Intermediate Representation (IR)

- Our simple, syntax directed interpretation scheme that we worked out for the exp1 language, where we computed values for expressions as soon as we recognized them in the input stream, will fail with more complex languages.
- Let's extend exp1 with conditional and unconditional jump instructions and call the language **exp1bytecode**

Exp1bytecode Language Design

- Four new statements:
  - stop ;
  - jumpT exp label ;
  - jumpF exp label ;
  - jump label ;
  - **Note**: exp is an integer expression and is interpreted as false if its value is zero otherwise it is true
- Labeled statements:

```
store x 5;
L1:
   store x (- x 1);
   jumpT x L1;
```

- Two new operators: =, =<, that return 0 when false otherwise they will return 1.
- Lastly, we also allow for negative integer constants:
  - -2, -12

Exp1bytecode Grammar

```
# %load code/exp1bytecode_gram.py
from ply import yacc
from exp1bytecode_lex import tokens, lexer

def p_grammar(_):
    '''
    prog : instr_list

    instr_list : labeled_instr instr_list
               | empty

    labeled_instr : label_def instr

    label_def : NAME ':'
              | empty

    instr : PRINT exp ';'
          | STORE NAME exp ';'
          | JUMPT exp label ';'
          | JUMPF exp label ';'
          | JUMP label ';'
          | STOP ';'
          | NOOP ';'
…
```

```
…
    exp : '+' exp exp
        | '-' exp exp
        | '-' exp
        | '*' exp exp
        | '/' exp exp
        | EQ exp exp
        | LE exp exp
        | '(' exp ')'
        | var
        | NUMBER

    label : NAME
    var : NAME
    '''
    pass

def p_empty(p):
    'empty :'
    pass

def p_error(t):
    print("Syntax error at '%s'" % t.value)

parser = yacc.yacc()
```

Exp1bytecode Lexer

```
# %load code/exp1bytecode_lex.py
# Lexer for Exp1bytecode

from ply import lex

reserved = {
  'store' : 'STORE',
  'print' : 'PRINT',
  'jumpT' : 'JUMPT',
  'jumpF' : 'JUMPF',
  'jump' : 'JUMP',
  'stop' : 'STOP',
  'noop' : 'NOOP'
}

literals = [':',';','+','-','*','/','(',')']

tokens = ['NAME','NUMBER','EQ','LE'] + list(reserved.values())

t_EQ = '='
t_LE = '=<'
t_ignore = ' \t'

…
```

Exp1bytecode Lexer (Con’t)

```
…
def t_NAME(t):
  r'[a-zA-Z_][a-zA-Z_0-9]*'
    t.type = reserved.get(t.value,'NAME')    # Check for reserved words
    return t

def t_NUMBER(t):
  r'[0-9]+'
    t.value = int(t.value)
    return t

def t_NEWLINE(t):
  r'\n'
  pass

def t_COMMENT(t):
  r'\#.*'
  pass

def t_error(t):
    print("Illegal character %s" % t.value[0])
  t.lexer.skip(1)

# build the lexer
lexer = lex.lex()
```

Exp1bytecode

- Here is a simple example program in this language:

```
// this program prints out a
// list of integers
   store x 10 ;
L1:
   print x ;
   store x (- x 1) ;
   jumpT x L1 ;
   stop ;
```

☞ **Problem:** in syntax directed interpretation all info needs to be available at statement execution time; the label definition is not available at jump time.
☞ **Answer:** we will use an IR to do the actual interpretation.

Syntax directed interpretation

```
...
def p_plus_exp(p):
    """
    exp : '+' exp exp
    """
    p[0] = p[2] + p[3]

def p_minus_exp(p):
    """
    exp : '-' exp exp
    """
    p[0] = p[2] - p[3]

def p_paren_exp(p):
    """
    exp : '(' exp ')'
    """
    p[0] = p[2]

def p_var_exp(p):
    "exp : var"
    p[0] = p[1]

def p_num_exp(p):
    "exp : num"
    p[0] = p[1]
...
```

In our simple expression interpreter we saw that all the info was available at expression execution.

Syntax directed interpretation fails…

```
prog : instr_list

instr_list : labeled_instr instr_list
           |empty

labeled_instr : label_def instr

label_def : NAME ':'
          |empty

instr : PRINT exp ';'
      |STORE NAME exp ';'
      | JUMPT exp label ';'
      | JUMPF exp label ';'
      | JUMP label ';'
      | STOP ';'
      | NOOP ';'
…
```

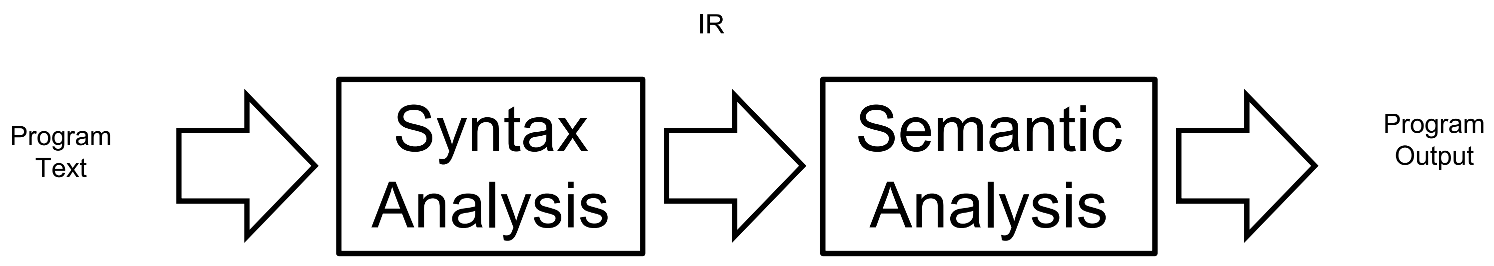
But exp1bytecode we see that label definitions are *non-local* to jump statements and therefore *cannot* be executed in a syntax directed manner.

Even if we were to implement some sort of label table, how do we represent the instructions that we want to jump to?

☞ **Answer:** we will use an IR to do the actual interpretation.

Top-level Design

- Our interpreter will follow the layout for an interpreter very closely

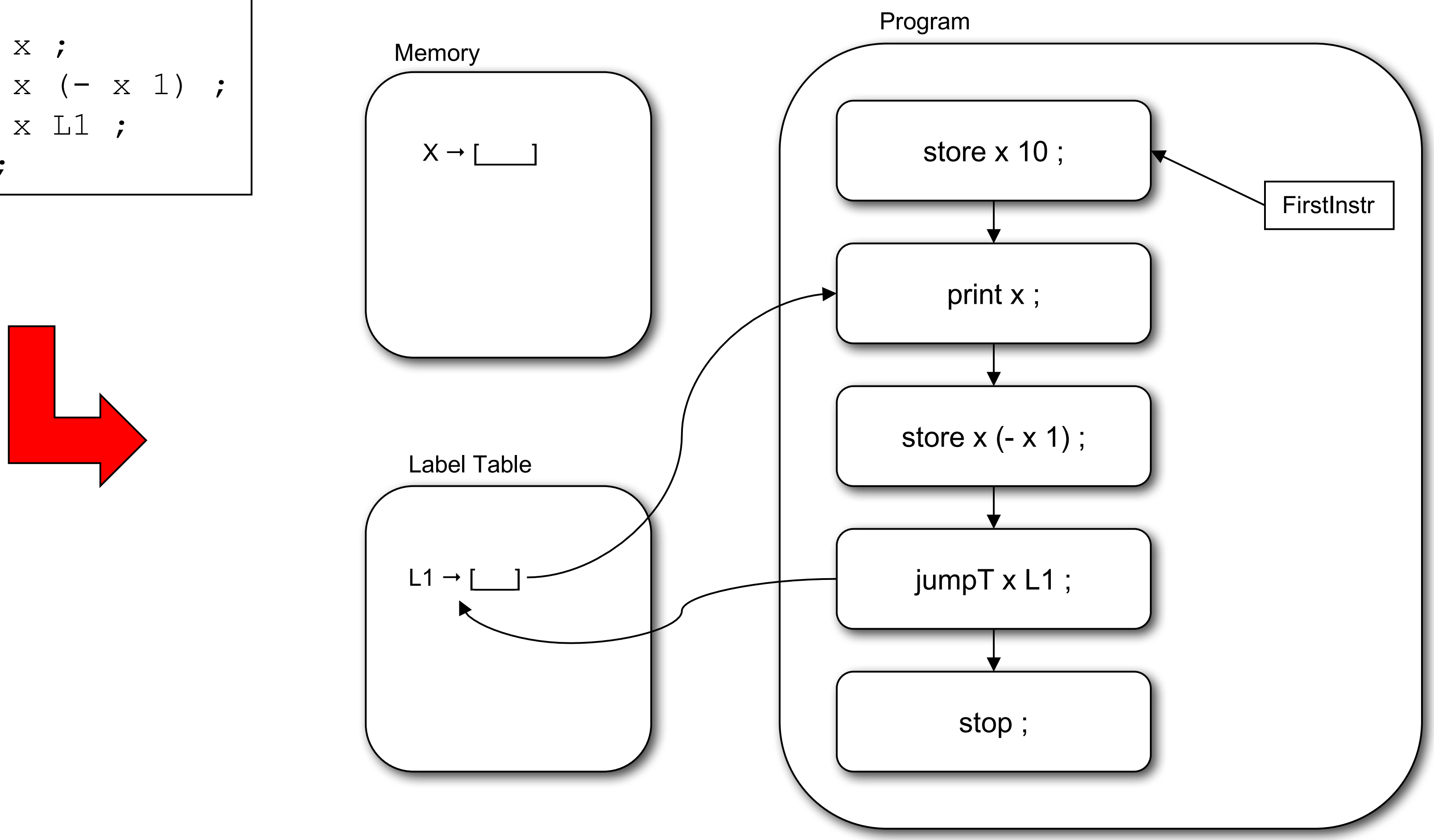


IR Design

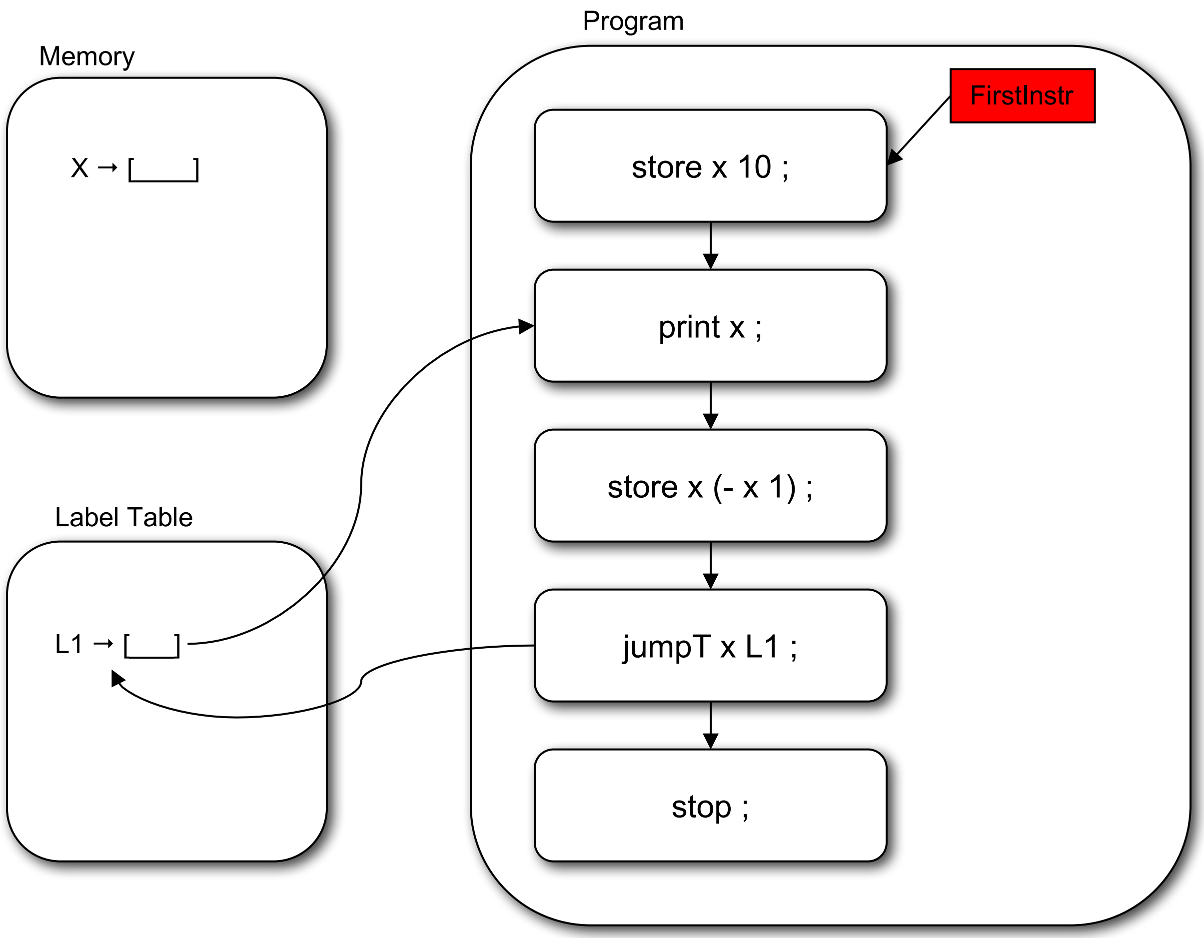
- For variable values we will use the *dictionary based symbol table* from before
- As our IR we will use an abstract representation of the program as a *list of instructions*
- For label definitions we will use a *label lookup* table that associates labels with instructions in our list of instructions

IR Design

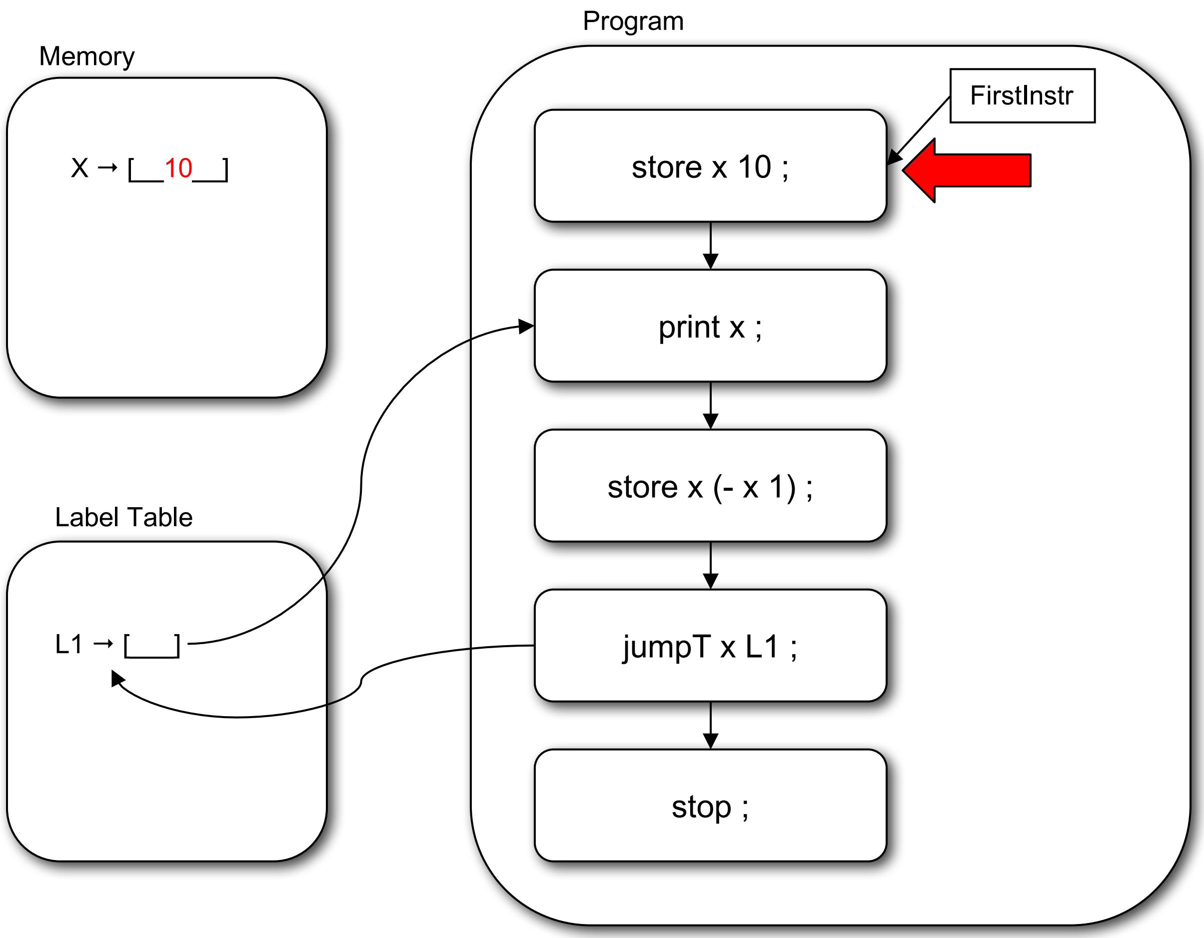
```
   store x 10 ;
L1:
   print x ;
   store x (- x 1) ;
   jumpT x L1 ;
   stop ;
```



Running the Program

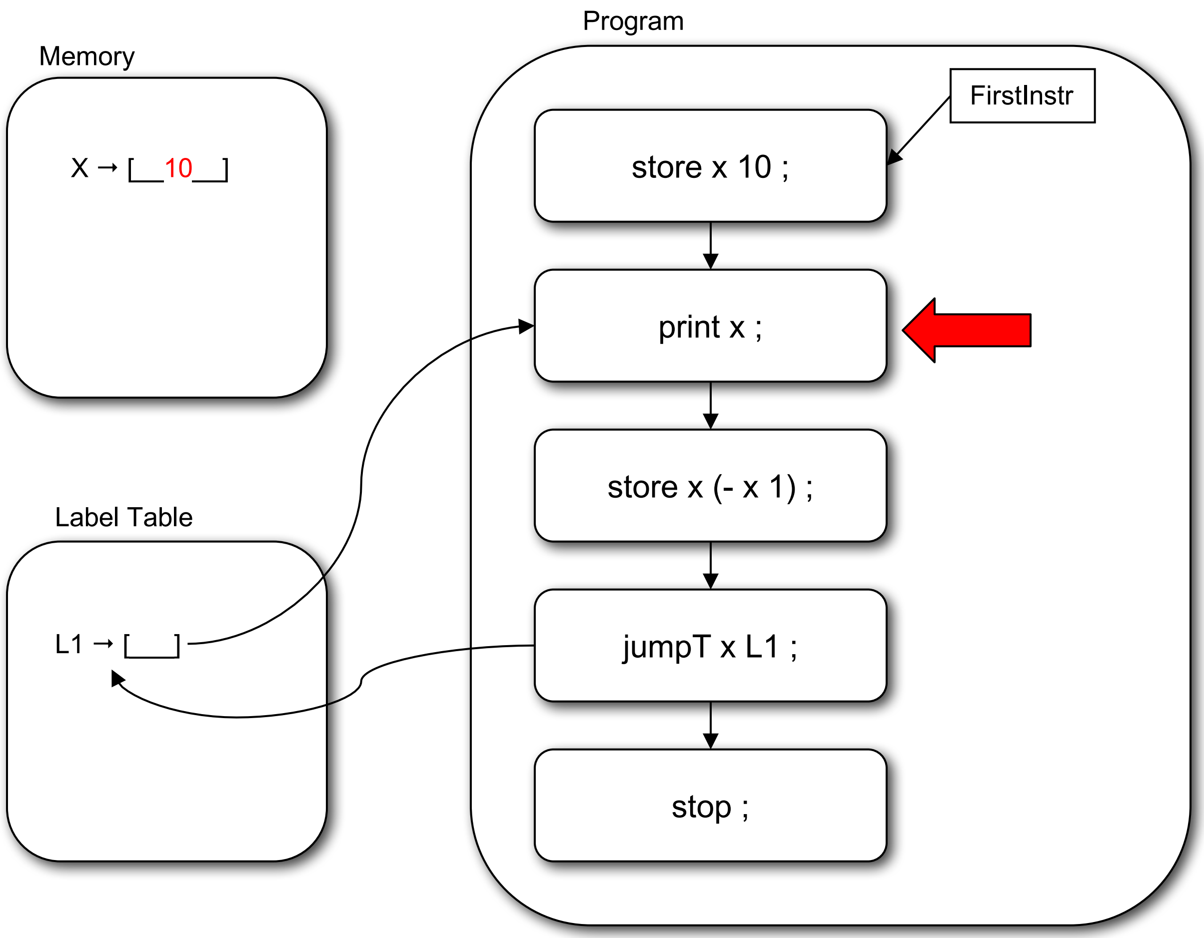


Running the Program



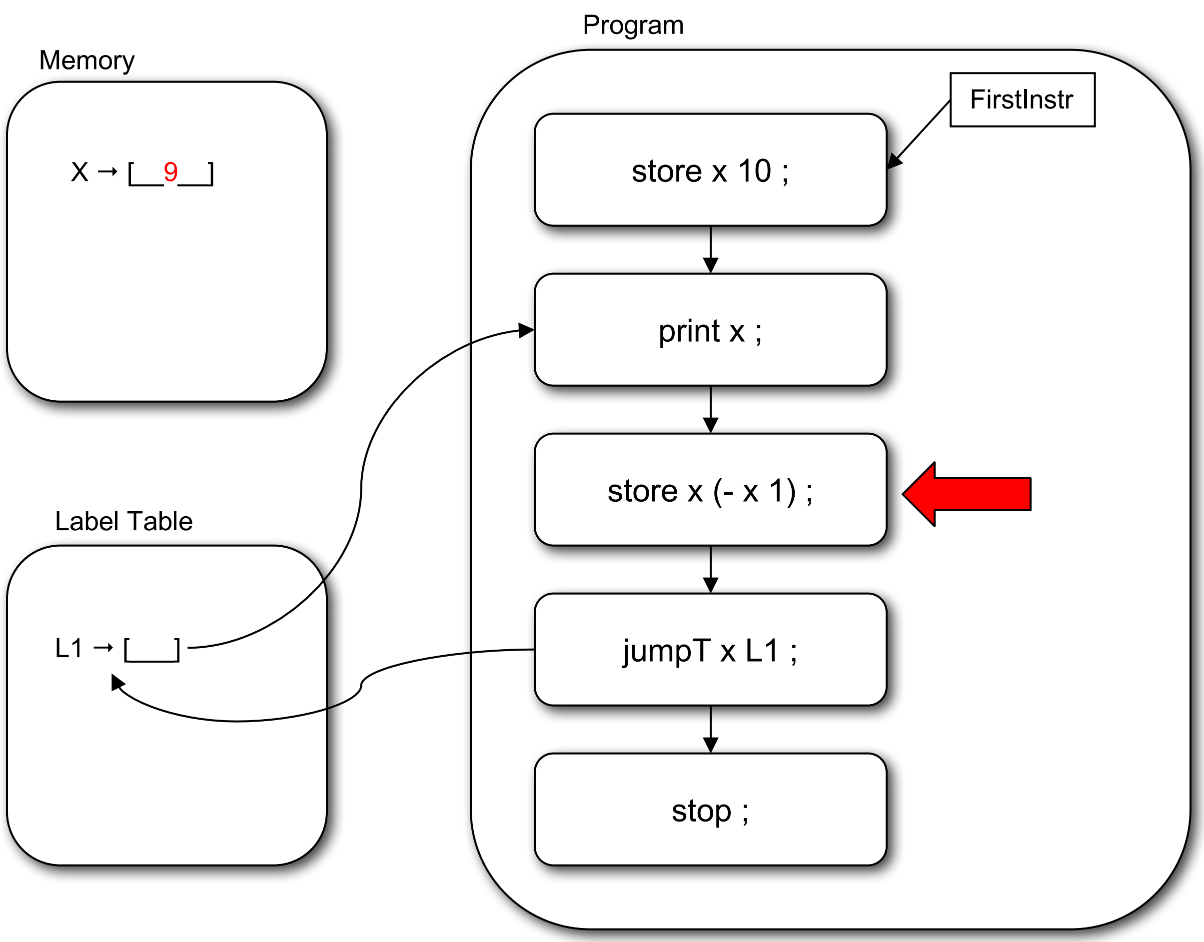
Running the Program

10



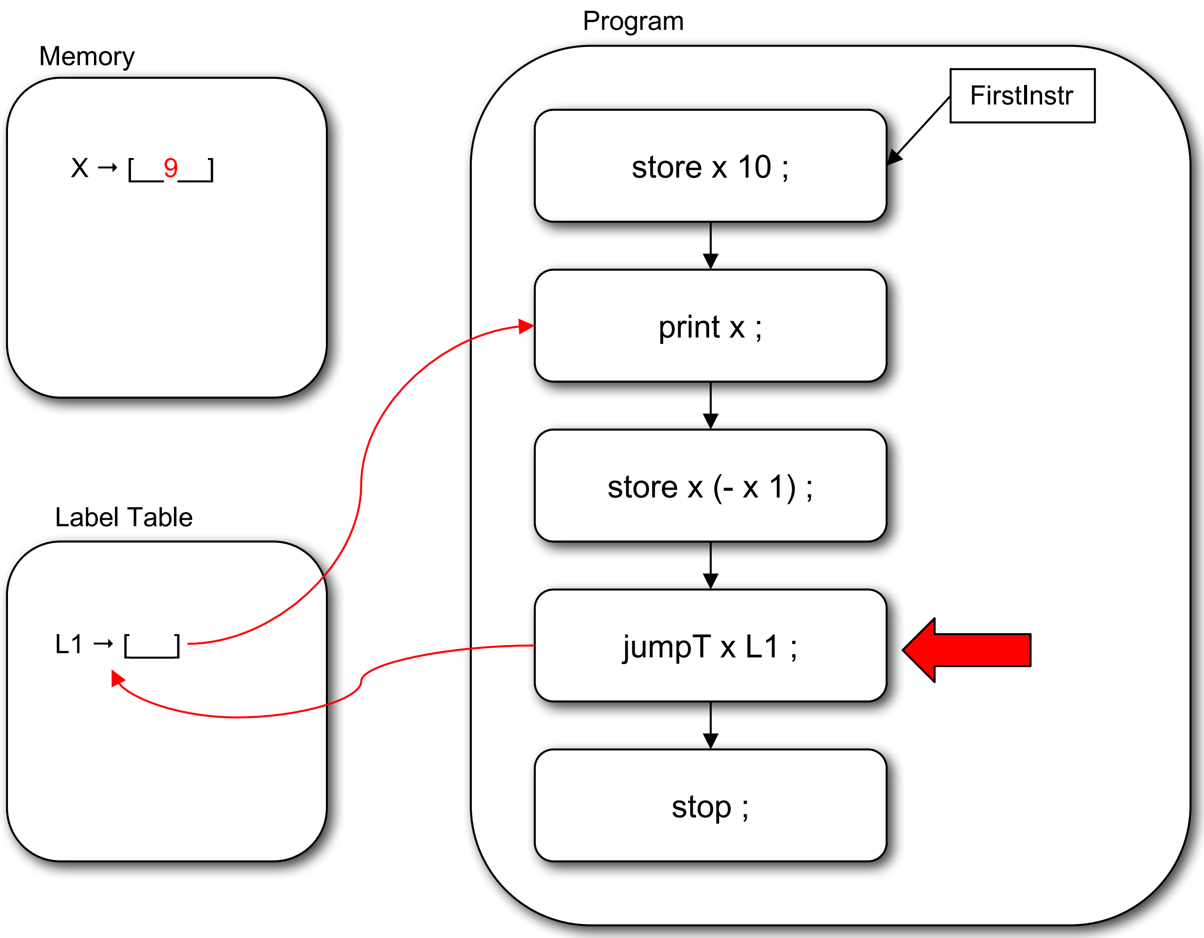
Running the Program

10



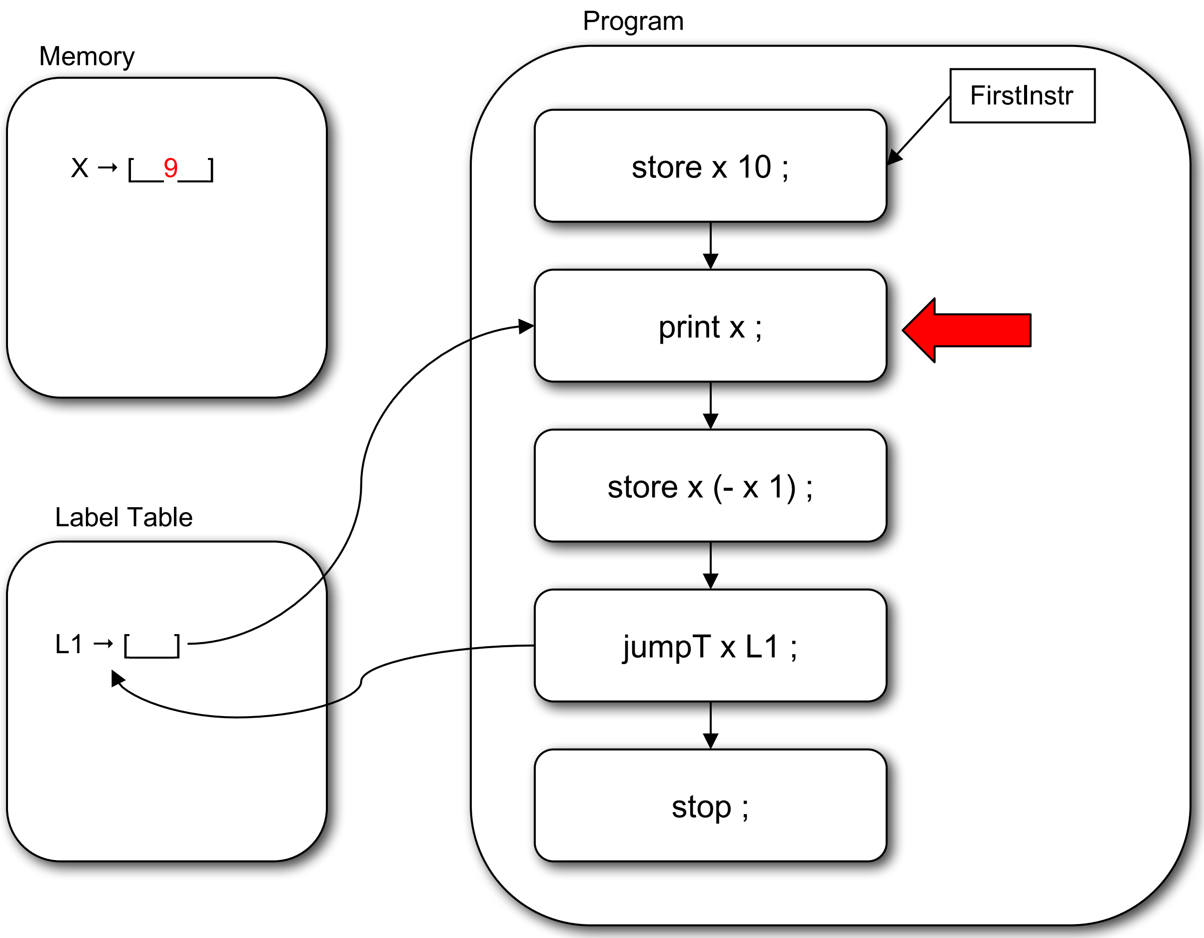
Running the Program

10



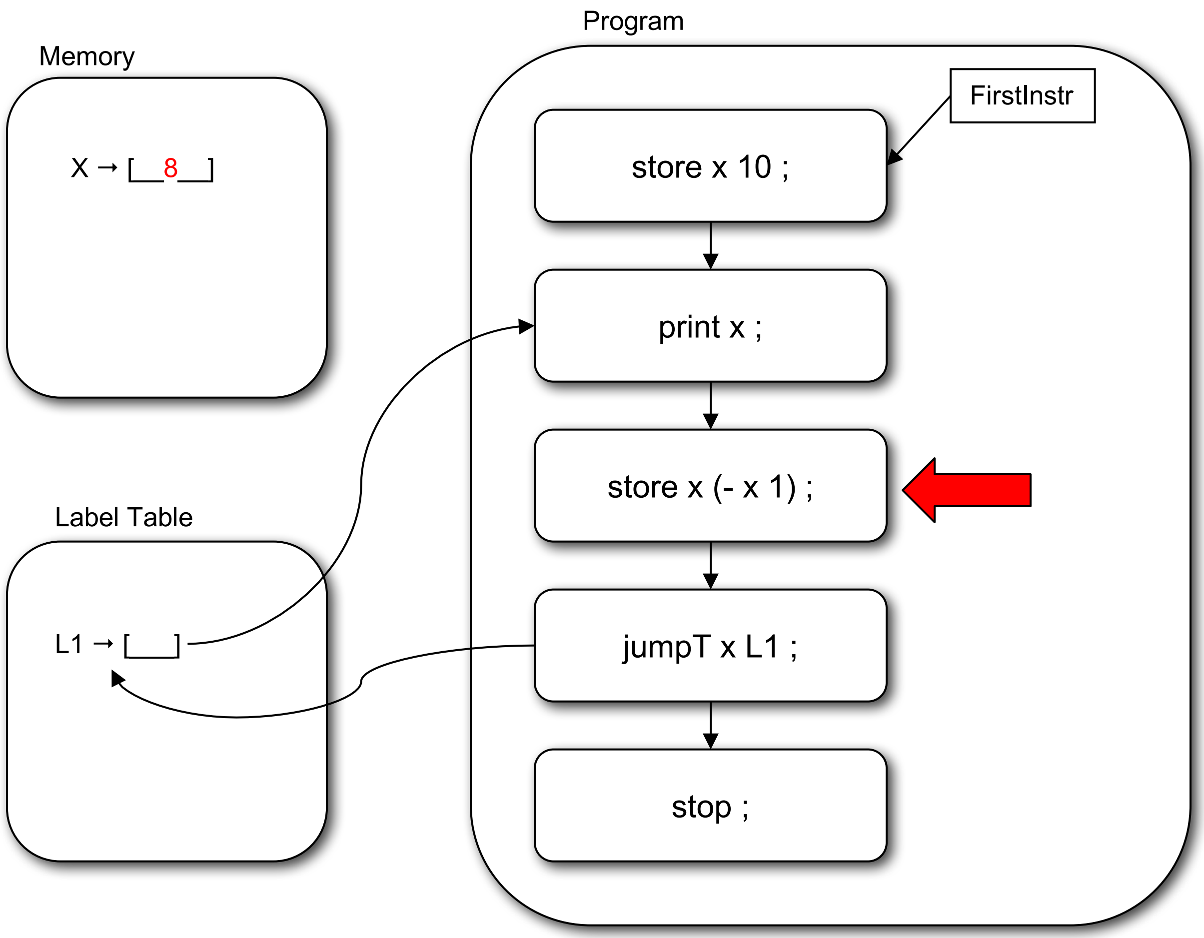
Running the Program

10 9



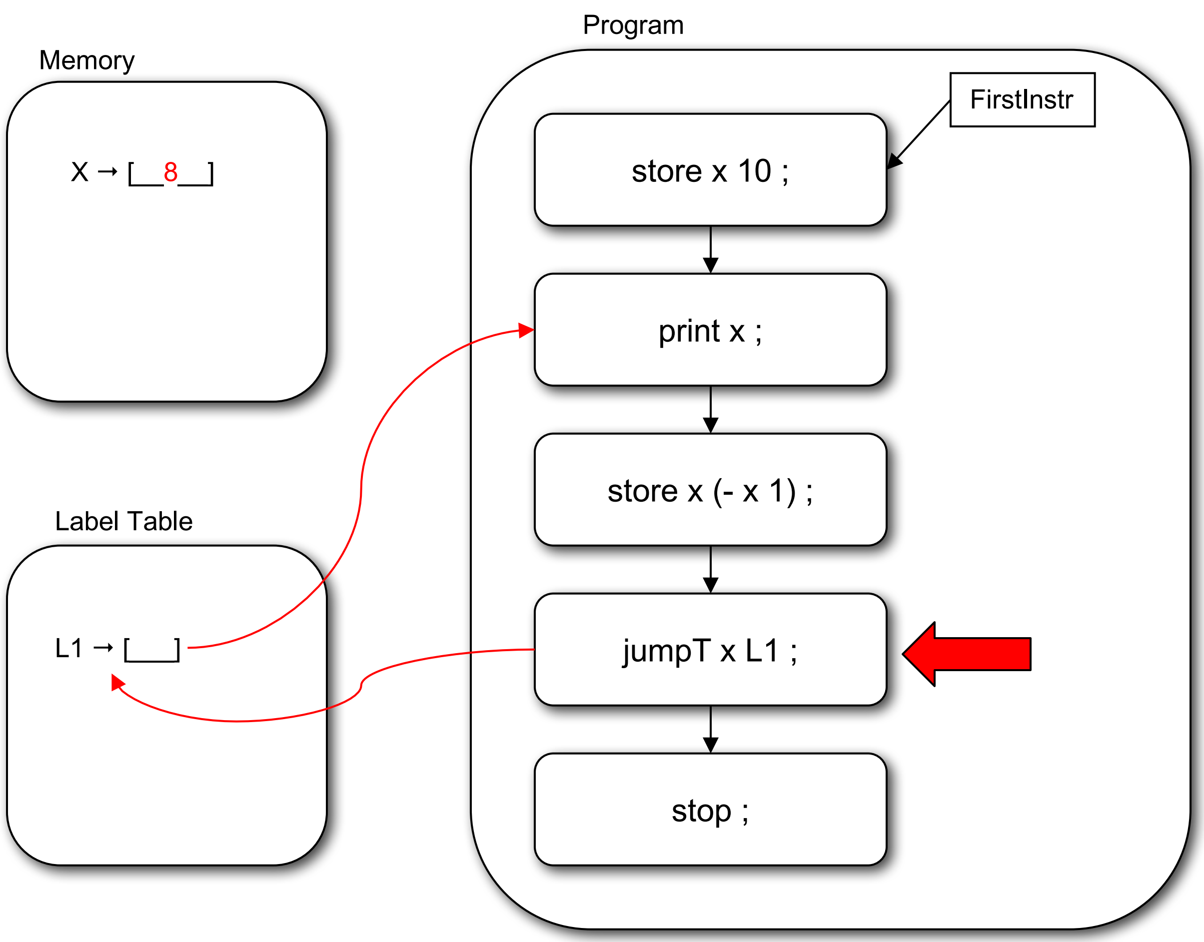
Running the Program

10 9



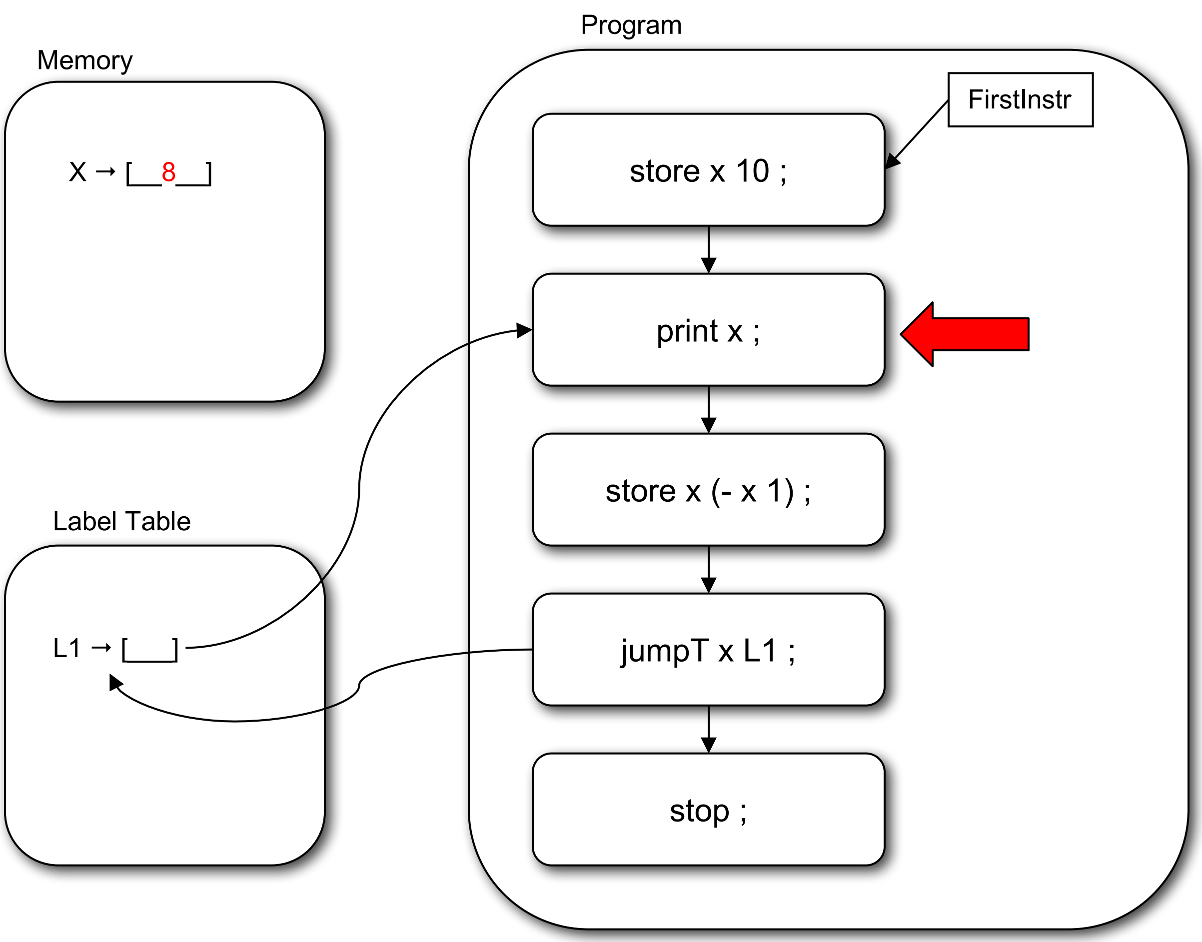
Running the Program

10 9



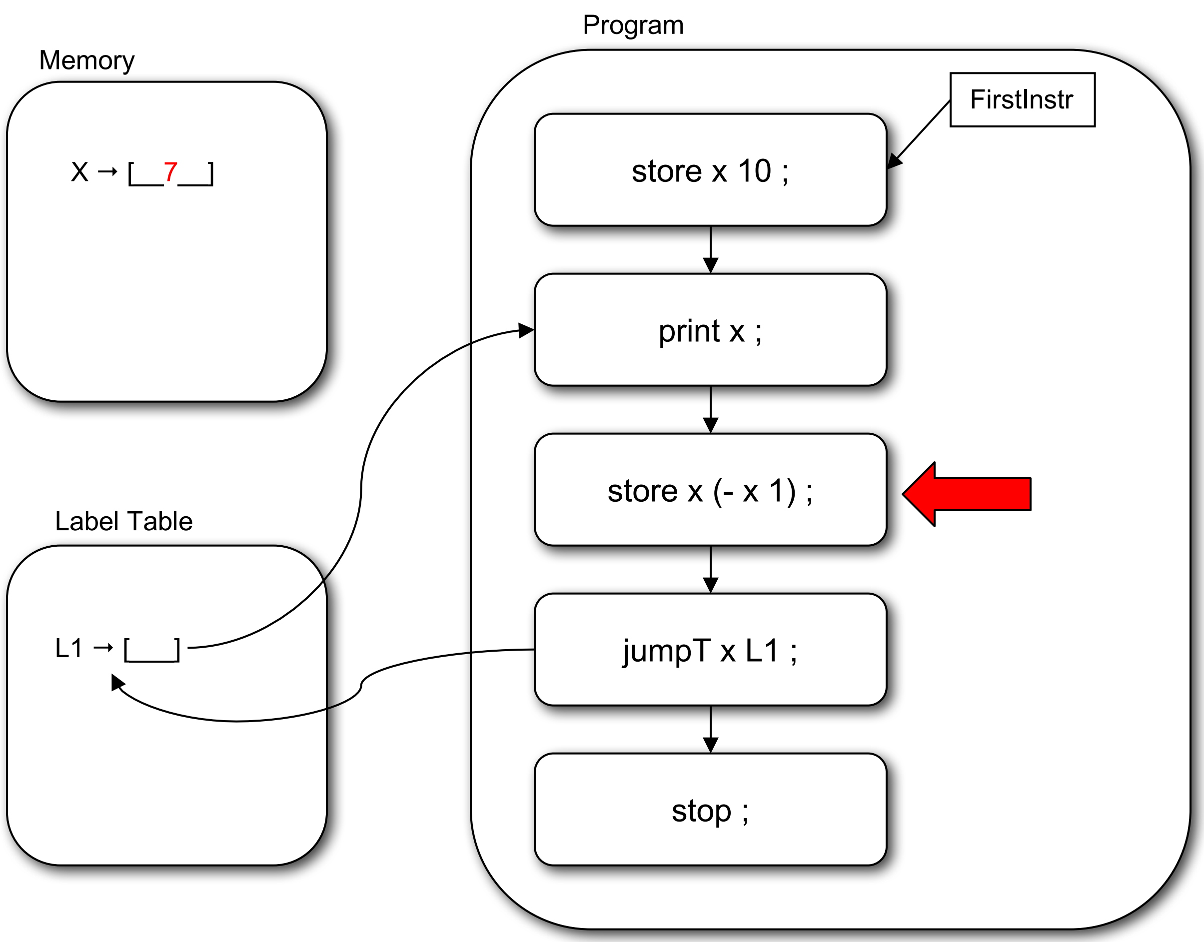
Running the Program

10 9 8



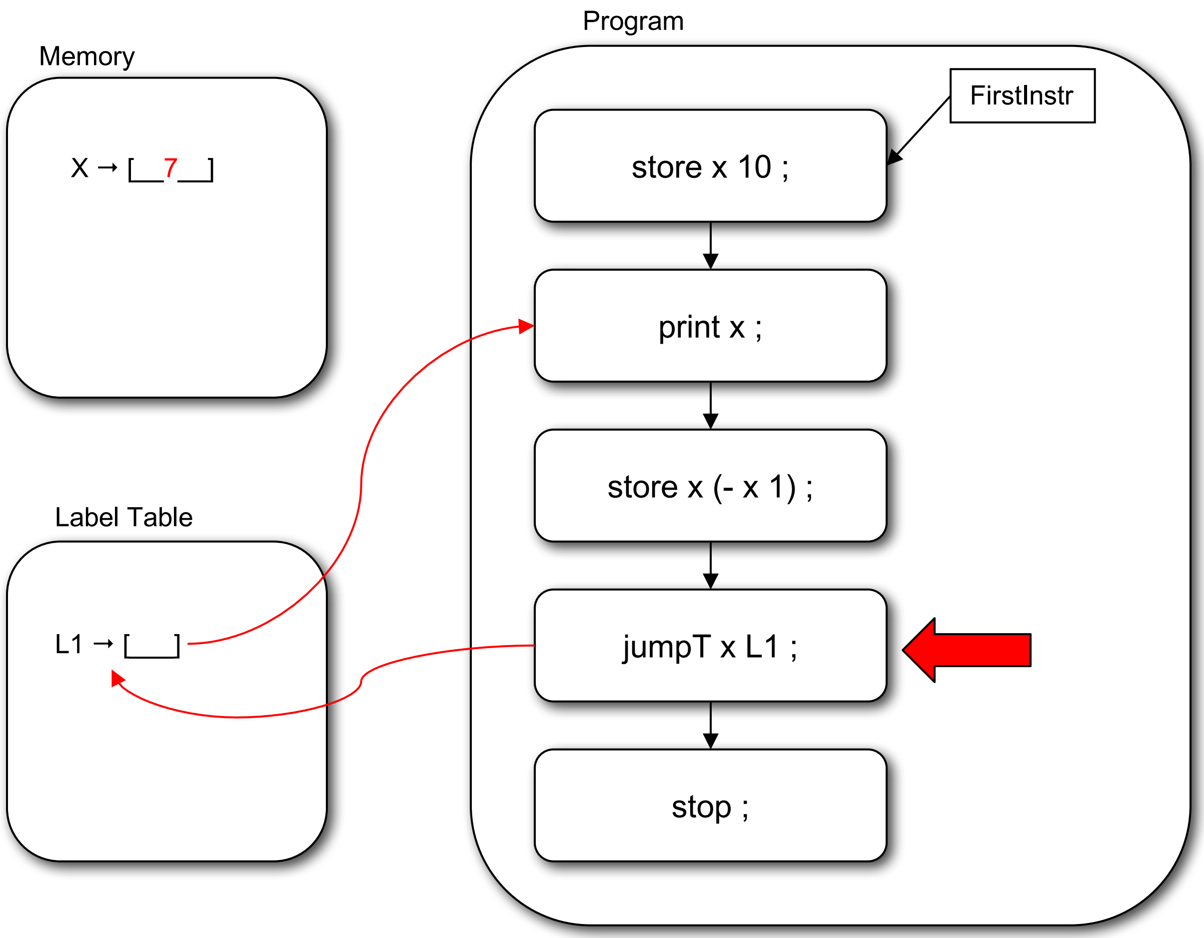
Running the Program

10 9 8



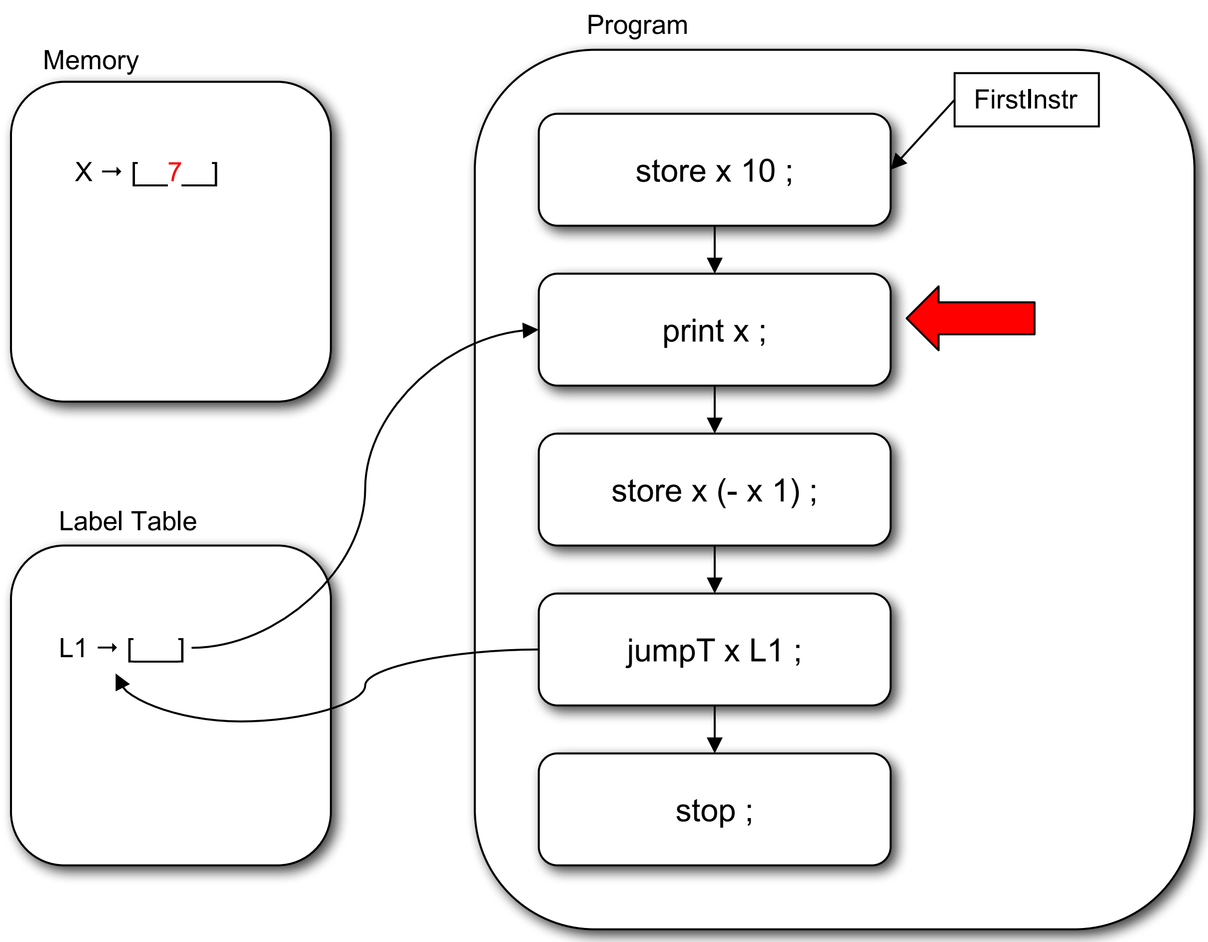
Running the Program

10 9 8



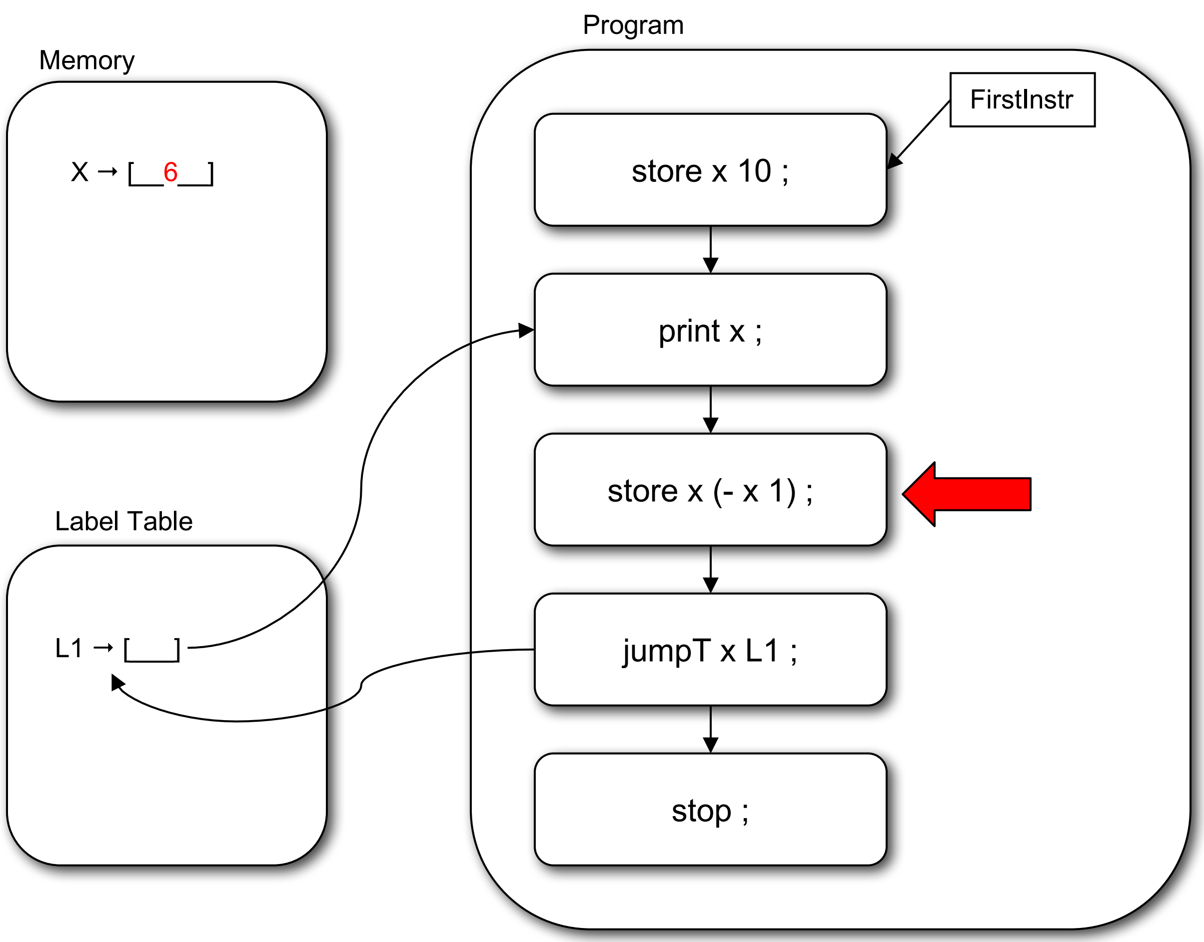
Running the Program

10 9 8



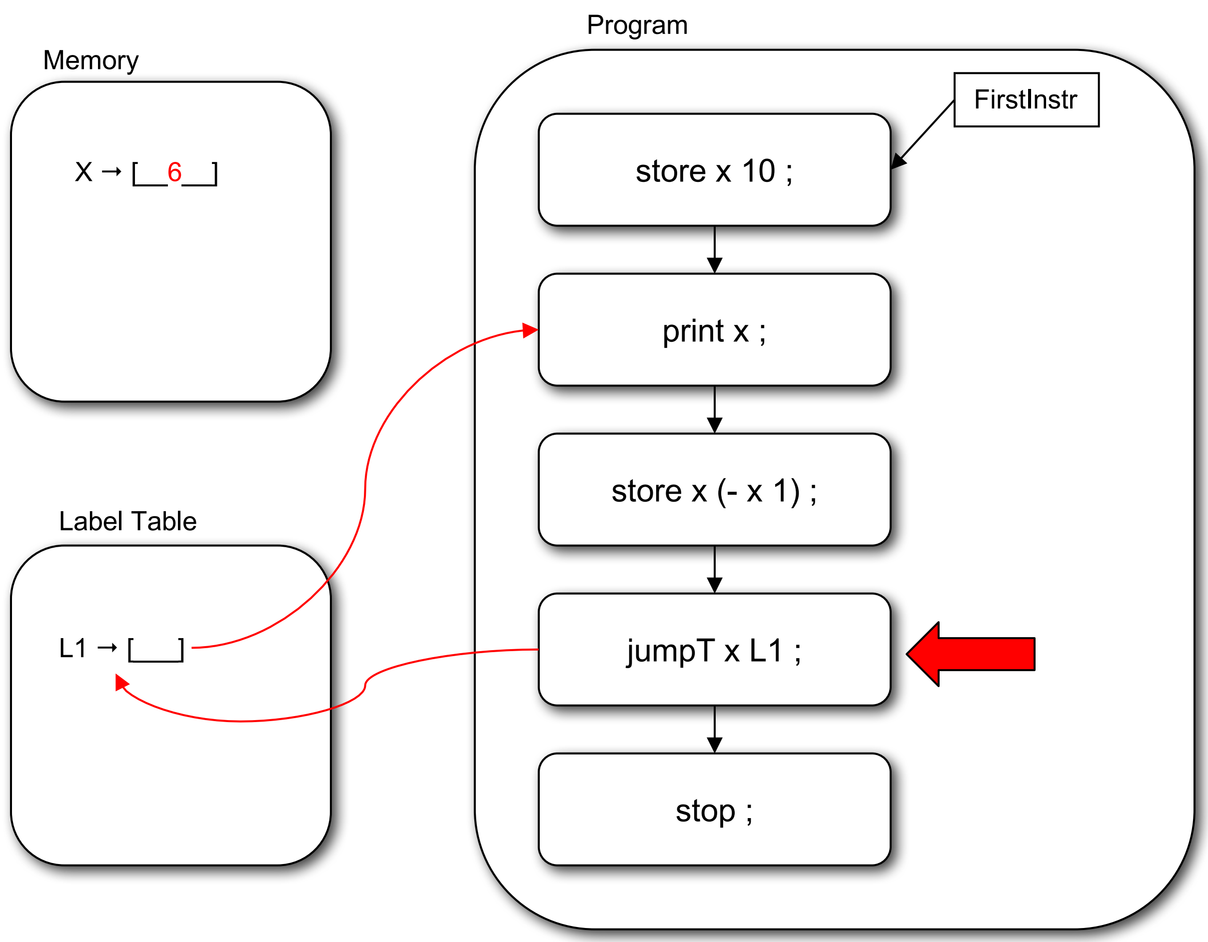
Running the Program

10 9 8 7



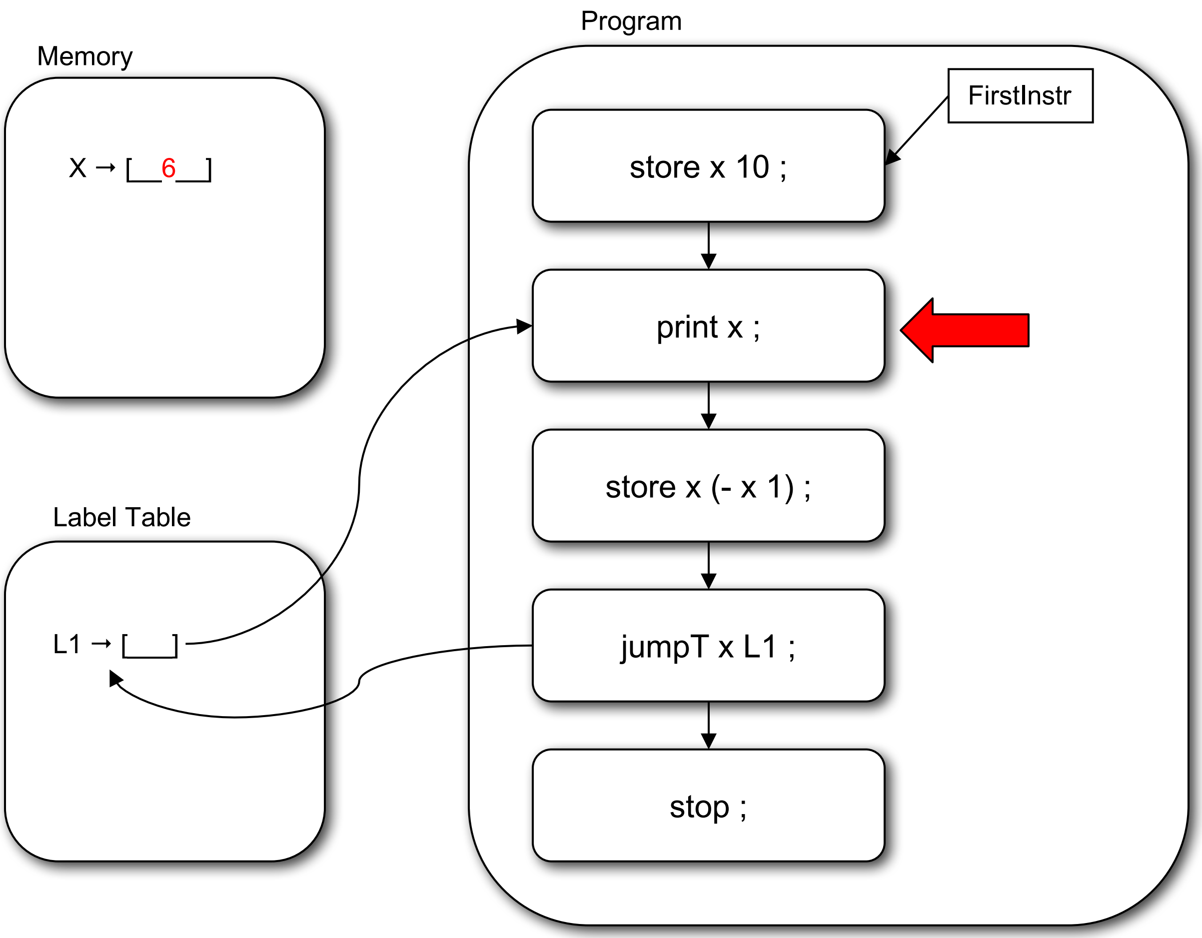
Running the Program

10 9 8 7



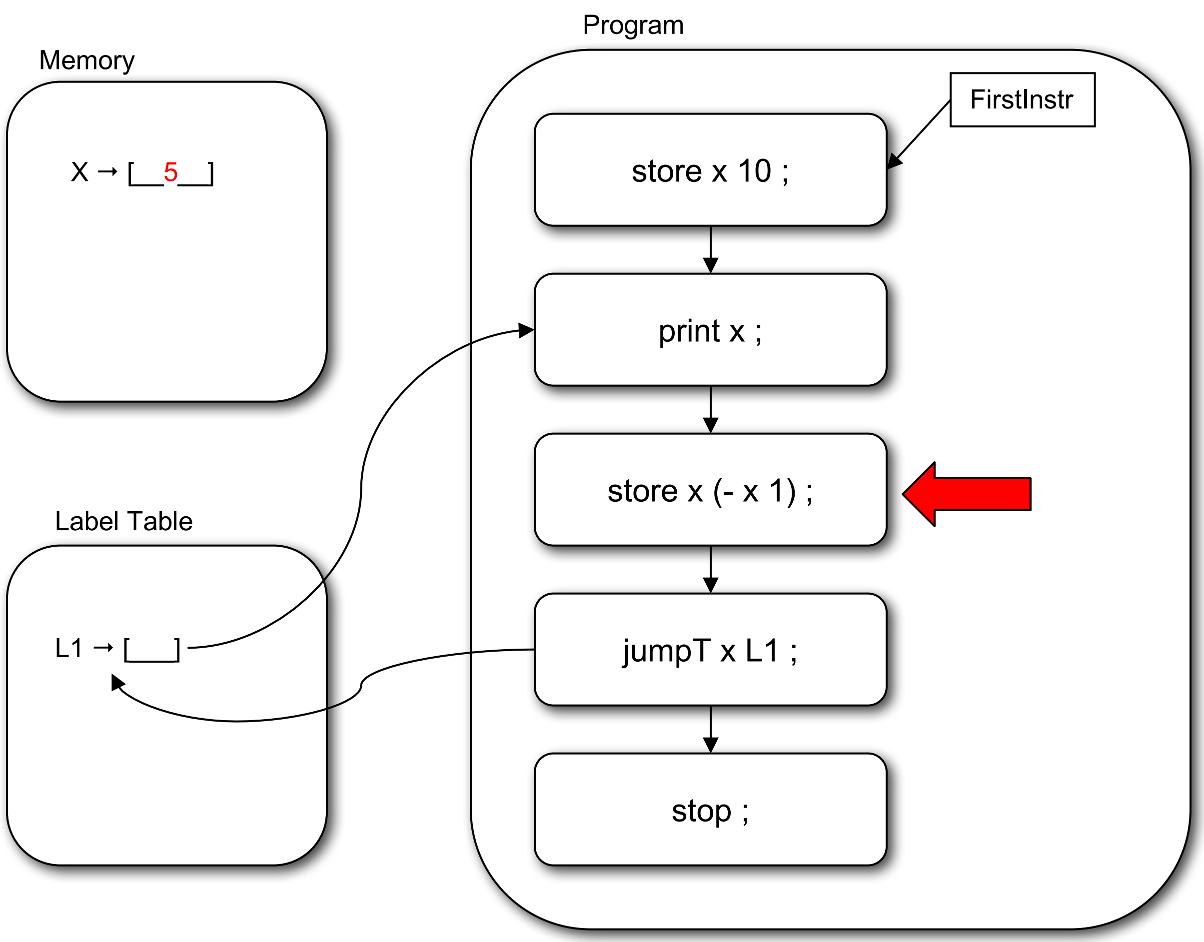
Running the Program

10 9 8 7 6



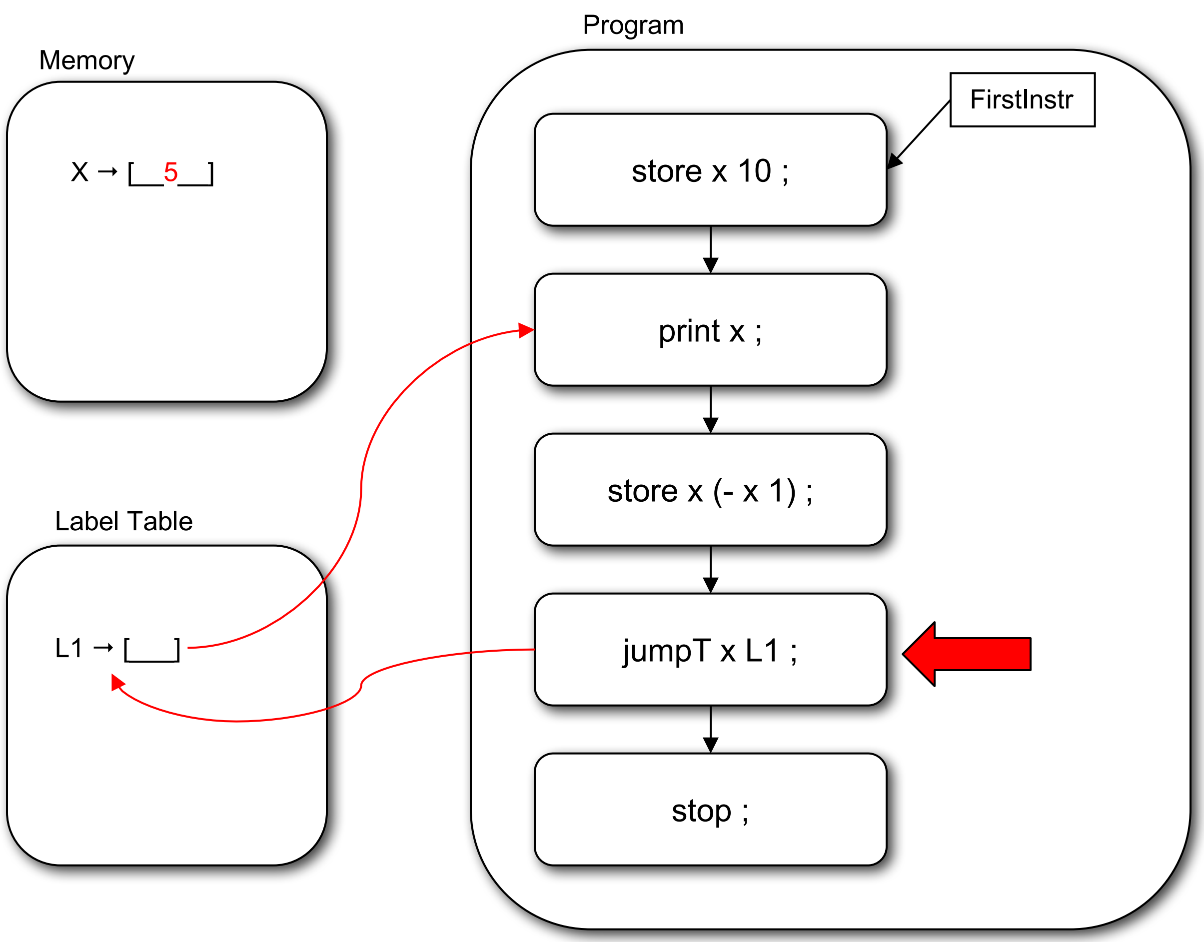
Running the Program

10 9 8 7 6



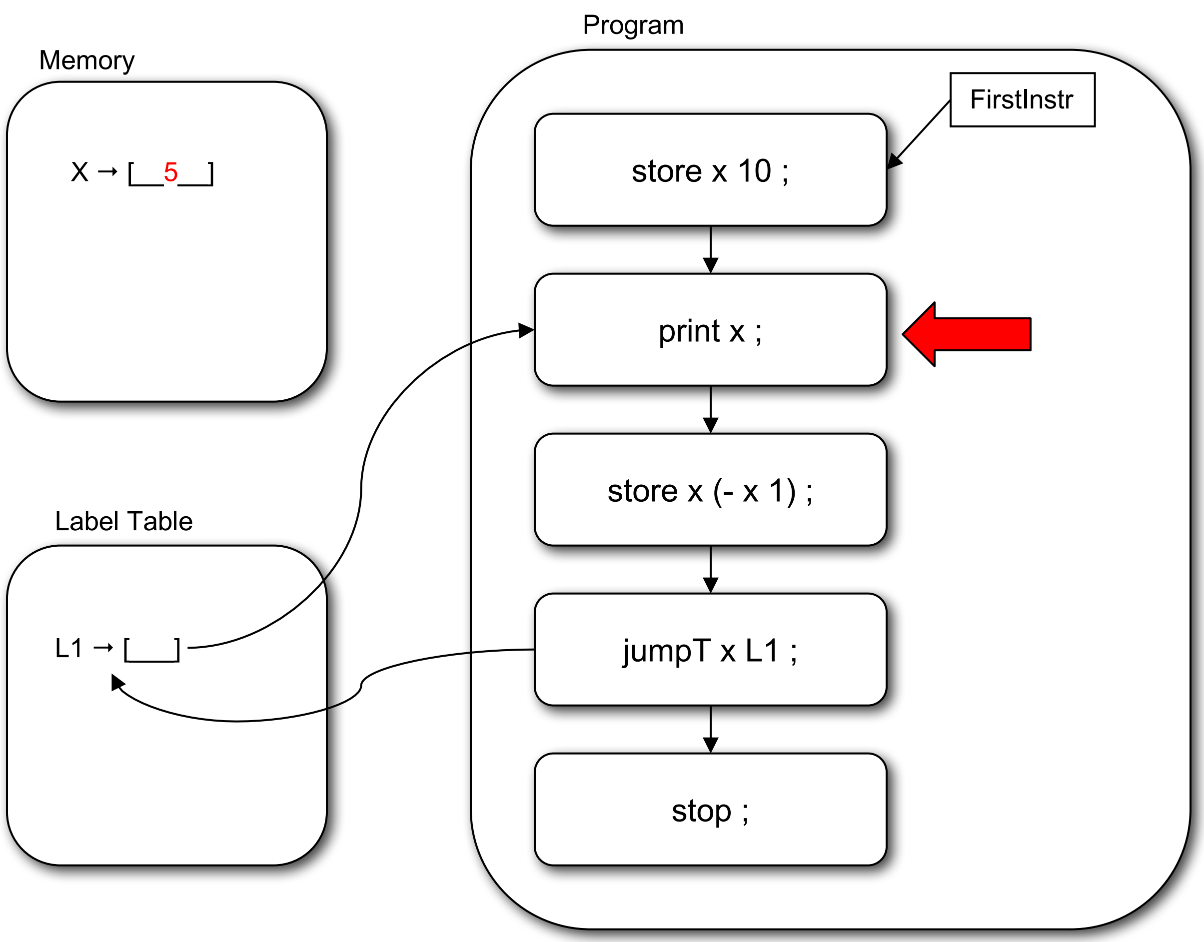
Running the Program

10 9 8 7 6



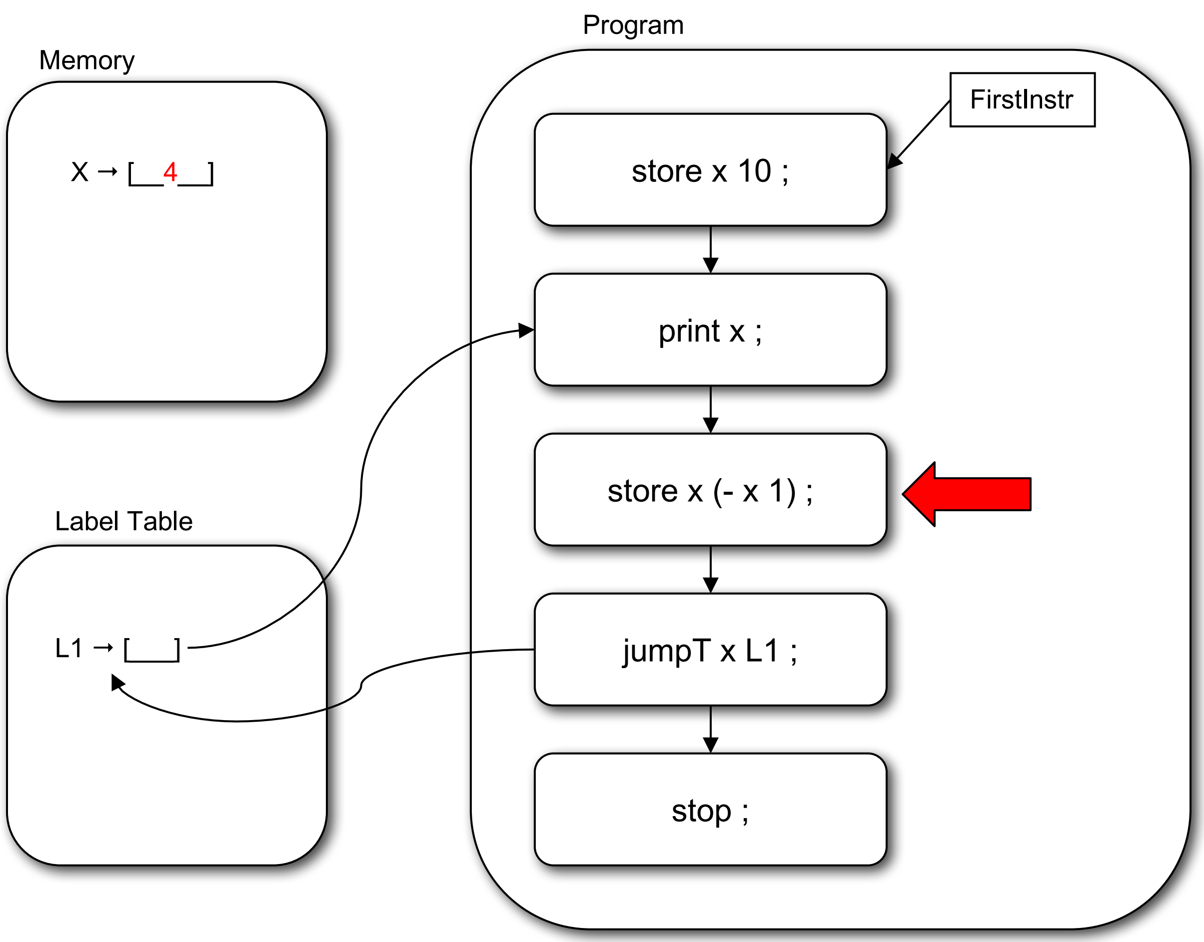
Running the Program

10 9 8 7 6 5



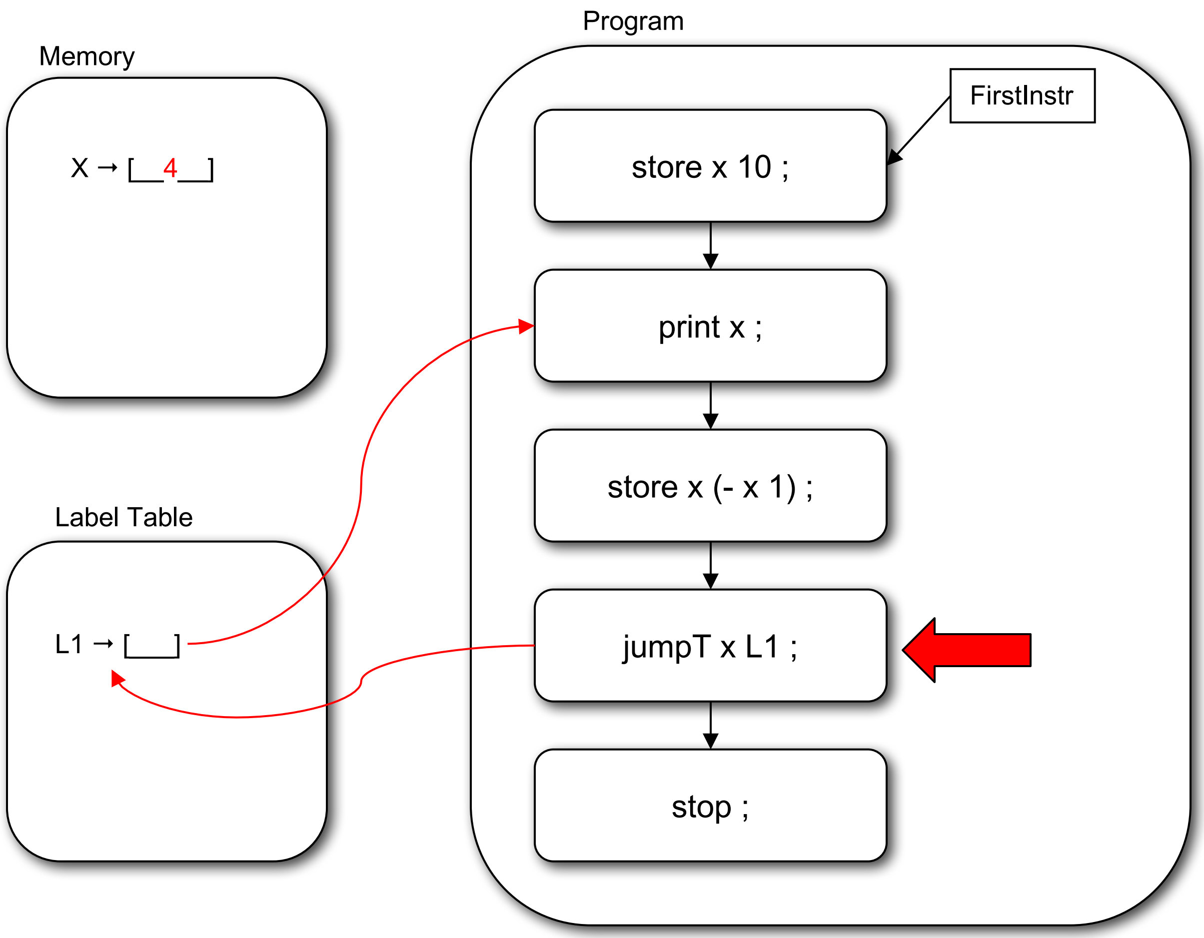
Running the Program

10 9 8 7 6 5



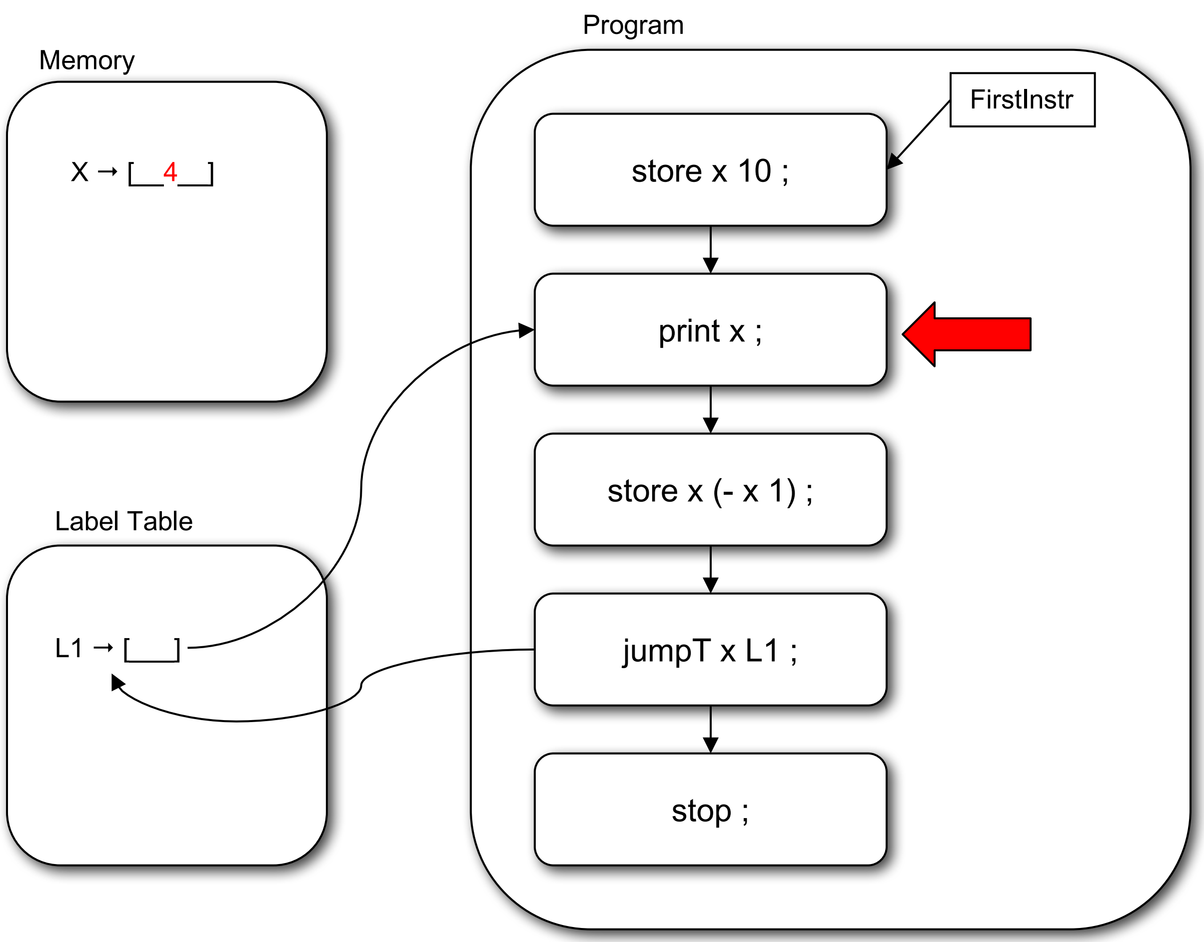
Running the Program

10 9 8 7 6 5



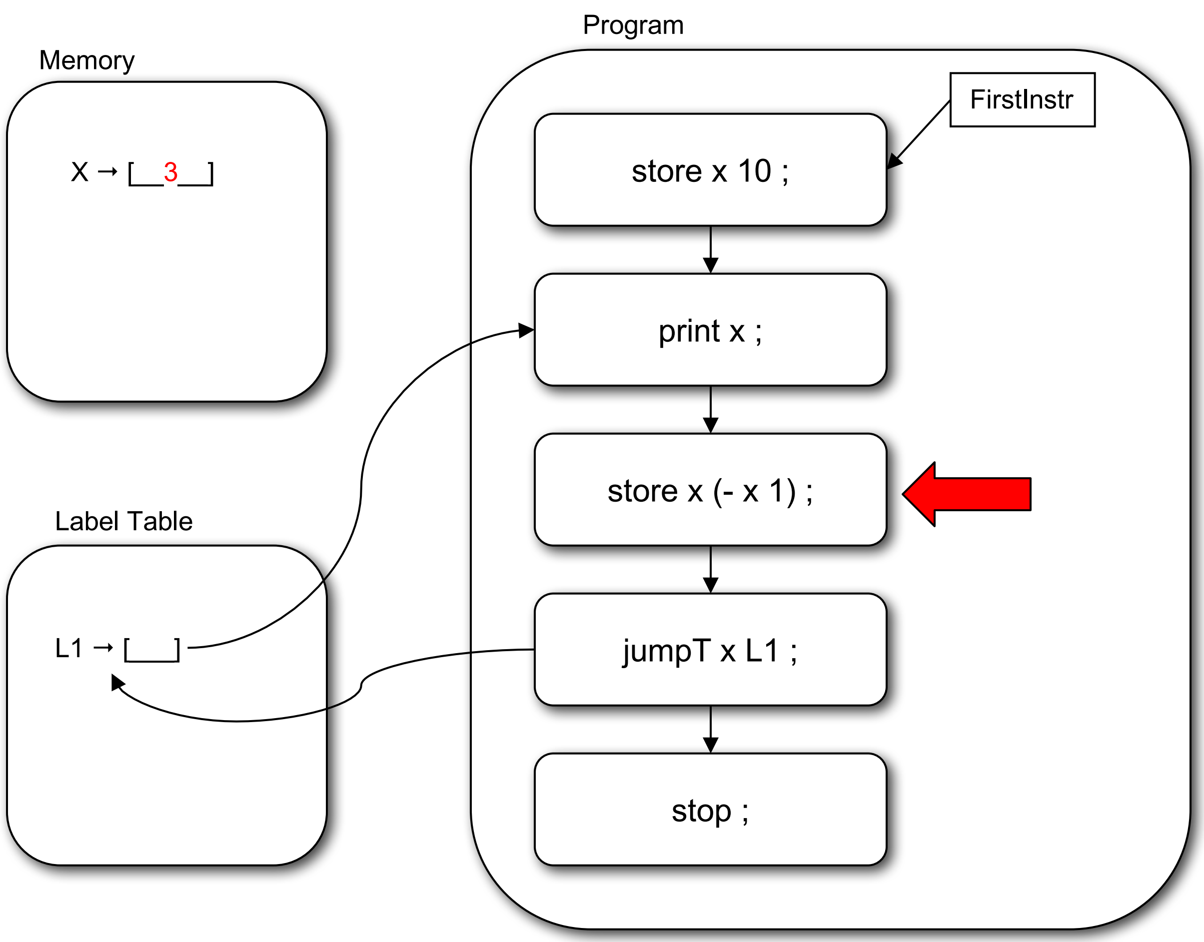
Running the Program

10 9 8 7 6 5 4



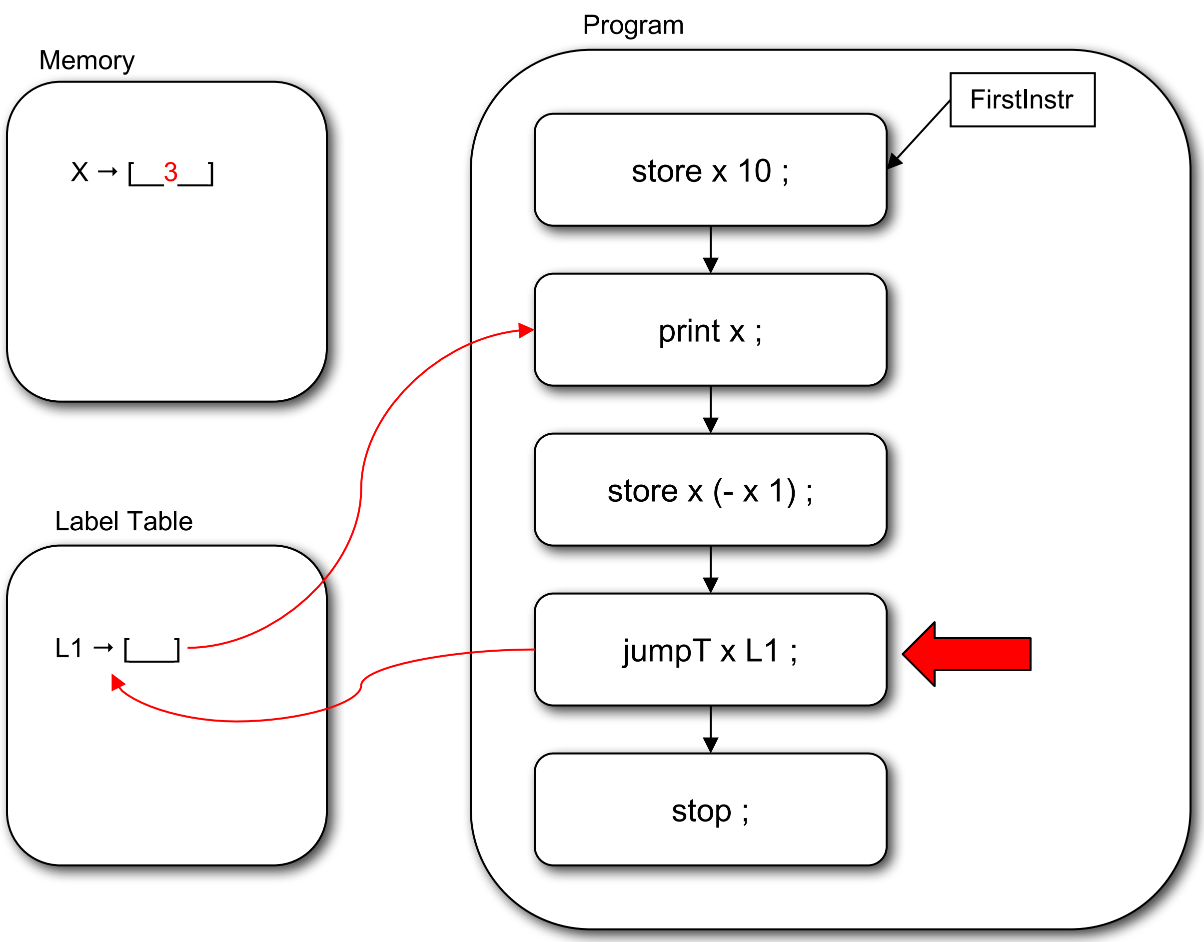
Running the Program

10 9 8 7 6 5 4



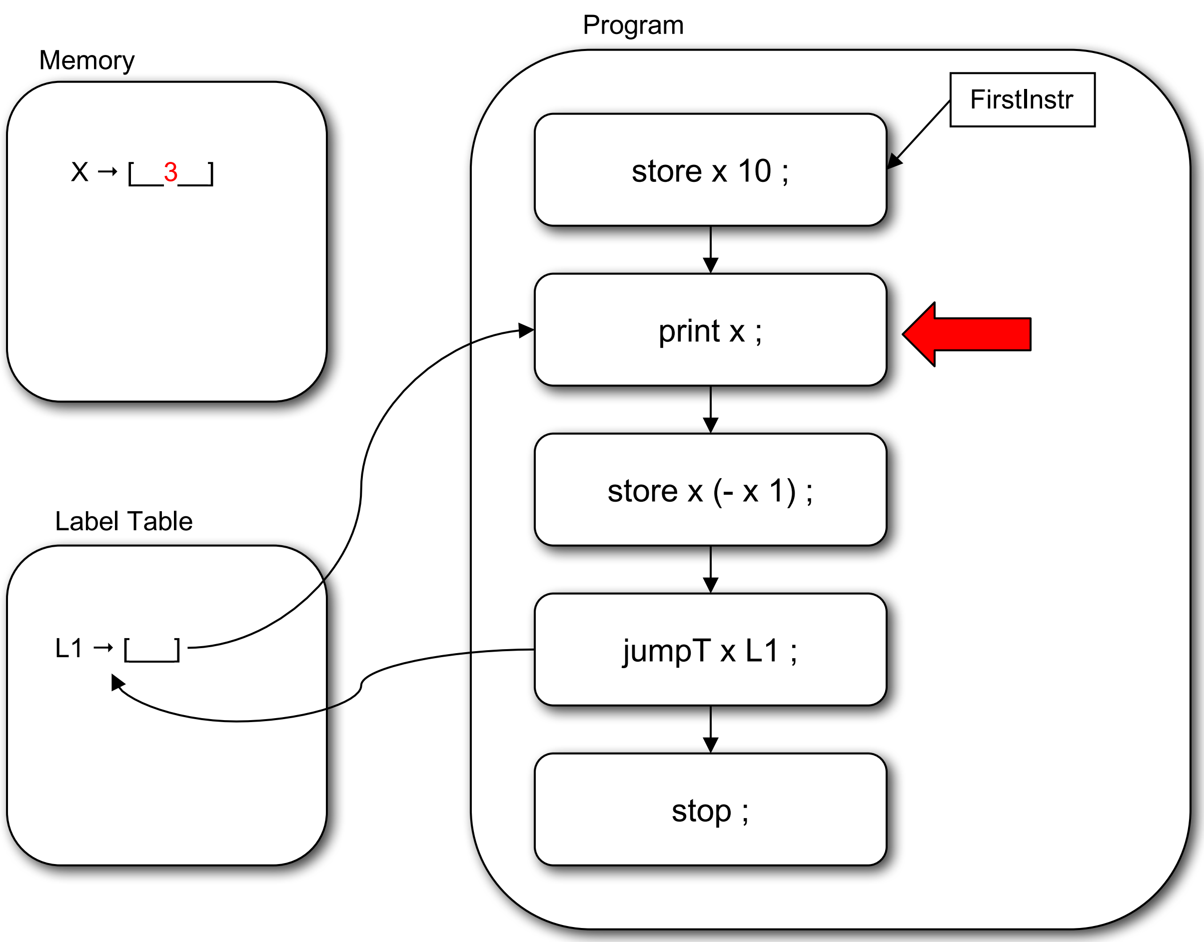
Running the Program

10 9 8 7 6 5 4



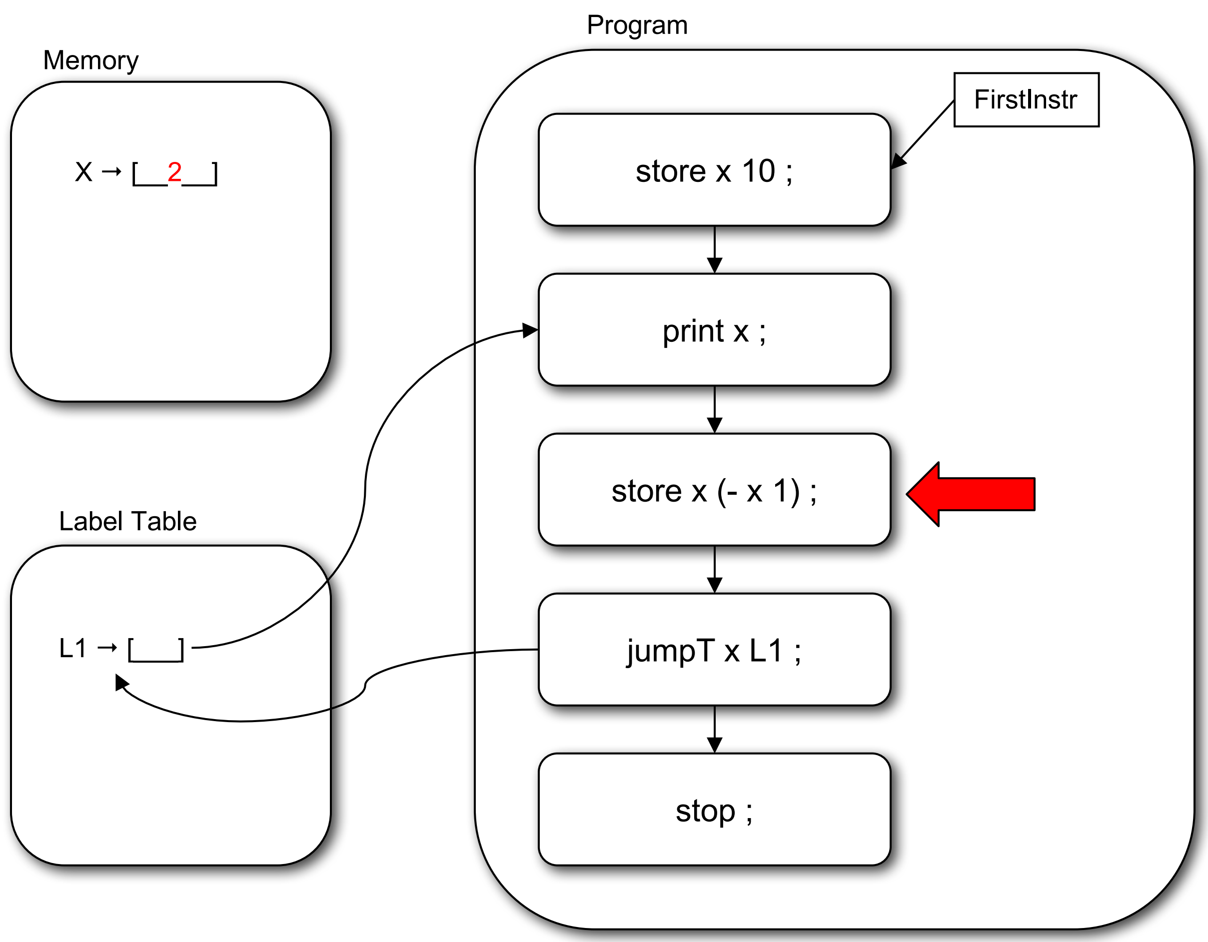
Running the Program

10 9 8 7 6 5 4 3



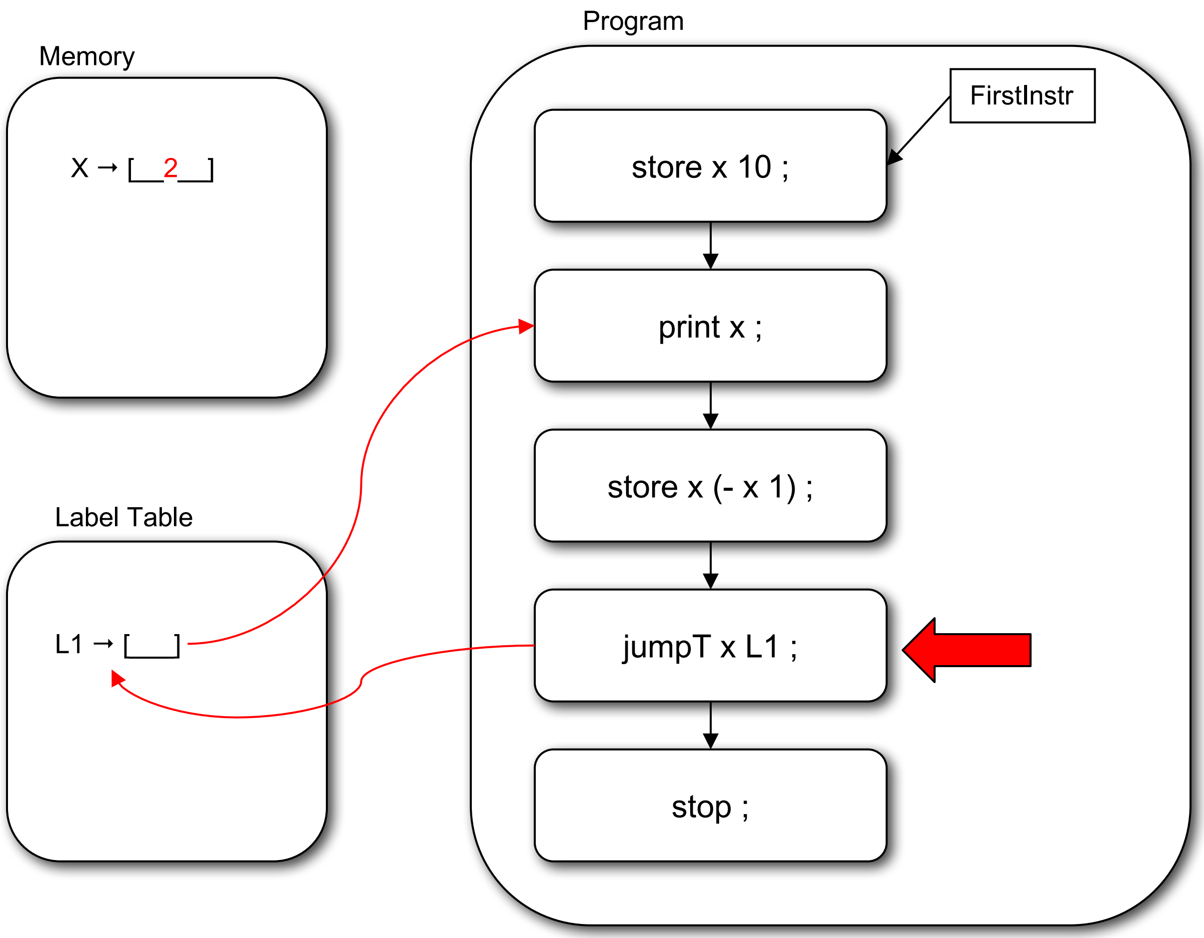
Running the Program

10 9 8 7 6 5 4 3



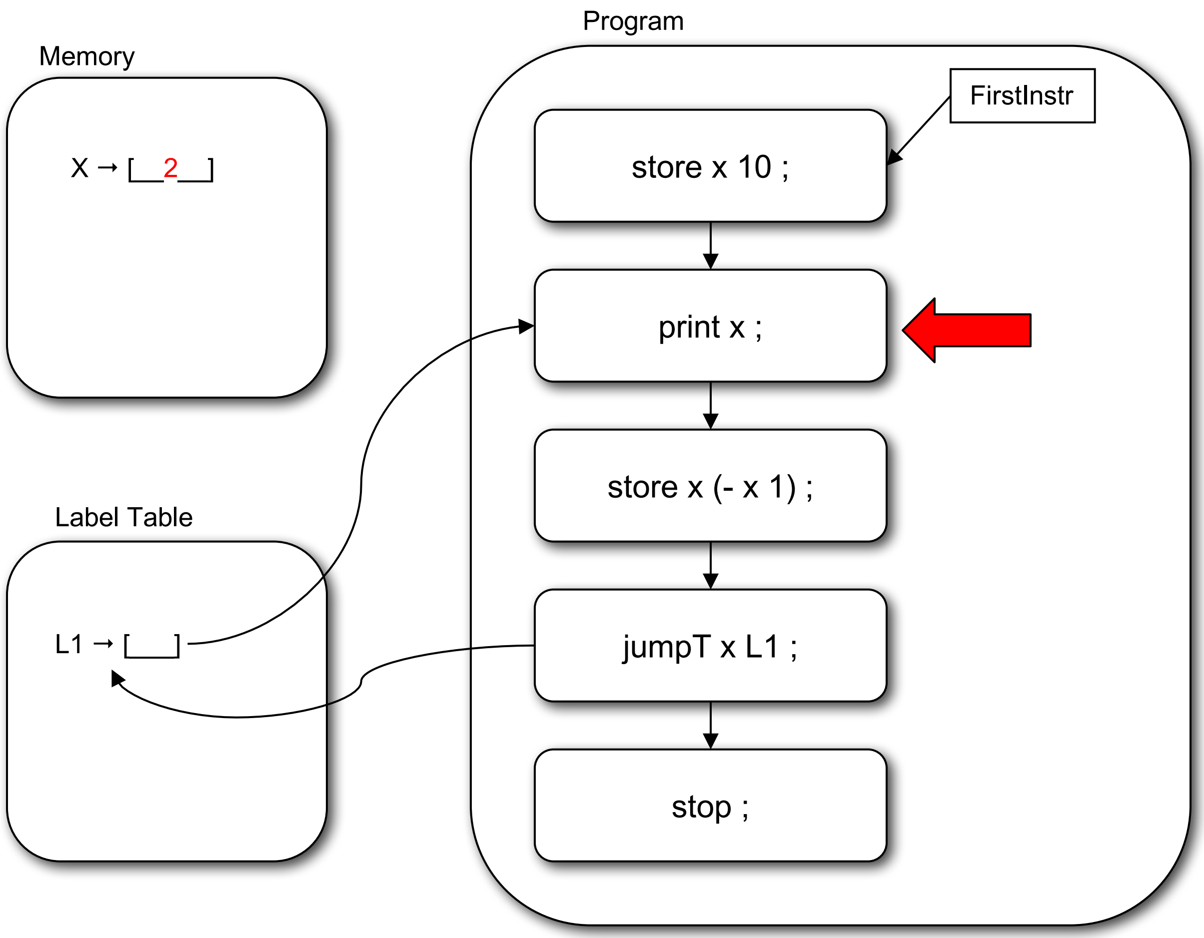
Running the Program

10 9 8 7 6 5 4 3



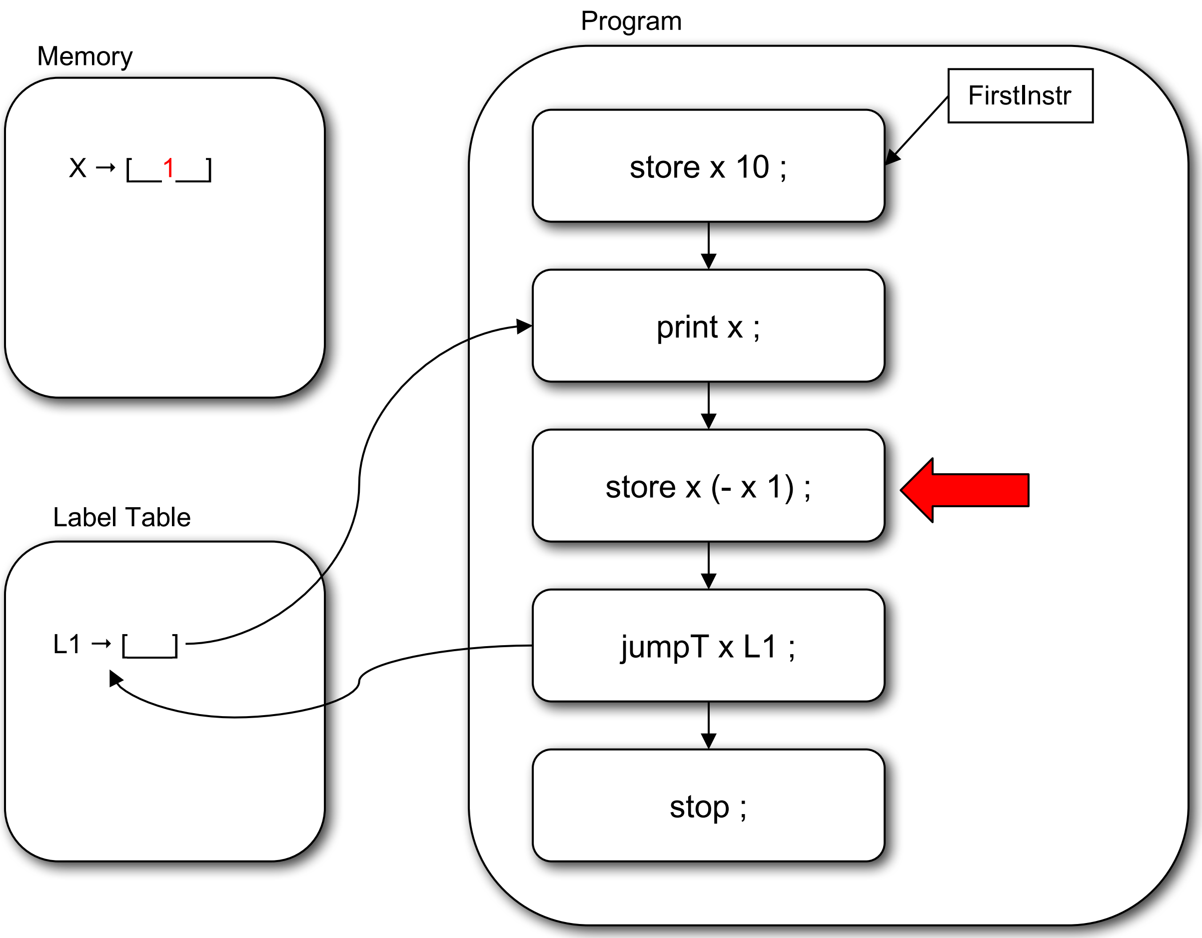
Running the Program

10 9 8 7 6 5 4 3 2



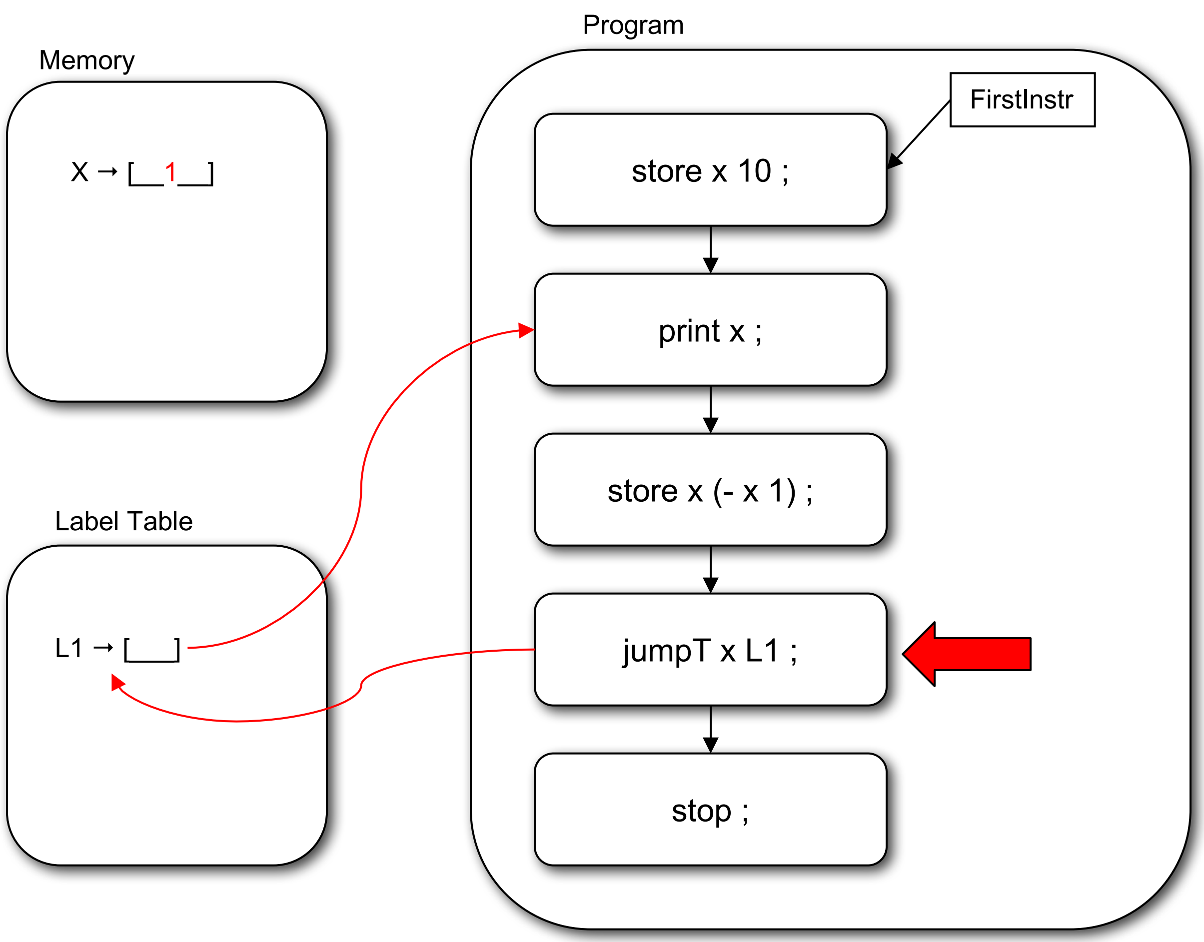
Running the Program

10 9 8 7 6 5 4 3 2



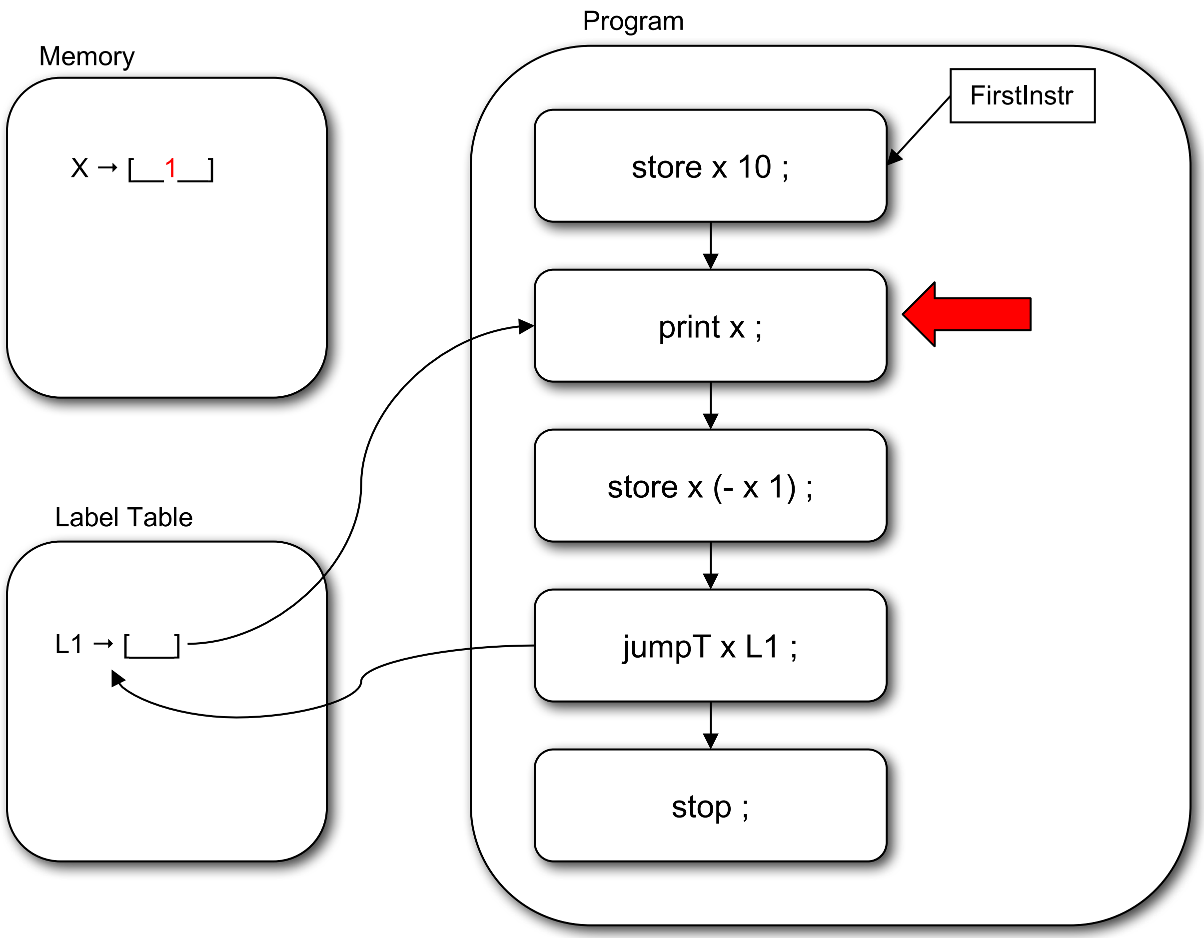
Running the Program

10 9 8 7 6 5 4 3 2



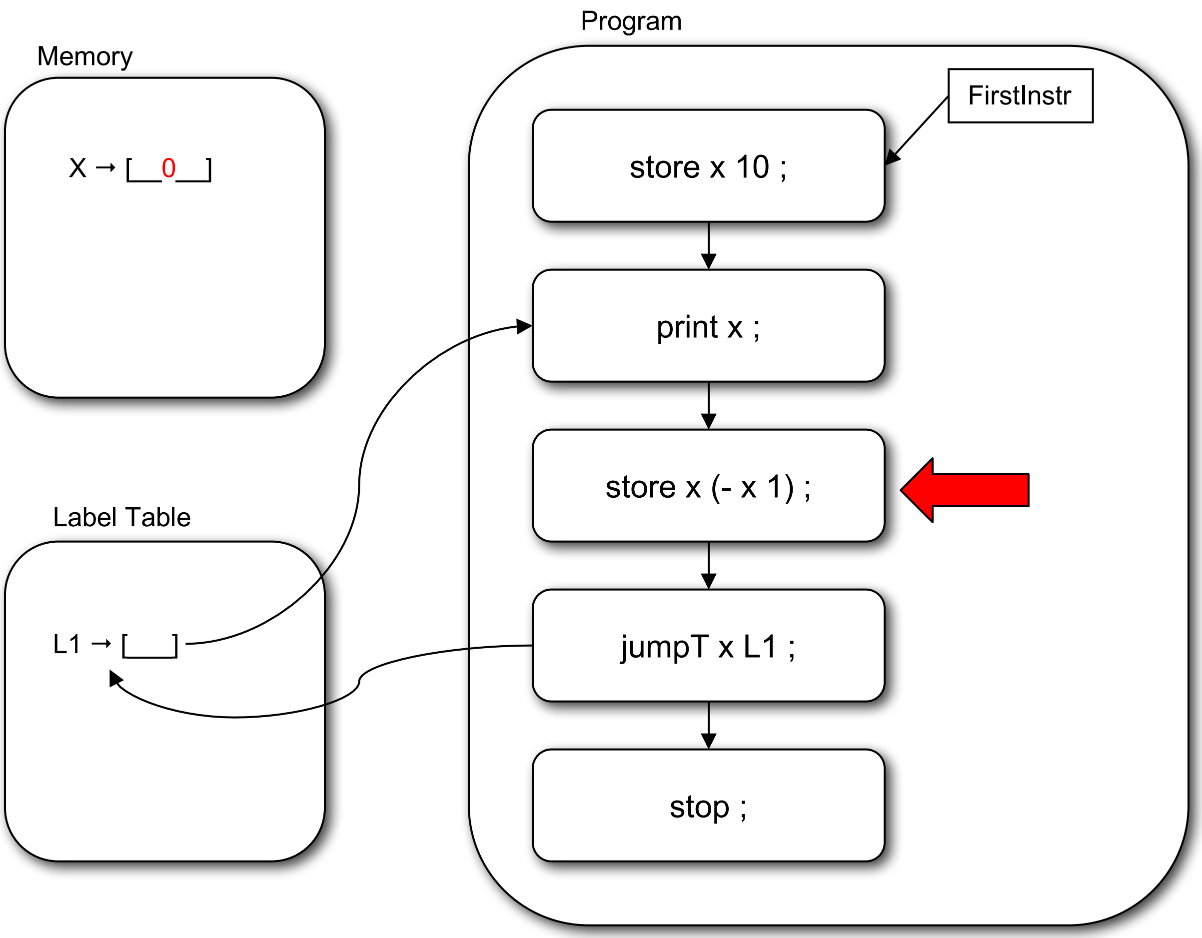
Running the Program

10 9 8 7 6 5 4 3 2 1



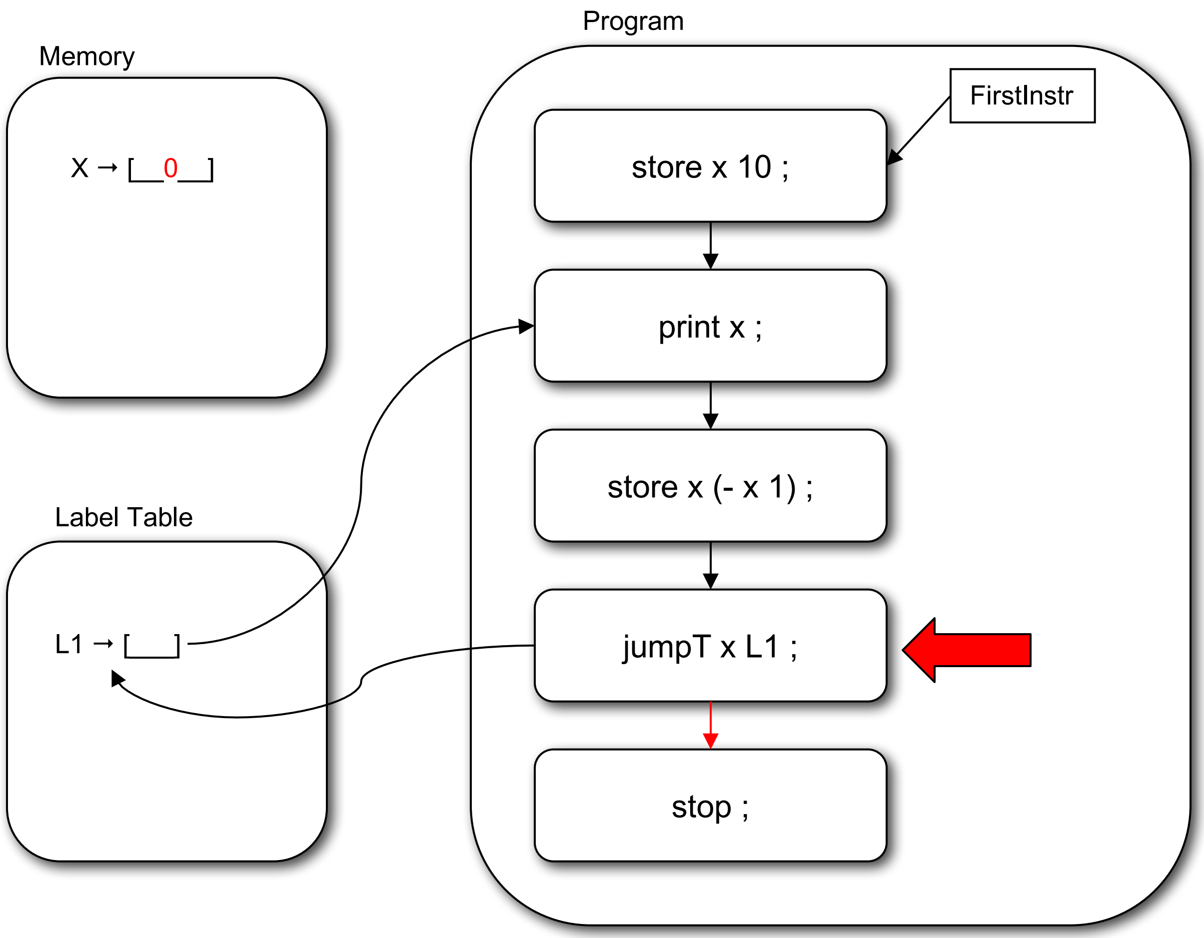
Running the Program

10 9 8 7 6 5 4 3 2 1



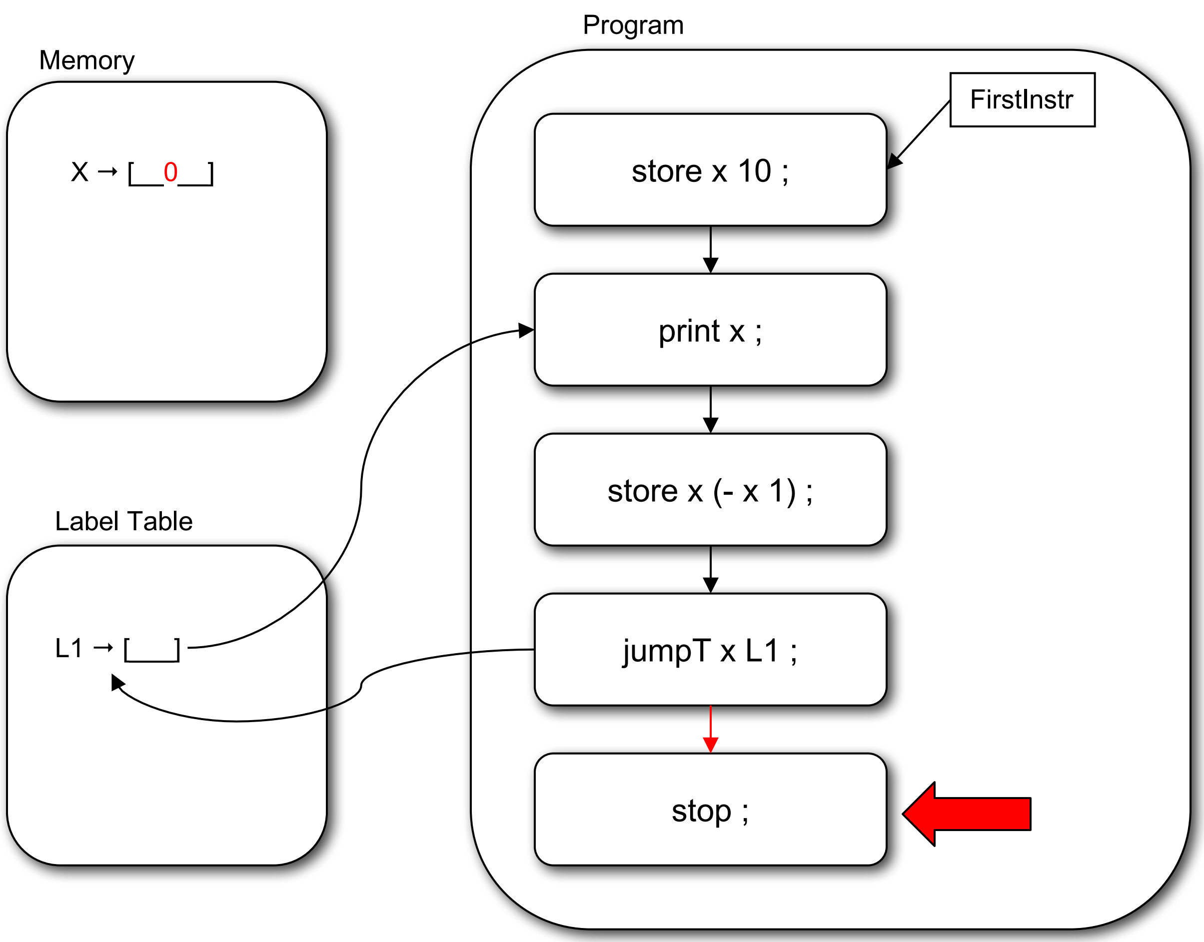
Running the Program

10 9 8 7 6 5 4 3 2 1



Running the Program

10 9 8 7 6 5 4 3 2 1



Implementation

```python
# define and initialize the structures of our abstract machine

class State:

    def __init__(self):
        self.initialize()

    def initialize(self):
        self.program = []
        self.symbol_table = dict()
        self.label_table = dict()
        self.instr_ix = 0

state = State()
```

exp1bytecode_interp_state.py

Implementation

exp1bytecode_interp_gram.py

Observation: the parser no longer performs computations but instead fills out our IR (the state to be precise).

```
# %load code/exp1bytecode_interp_gram
from ply import yacc
from exp1bytecode_lex import tokens, lexer
from exp1bytecode_interp_state import state

def p_prog(_):
  '''
  prog : instr_list
  '''
  pass

def p_instr_list(_):
  '''
  instr_list : labeled_instr instr_list
       | empty
  '''
  pass

def p_labeled_instr(p):
  '''
  labeled_instr : label_def instr
  '''
  # if label exists record it in the label table
  if p[1]:
    state.label_table[p[1]] = state.instr_ix
  # append instr to program
  state.program.append(p[2])
  state.instr_ix += 1

def p_label_def(p):
  '''
  label_def : NAME ':'
            | empty
  '''
  p[0] = p[1]
...
```

Implementation

exp1bytecode_interp_gram.py

Observation: the parser constructs tuple structures…

```
…
def p_instr(p):
  '''
  instr : PRINT exp ';'
          | INPUT NAME ';'
          | STORE NAME exp ';'
          | JUMPT exp label ';'
          | JUMPF exp label ';'
          | JUMP label ';'
          | STOP ';'
          | NOOP ';'
  '''
    # for each instr assemble the appropriate tuple
  if p[1] == 'print':
    p[0] = ('print', p[2])
  elif p[1] == 'input':
    p[0] = ('input', p[2])
  elif p[1] == 'store':
    p[0] = ('store', p[2], p[3])
  elif p[1] == 'jumpT':
    p[0] = ('jumpT', p[2], p[3])
  elif p[1] == 'jumpF':
    p[0] = ('jumpF', p[2], p[3])
  elif p[1] == 'jump':
    p[0] = ('jump', p[2])
  elif p[1] == 'stop':
    p[0] = ('stop',)
  elif p[1] == 'noop':
    p[0] = ('noop',)
  else:
        raise ValueError("Unexpected instr value: %s" % p[1])

def p_label(p):
  '''
    label : NAME
    '''
  p[0] = p[1]
…
```

Implementation

exp1bytecode_interp_gram.py

Tuples! →

```
…
def p_bin_exp(p):
    '''
    exp : '+' exp exp
        | '-' exp exp
        | '*' exp exp
        | '/' exp exp
        | EQ exp exp
        | LE exp exp
    '''
    p[0] = (p[1], p[2], p[3])

def p_uminus_exp(p):
    '''
    exp : '-' exp
    '''
    p[0] = ('UMINUS', p[2])
…
```

```
…
def p_not_exp(p):
    '''
    exp : '!' exp
    '''
    p[0] = ('!', p[2])

def p_paren_exp(p):
    '''
    exp : '(' exp ')'
    '''
      # parens are not necessary in trees
    p[0] = p[2]

def p_var_exp(p):
    '''
      exp : NAME
    '''
    p[0] = ('NAME', p[1])

def p_number_exp(p):
    '''
      exp : NUMBER
    '''
    p[0] = ('NUMBER', int(p[1]))
…
```

Implementation

exp1bytecode_interp_gram.py

```
…
def p_empty(p):
  '''
    empty :
  '''
  p[0] = ''

def p_error(t):
    print("Syntax error at '%s'" % t.value)

parser = yacc.yacc(debug=False, tabmodule='exp1bytecodeparsetab')
```

A Note on the Expressions

- We are delaying the evaluation of expressions until we have the IR constructed
- We need to have some sort of representation of the expression value that we can evaluate later to actually compute a value.
- The idea is that we construct an expression or term tree from the source expression and that term tree can then be evaluated later to compute an actual integer value.
- Actually we are constructing a tuple expression.

A Note on the Expressions

```
'''
exp : '+' exp exp
    | '-' exp exp
    | '*' exp exp
    | '/' exp exp
    | EQ exp exp
    | LE exp exp
'''
p[0] = (p[1], p[2], p[3])
```

```
'''
exp : NUMBER
'''
p[0] = ('NUMBER', int(p[1]))
```

According to these rules the expression,

=< + 3 2 * 3 2

gives rise to the term tree,

('=<', ('+', ('NUMBER', 3), ('NUMBER', 2)), ('*', ('NUMBER', 3), ('NUMBER', 2)))

Testing our Parser

In [6]:
```
from exp1bytecode_interp_state import state
from exp1bytecode_interp_gram import parser
import pprint
pp = pprint.PrettyPrinter()
```

```
Generating LALR tables
WARNING: 9 shift/reduce conflicts
```

Setting up the input stream with our Exp1bytecode program.

In [7]:
```
input_stream = \
'''
   store x 10 ;
L1:
   print x ;
   store x (- x 1) ;
   jumpT x L1 ;
   stop ;
'''
```

Running the parser.

In [8]:
```
parser.parse(input_stream)
```

Testing our Parser

```
In [9]:  # print out the program list of statement tuples
         pp.pprint(state.program)

         [('store', 'x', ('NUMBER', 10)),
          ('print', ('NAME', 'x')),
          ('store', 'x', ('-', ('NAME', 'x'), ('NUMBER', 1))),
          ('jumpT', ('NAME', 'x'), 'L1'),
          ('stop',)]
```

```
In [10]: # print out the label table
         pp.pprint(state.label_table)

         {'L1': 1}
```

```
In [11]: # print the symbol table
         pp.pprint(state.symbol_table)

         {}
```

The symbol table is empty since we have not executed the program yet! We have just initialized our abstract machine.

Interpretation – running the abstract machine

- In order to interpret the programs in our IR we need two functions:
  - The first one is the interpretation of instructions on the program list.
  - The second one for the interpretation of expresssion

Interpreting Instructions

exp1bytecode_interp.py

One big loop that interprets the instructions on the list (program)

```
def interp_program():
  'abstract bytecode machine'

  # We cannot use the list iterator here because we
  # need to be able to interpret jump instructions

  # start at the first instruction in program
  state.instr_ix = 0

  # keep interpreting until we run out of instructions
  # or we hit a 'stop'
  while True:
   if state.instr_ix == len(state.program):
     # no more instructions
     break
   else:
    # get instruction from program
    instr = state.program[state.instr_ix]

    # instruction format: (type, [arg1, arg2, ...])
    type = instr[0]

        # interpret instruction
    if type == 'print':
      # PRINT exp
      exp_tree = instr[1]
      val = eval_exp_tree(exp_tree)
      print("> {}".format(val))
      state.instr_ix += 1

    elif type == 'input':
      # INPUT NAME
      var_name = instr[1]
            val = input("Please enter a value for {}: ".format(var_name))
            state.symbol_table[var_name] = int(val)
      state.instr_ix += 1

    elif type == 'store':
      # STORE type exp
      var_name = instr[1]
      val = eval_exp_tree(instr[2])
            state.symbol_table[var_name] = val
      state.instr_ix += 1
...
```

Interpreting Instructions

exp1bytecode_interp.py

```
…
    elif type == 'jumpT':
        # JUMPT exp label
        val = eval_exp_tree(instr[1])
        if val:
            state.instr_ix = state.label_table.get(instr[2], None)
        else:
            state.instr_ix += 1

    elif type == 'jumpF':
        # JUMPF exp label
        val = eval_exp_tree(instr[1])
        if not val:
            state.instr_ix = state.label_table.get(instr[2], None)
        else:
            state.instr_ix += 1

    elif type == 'jump':
        # JUMP label
        state.instr_ix = state.label_table.get(instr[1], None)

    elif type == 'stop':
        # STOP
        break

    elif type == 'noop':
        # NOOP
        state.instr_ix += 1

    else:
        raise ValueError("Unexpected instruction type: {}".format(p[1]))
```

Interpreting Expressions

exp1bytecode_interp.py

Recursive function that walks the expression tree and evaluates it.

```
def eval_exp_tree(node):
    'walk expression tree and evaluate to an integer value'

    # tree nodes are tuples (TYPE, [arg1, arg2,...])

  type = node[0]

  if type == '+':
    # '+' exp exp
    v_left = eval_exp_tree(node[1])
    v_right = eval_exp_tree(node[2])
    return v_left + v_right

  elif type == '-':
    # '-' exp exp
    v_left = eval_exp_tree(node[1])
    v_right = eval_exp_tree(node[2])
    return v_left - v_right

  elif type == '*':
    # '*' exp exp
    v_left = eval_exp_tree(node[1])
    v_right = eval_exp_tree(node[2])
    return v_left * v_right

  elif type == '/':
    # '/' exp exp
    v_left = eval_exp_tree(node[1])
    v_right = eval_exp_tree(node[2])
    return v_left // v_right

…
```

Interpreting Expressions

exp1bytecode_interp.py

```
…
  elif type == '==':
    # '=' exp exp
    v_left = eval_exp_tree(node[1])
    v_right = eval_exp_tree(node[2])
    return 1 if v_left == v_right else 0

  elif type == '<=':
    # '<=' exp exp
    v_left = eval_exp_tree(node[1])
    v_right = eval_exp_tree(node[2])
    return 1 if v_left <= v_right else 0

    elif type == 'UMINUS':
    # 'UMINUS' exp
    val = eval_exp_tree(node[1])
    return - val

  elif type == '!':
    # '!' exp
    val = eval_exp_tree(node[1])
    return 0 if val != 0 else 1

  elif type == 'NAME':
    # 'NAME' var_name
    return state.symbol_table.get(node[1],0)

  elif type == 'NUMBER':
    # NUMBER val
    return node[1]

  else:
    raise ValueError("Unexpected instruction type: {}".format(type))
```

Top-level Function

exp1bytecode_interp.py

```
def interp(input_stream):
    'driver for our Exp1bytecode interpreter.'

    # initialize our abstract machine
    state.initialize()

    # build the IR
    parser.parse(input_stream, lexer=lexer)

    # interpret the IR
    interp_program()
```

Interpreter Script

```
#!/usr/bin/env python

from argparse import ArgumentParser
from exp1bytecode_lex import lexer
from exp1bytecode_interp_gram import parser
from exp1bytecode_interp_state import state

#########################################################################################
def interp_program():
 'execute abstract bytecode machine'
…

#########################################################################################
def eval_exp_tree(node):
  'walk expression tree and evaluate to an integer value'
…

#########################################################################################
def interp(input_stream):
  'driver for our Exp1bytecode interpreter.'
…

#########################################################################################
if __name__ == '__main__':
  # parse command line args
  aparser = ArgumentParser()
  aparser.add_argument('input')

  args = vars(aparser.parse_args())

  f = open(args['input'], 'r')
  input_stream = f.read()
  f.close()

  interp(input_stream=input_stream)
```

Testing our Interpreter

In [22]:
```
from exp1bytecode_interp import interp
```

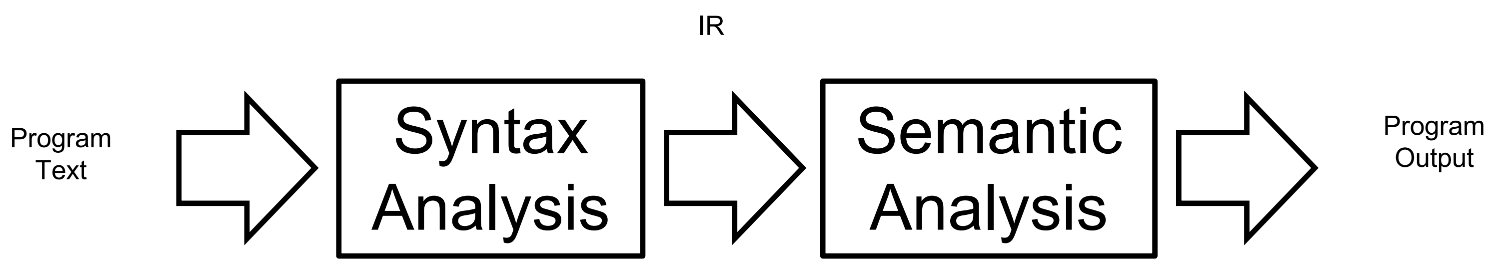
In [20]:
```
input_stream = \
'''
    store x 10 ;
L1:
    print x ;
    store x (- x 1) ;
    jumpT x L1 ;
    stop ;
'''
```

In [21]:
```
interp(input_stream)
```

```
> 10
> 9
> 8
> 7
> 6
> 5
> 4
> 3
> 2
> 1
```

Interpreter with IR

- The advantage of IR based interpretation is that we are <u>decoupling</u> program recognition (parsing/reading) from executing the program.
- As we saw this decoupling allows us to create IRs that are convenient to use!



Assignments

- Assignment #4 – see website