optional:

DECL var [val]	-- declare a new variable, default to 0 for value
OUT var/int	-- output the value of var
IN var	-- print a prompt, wait for input, store in var
PUSH var/int	-- push value onto the stack
POP [var]	-- pop TOS into variable var (or discard value)
TOP var	-- copy TOS into variable var, leave on stack
DUP	-- duplicate TOS and push
DEC	-- decrement the TOS value by 1
INC	-- increment the TOS value by 1
ADD	-- pop two values, add, push result
SUB	-- pop two values, sub top from top2, push result

Continues on the next page.

```

MUL          -- pop two values, multiply, push result
MOD          -- pop two values, compute s2 mod s1, push result
JMP label   -- change pc to point to label
CMP var/int  -- compare TOS to var or value, set flags
JEQ label   -- jump to label if comparison = 0
JNE label   -- jump to label if comparison != 0
JLT label   -- jump to label is comparison < 0
JGT label   -- jump to label is comparison > 0
JGE, label  -- jump to label if comparison >= 0
JLE label   -- jump to label if comparison <= 0
CALL label  -- jump to label, save return address on callstack
RET         -- pop return address from callstack, jump
HLT         -- halt the machine

```

What problems do we have to solve?

- 1 Input the program from an external file
- 2 Parse the program into internal format
- 3 Decide whether the program is syntactically well-formed (legal)
- 4 Decide whether the program is semantically legal
- 5 Execute the program

We'll do steps 4 and 5 together. That is, we'll attempt to execute the program and fail if the program is not semantically legal.

## What Modules Do We Need?

As with any program, there are multiple ways to write this one. We may want the following classes and modules:

- **HelperFunctions**: various useful functions (module, not class).
- **LowInstruction**: an instruction has fields: label, opcode, args; parse an instruction into an internal form; check that the instruction is legal.
- **LowProgram**: a low program object is just a wrapper for a list of **LowInstruction** objects.
- **LowInterpreter**: give an (operational) semantics to **LowInstruction** objects and to **LowProgram** objects by showing their effect on the system state.
- **TestProgram**: the top level driver program.

## Some Useful Helper Functions

The following two slides define some functions useful in our model. By putting these into a file called `HelperFunctions.py`, you can treat this as a module to be imported as needed. Notice that a module need not contain any classes.

```

def nonEmpty(str):
    """Is the string empty? Returns '' or a nonempty string.
    This works in a boolean context."""
    return ( str.strip() )

def isVar(str):
    """Is the string a legal variable name?"""
    # must contain only letters and digits, but the first
    # must be a letter.
    return ( str.isalnum() and str[0].isalpha() )

def isConst(str):
    """Is the string a legal integer constant?"""
    # isdigit requires at least one character, or breaks
    return ( str.isdigit()
            or ( str[1:] and str[0] == "-" and str[1:].isdigit() ) )

```

```
def isVarOrConst(str):
    """Is the string a legal variable name or constant?"""
    return ( isVar( str ) or isConst( str ) )

def isLabel(str):
    """A legal label has same syntax as variable."""
    return isVar( str )

def isCommentLine(line):
    """A comment line is one that is empty or begins
    with (optionally) whitespace and a '#'. """
    line = line.strip()
    return not line or line[0] == "#"
```

Assume the program is stored on the file system in a file called `inputFile`:

```
fileName = "inputFile"
# prints a helpful message:
print ("Reading instructions from ", fileName)

# opens the file for reading, assigns a Python file object
assemblerFile = open(fileName, 'r')

# reads all of the lines from the file, creates a list
# of strings for manipulation by the program
lines = assemblerFile.readlines()
```

That's all there is to it!

## LowInstruction class

A `LowInstruction` object has an opcode and three other (potentially empty) fields: `label`, `arg1`, `arg2`.

```
class LowInstruction:
    """An instruction contains four fields:

    - a label, which may be empty
    - an opcode
    - up to two arguments

    All of these must be strings (or None).
    """

    def __init__(self, opcode, label=None, arg1=None, arg2=None):
        """Create an Instruction object from a line in a file."""
        self._opcode = opcode
        # There may or may not be arguments or a label
        self._label = label
        self._arg1 = arg1
        self._arg2 = arg2

    # Accessors and mutators for these fields...
```

## Parsing Each Instruction

Now we take the strings we've read in from our input file and turn them (parse them) into `LowInstructionObjects`:

```
# Notice that this is a class method, not an instance method.
def parseLowInstruction(inputStr):
    fields = inputStr.split()
    # Is there a label?
    if ( not fields[0].endswith(":") ):
        # if not, fill in a None
        fields.insert(0, None)
    if ( len(fields) == 2 ):
        # if no args, fill in two Nones
        fields = fields + [None, None]
    elif ( len(fields) == 3 ):
        # If there's only one arg
        fields.append( None )
    # If there is a label strip off the trailing ':'
    label = ( fields[0][:-1] if fields[0] else None )
    opcode = fields[1]; arg1 = fields[2]; arg2 = fields[3]
    instructionObject = LowInstruction(opcode, label, arg1, arg2)
    return instructionObject
```

Now that we're able to parse individual instructions, it's easy to parse a list of instructions:

```
def parseLowInstructionList (lst):
    prog = []
    for instr in lst:
        if isCommentLine( instr ):
            print ("Comment line found: ", instr)
            continue
        instObj = LowInstruction.parseLowInstruction( instr )
        prog.append( instObj )
    return prog
```

Now we have parsed our input instruction strings into LowInstruction objects. Let's check that each one is legal. To do that, we need to be able to check each different type of instruction (depending on the opcode).

We write a separate function for each instruction type. For examples:

```
def legalDecl( self ):
    opcode = self.getOpcode()
    arg1 = self.getArg1()
    arg2 = self.getArg2()
    return ( opcode == "decl"
            and isVar( arg1 )
            and ( isConst( arg2 ) if arg2 else True ))

def legalPush( self ):
    opcode = self.getOpcode()
    arg1 = self.getArg1()
    arg2 = self.getArg2()
    return ( opcode == "push"
            and isVarOrConst( arg1 )
            and not arg2)
```

We check an instruction by checking the label and then checking the legality of each instruction type:

```
def legalInstruction( self ):
    # Check if the label is legal
    label = self.getLabel()
    if ( label and not isLabel( label )):
        print ("Bad instruction label: ", label)
        return
    # Depending on the opcode, call the appropriate
    # recognizer predicate.
    opcode = self.getOpcode()

    if ( opcode == "decl" ):
        return self.legalDecl( )
    ...
    elif ( opcode == "push" ):
        return self.legalPush( )
    ...
    else:
        print ("Unrecognized operation: ", opcode)
        return False
```

A LowProgram object is a “wrapper” for a list of instructions:

```
class LowProgram:
    """A program object is simply a list of LowInstruction objects."""

    def __init__(self, code=[]):
        """A program is just a list of LowInstruction objects, but
        is an empty list by default."""
        self._code = code

    def getCode(self):
        return self._code

    def getCodeLen(self):
        return len( self._code )
```

Given a LowProgram object, we need to be able to find the PC corresponding to a label and get an instruction at a particular PC.

```
def findLabel(self, label):
    """Return the program counter into the code that
    corresponds to the instruction with the given label."""
    if (not label): return -1
    pos = 0
    code = self._code
    while pos < len(code):
        if ( code[pos].getLabel() == label ):
            return pos
        pos += 1
    return -1

def getInstruction(self, index):
    """Return the instruction object at that index."""
    if ( index < 0 or index >= len(self._code) ):
        return None
    else:
        return self._code[index]
```

Not only should the instruction list be syntactically legal as defined above, but no label should be multiply-defined.

```
def legalProgramAux( code, labelsFound ):
    for inst in code:
        label = LowInstruction.getLabel( inst )
        if ( label in labelsFound ):
            print ("Label multiply defined: ", label)
            return False
        b = inst.legalInstruction()
        if (not b):
            print ( "Illegal instruction found: ", inst )
            return False
    return True

def legalProgram ( self ):
    print ("Checking legality of the code")
    progOK = LowProgram.legalProgramAux( self.getCode(), [] )
    return progOK
```

## The Interpreter

We're defining an *operational semantics* for our language, meaning that we define what each instruction does to the "system state."

```
class LowInterpreter:
    """The interpreter operates on a state that consists of:
    - a dictionary of variables and their values
    - a run-time stack for execution
    - a call stack
    - a program counter
    - a comparison flag (an integer: pos, neg, or 0)

    The program is separate from the state.
    """

    def __init__(self):
        # variables is a dictionary mapping var names to integer values
        self._variables = {}
        # The stack should contain only integer values
        self._stack = Stack()
        self._callstack = Stack()
        self._pc = 0
        self._flag = 0

    # the accessors and mutators
```

## Querying and Updating the State

In addition to the usual getters and setters on the state, we may want to define some special methods to perform common operations:

```
def incPc(self):
    # increment the PC by 1
    self._pc += 1

def getValue(self, var):
    # Get the value associated with a variable
    return self._variables[var]

def storeValue(self, var, val):
    # Update the value associated with a variable
    self._variables[var] = val
```

Each individual instruction is defined by explaining how it updates the system state:

```
def declOp (self, var, val=0):
    self.incPc()
    self.storeValue( var, int(val) if val else 0 )

def pushOp (self, arg):
    """Push pushes a value onto the runtime stack. It takes
    either a variable name or an integer constant."""
    self.incPc()
    if ( isVar( arg )):
        val = self.getValue(arg)
    elif ( isConst( arg )):
        val = int(arg)
    else:
        print ("Bad argument to PUSH instruction:", arg)
    self._stack.push(val)

def jeqOp (self, labelLoc):
    """Jump to label if flag is zero."""
    if ( self.getFlag() == 0 ):
        self.setPc(labelLoc)
    else:
        self.incPc()
```

The “meaning” of an instruction is the effect its execution has on the system state.

```
def executeInstruction(self, inst, prog):
    """Execute an instruction, in the form of
    an instruction object. The program is needed
    is needed to compute jump targets."""

    opcode = inst.getOpcode()
    arg1 = inst.getArg1()
    arg2 = inst.getArg2()

    if ( opcode == "decl" ):
        self.declOp(arg1, arg2)
    ...
    elif ( opcode == "push" ):
        self.pushOp(arg1)
    ...
    elif ( opcode == "jeq" ):
        self.jeqOp(prog.findLabel(arg1))
    ...
    else:
        print ("Unrecognized operation: ", opcode)
        self.hltOp()
```

## The Interpreter

The interpreter is a function that takes a program, a starting label, and a clock and runs until completion or the clock “times out.”

```
def runI (self, clk, prog, label=None):
    print ("Initial state:")
    self.printState()
    # Start at label, or 0 if None given
    if label:
        labelLoc = prog.findLabel( label )
        if ( labelLoc == -1 ):
            print ("Start label ", label, " not found")
            self._pc = labelLoc
        else:
            self._pc = 0
    stepCnt = 1

# runI code continues on next slide
```

## The Interpreter (Continued)

```
# Terminates when we time out or run off the
# end of the list
while True:
    if ( clk <= 0 ):
        print ("Terminating: timed out")
        self.printState()
        return
    cl = prog.getCodeLen()
    pc = self._pc
    # If the pc ever gets outside the program, break.
    if ( pc < 0 or pc >= cl ):
        break
    inst = prog.getInstruction( pc )
    print (("Step %3d: %s" % (stepCnt, inst)))
    self.executeInstruction ( inst, prog )
    self.printState()
    clk -= 1
    stepCnt += 1
print( "Program terminated with the following values:")
self.printState()
```

This is the top level function that puts everything together.

```
def TestFileInput():
    # These create a list of lines (strings) from the file.
    fileName = "inputFile5"
    print ("Reading instructions from ", fileName)
    assemblerFile = open(fileName, 'r')
    lines = assemblerFile.readlines()

    # parse and print the program
    instList = LowInstruction.parseLowInstructionList( lines )
    program = LowProgram(instList)

    # Check if program is legal
    if program.legalProgram():
        program.printProgram()
        # Now execute the program
        state = LowInterpreter()
        state.runI(50, program, "main")
```

Recall our initial program:

```
# This simple subroutine takes an argument on the
# stack, squares it, and leaves the result on the
# stack
square:  dup
         mul
         ret

# The test routine.  Read in an integer, square it
# and output the result.
main:    decl n
         in n
         push n
         call square
         pop ans
         out ans
         hlt
```

It accepts an input number from the user, squares it, and displays the result.

Calling `TestFileInput()` at the top level produces the input on the following 5 slides:

```
felix:~/cs313e/python/languages> python LanguagesLow.py
Reading instructions from inputFile5
Comment line found: # This simple subroutine takes an argument on the
Comment line found: # stack, squares it, and leaves the result on the
Comment line found: # stack
Comment line found:
Comment line found: # The test routine.  Read in an integer, square it
Comment line found: # and output the result.
```

Checking legality of the code

```
0:  square:  dup
1:          mul
2:          ret
3:  main:    decl n
4:          in n
5:          push n
6:          call square
7:          pop ans
8:          out ans
9:          hlt
```

```
Initial state:
variables: {}
stack:
callstack:
pc: 0
flag: 0
```

```
Step 1: main:  decl n
variables: {'n': 0}
stack:
callstack:
pc: 4
flag: 0
```

```
Step 2:          in n
Waiting for integer input: 67
variables: {'n': 67}
stack:
callstack:
pc: 5
flag: 0
```

```

Step 3:          push n
variables: {'n': 67}
stack:      67
callstack:
pc:         6
flag:       0

Step 4:          call square
variables: {'n': 67}
stack:      67
callstack:  7
pc:         0
flag:       0

Step 5: square: dup
variables: {'n': 67}
stack:      67 67
callstack:  7
pc:         1
flag:       0

```

```

Step 6:          mul
variables: {'n': 67}
stack:      4489
callstack:  7
pc:         2
flag:       0

Step 7:          ret
variables: {'n': 67}
stack:      4489
callstack:
pc:         7
flag:       0

Step 8:          pop ans
variables: {'ans': 4489, 'n': 67}
stack:
callstack:
pc:         8
flag:       0

```

```

Step 9:          out ans
Value returned: 4489
variables: {'ans': 4489, 'n': 67}
stack:
callstack:
pc:         9
flag:       0

Step 10:         hlt
variables: {'ans': 4489, 'n': 67}
stack:
callstack:
pc:        -1
flag:       0

Program terminated with the following values:
variables: {'ans': 4489, 'n': 67}
stack:
callstack:
pc:        -1
flag:       0

```