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**

Reading

- Chap 4

Exp1bytecode Language Design

- New statements:
  - stop ;
  - noop ;
  - jumpT exp label ;
  - jumpF exp label ;
  - jump label ;
  - Input name ;
  - **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

Listing 4.1: Grammar for the Exp1bytecode language.

```
instr_list : (labeled_instr)*

labeled_instr : label_def instr
              | instr

label_def : label \:

instr : print exp ;
      | store var exp ;
      | input var ;
      | jumpt exp label ;
      | jumpf exp label ;
      | jump label ;
      | stop ;
      | noop ;

exp : + exp exp
    | - exp exp?
    | \* exp exp
    | / exp exp
    | ! exp
    | == exp exp
    | =< exp exp
    | \( exp \)
    | var
    | num

label : <any valid label name>
var : <any valid variable name>
num : <any valid integer number>
```

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

- Values are computed as soon as structure is recognized
- All relevant information has to be accessible at parse time

```python
def exp(stream):
    token = stream.pointer()
    if token.type in ['PLUS']:
        stream.match('PLUS')
        vleft = exp(stream)
        vright = exp(stream)
        return vleft+vright
    elif token.type in ['MINUS']:
        stream.match('MINUS')
        vleft = exp(stream)
        vright = exp(stream)
        return vleft-vright
    elif token.type in ['LPAREN']:
        stream.match('LPAREN')
        v = exp(stream)
        stream.match('RPAREN')
        return v
    elif token.type in ['NAME']:
        global symboltable
        name = var(stream)
        return symboltable.get(name,0)
    elif token.type in ['NUMBER']:
        v = num(stream)
        return v
    else:
        raise SyntaxError("exp: syntax error at {}".format(token.value))
```

Syntax directed interpretation fails…

```
instr_list : (labeled_instr)*

labeled_instr : label_def instr
              | instr

label_def : label \:

instr : print exp ;
      | store var exp ;
      | input var ;
      | jumpt exp label ;
      | jumpf exp label ;
      | jump label ;
      | stop ;
      | noop ;
```

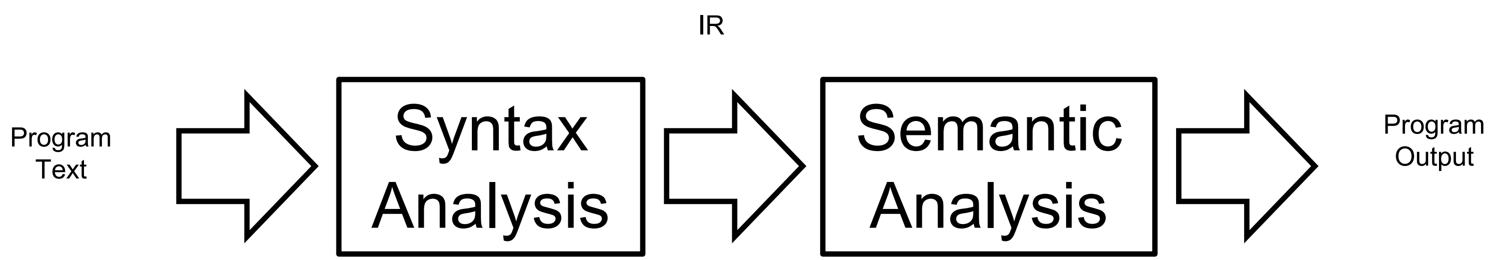
In exp1bytecode we see that label definitions are *non-local* to jump statements and therefore *cannot* be executed in a syntax directed manner.

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

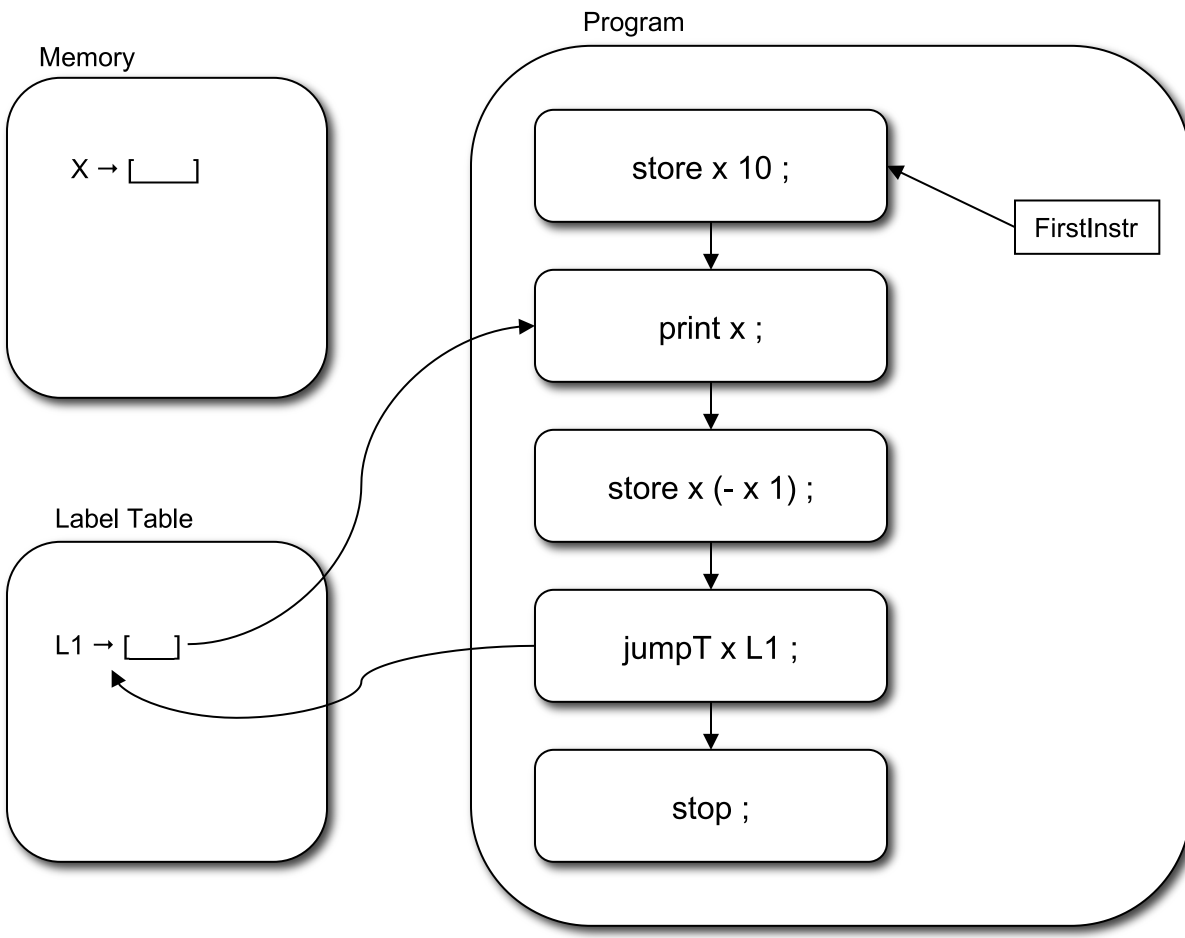


IR Design

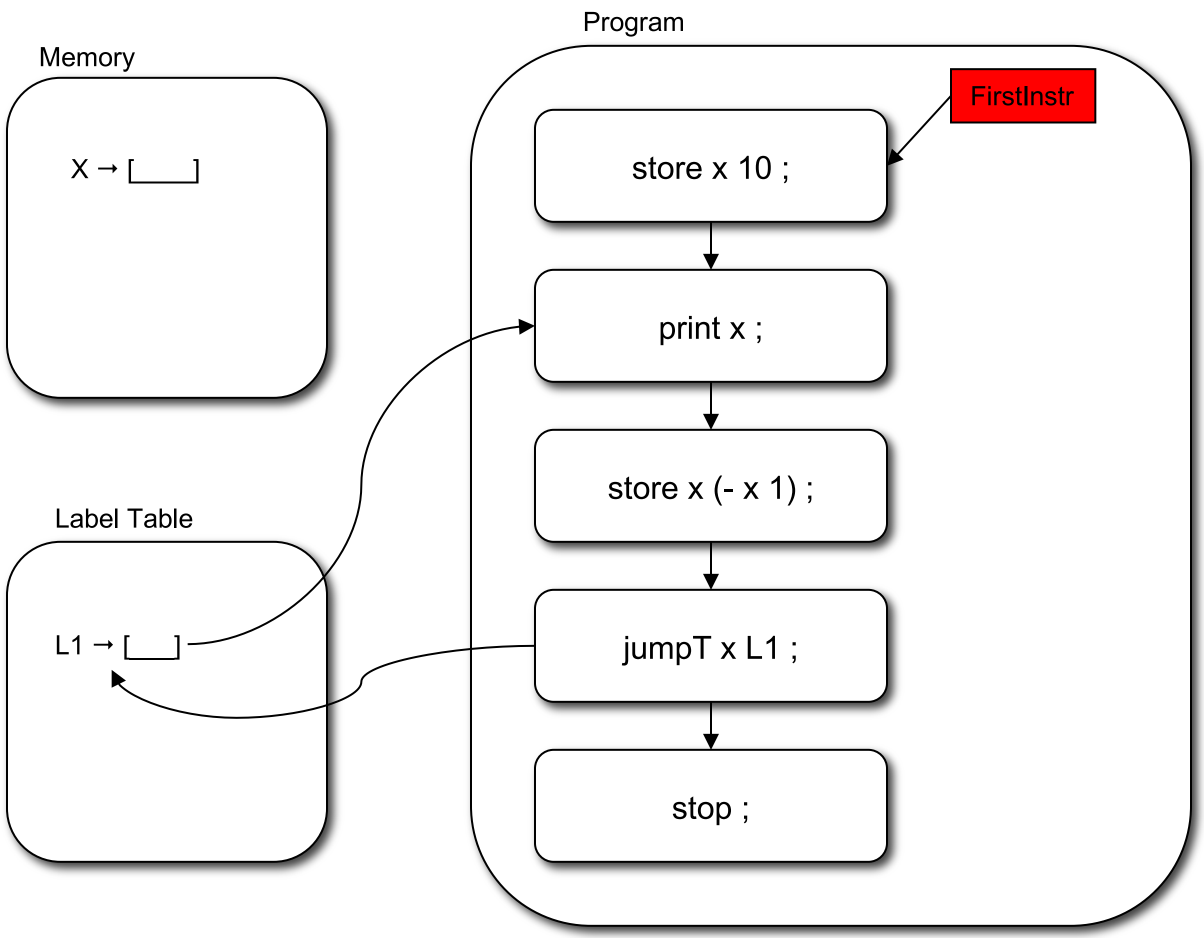
- For variable values we will use the *dictionary based symbol table* from before
- As our IR we will use an abstract representation of the program as a *list of instructions*
- For label definitions we will use a *label lookup* table that associates labels with instructions in our list of instructions

IR Design

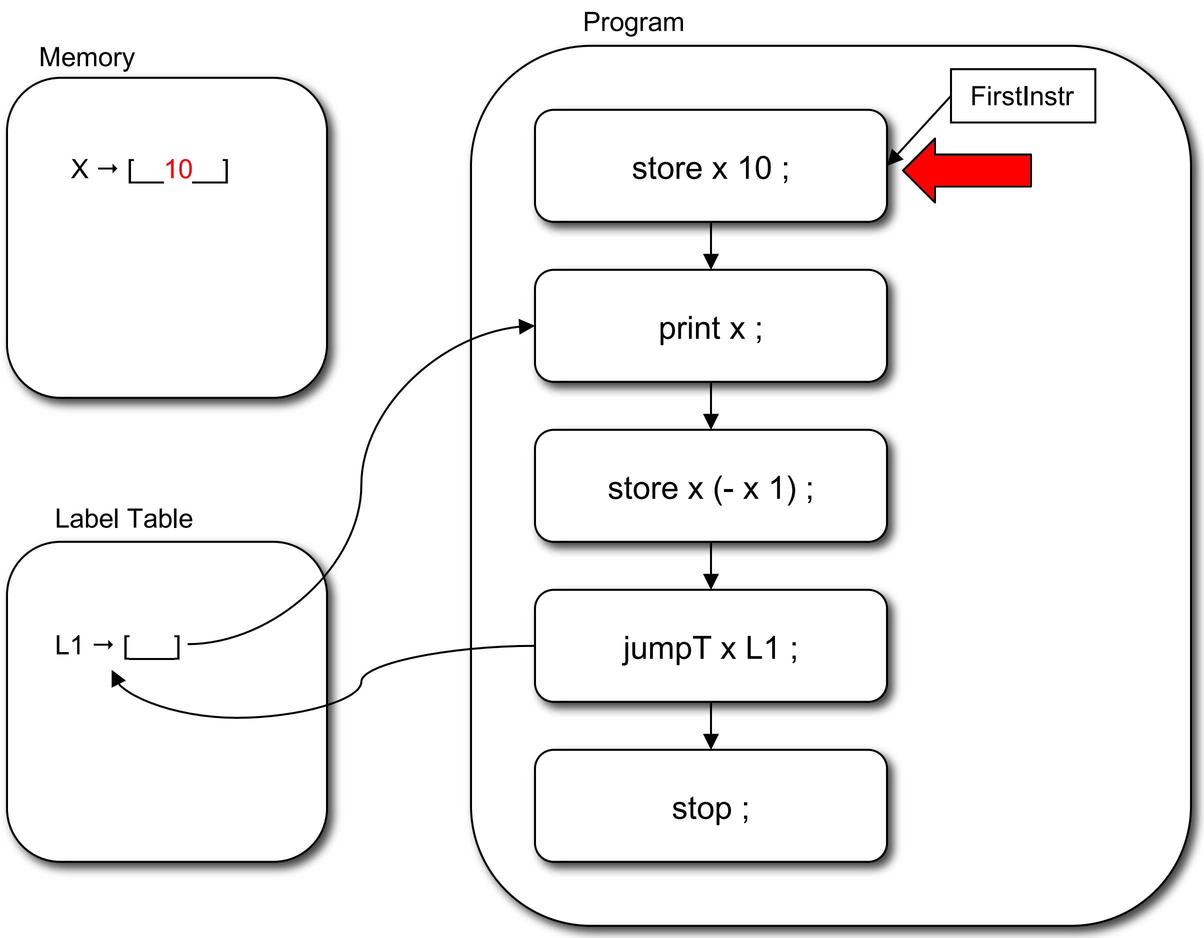
```
   store x 10 ;
L1:
   print x ;
   store x (- x 1) ;
   jumpT x L1 ;
   stop ;
```



Running the Program

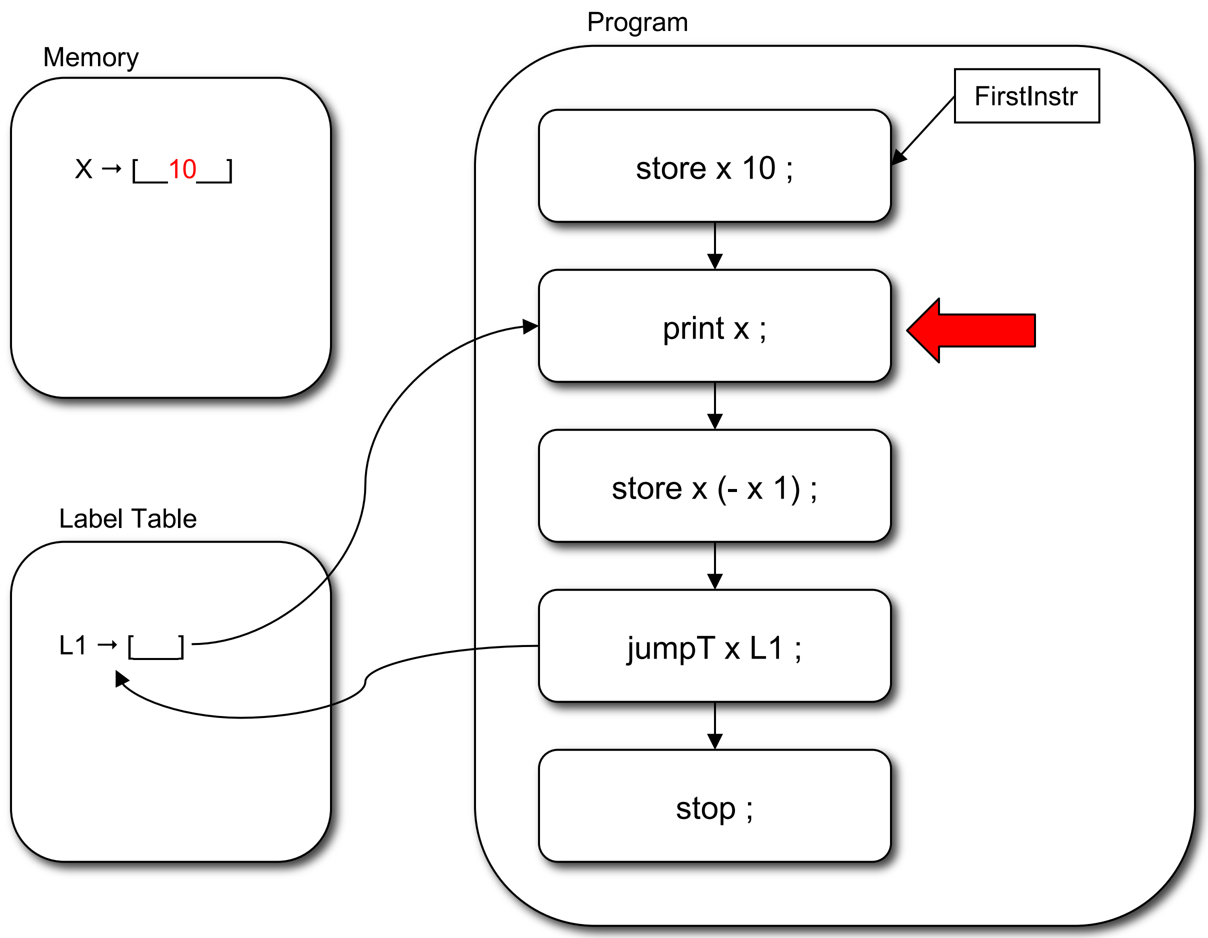


Running the Program



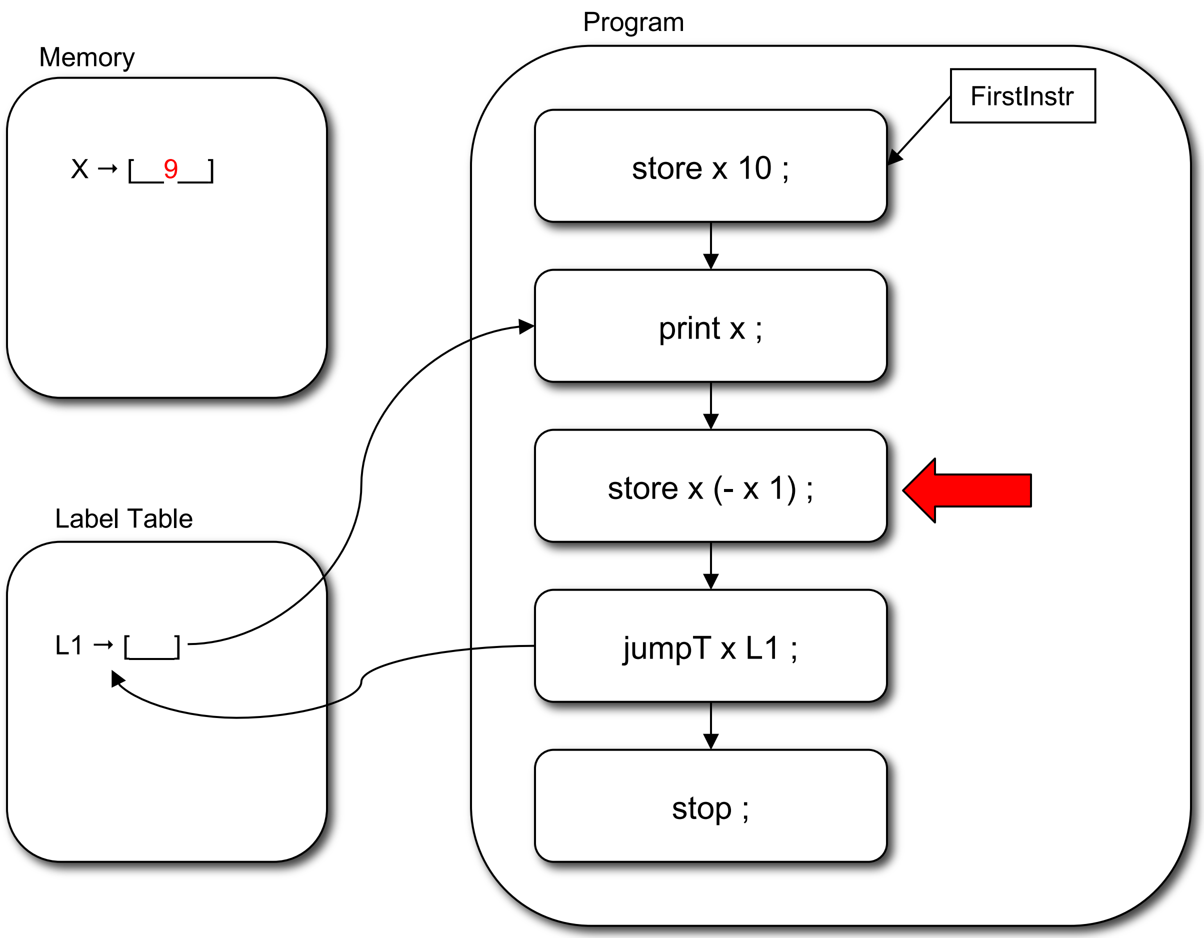
Running the Program

10



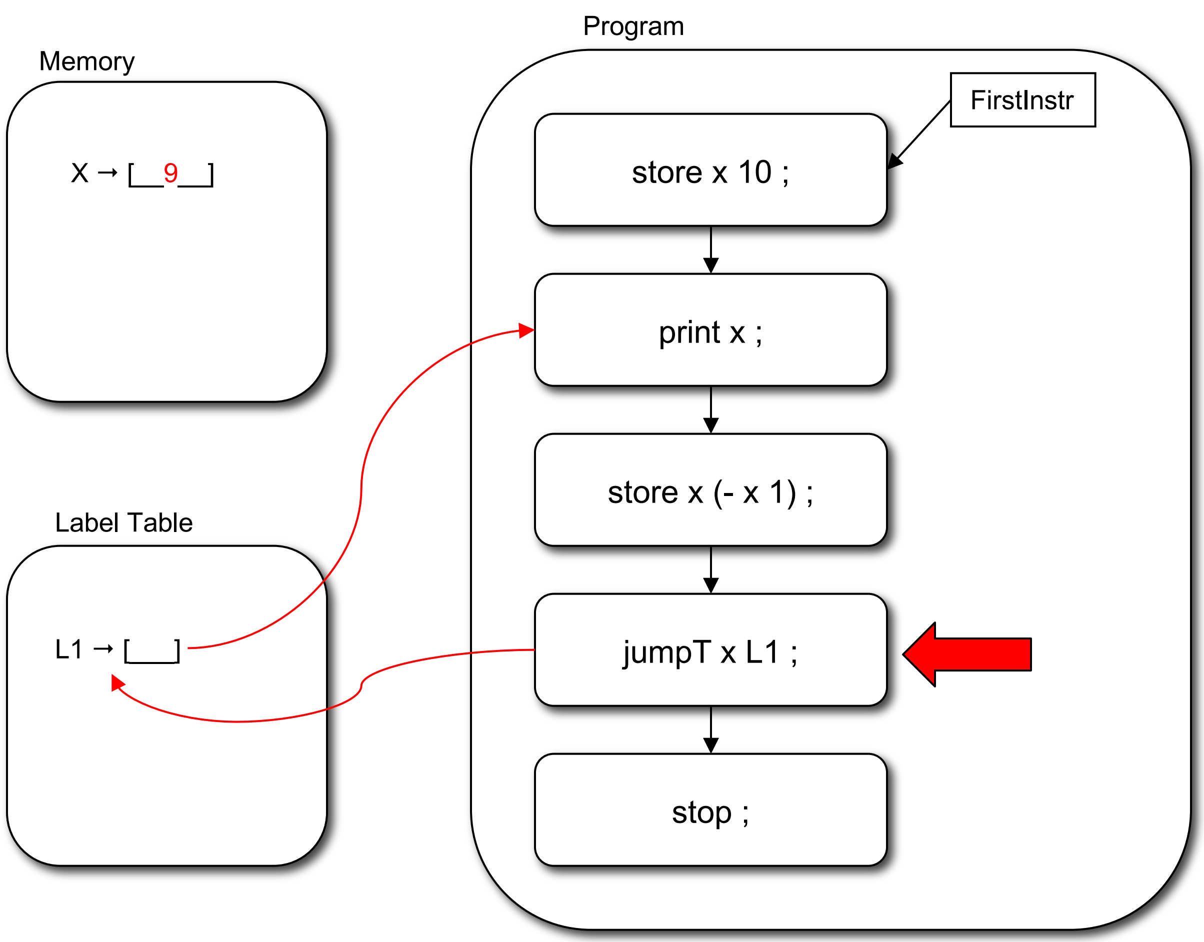
Running the Program

10



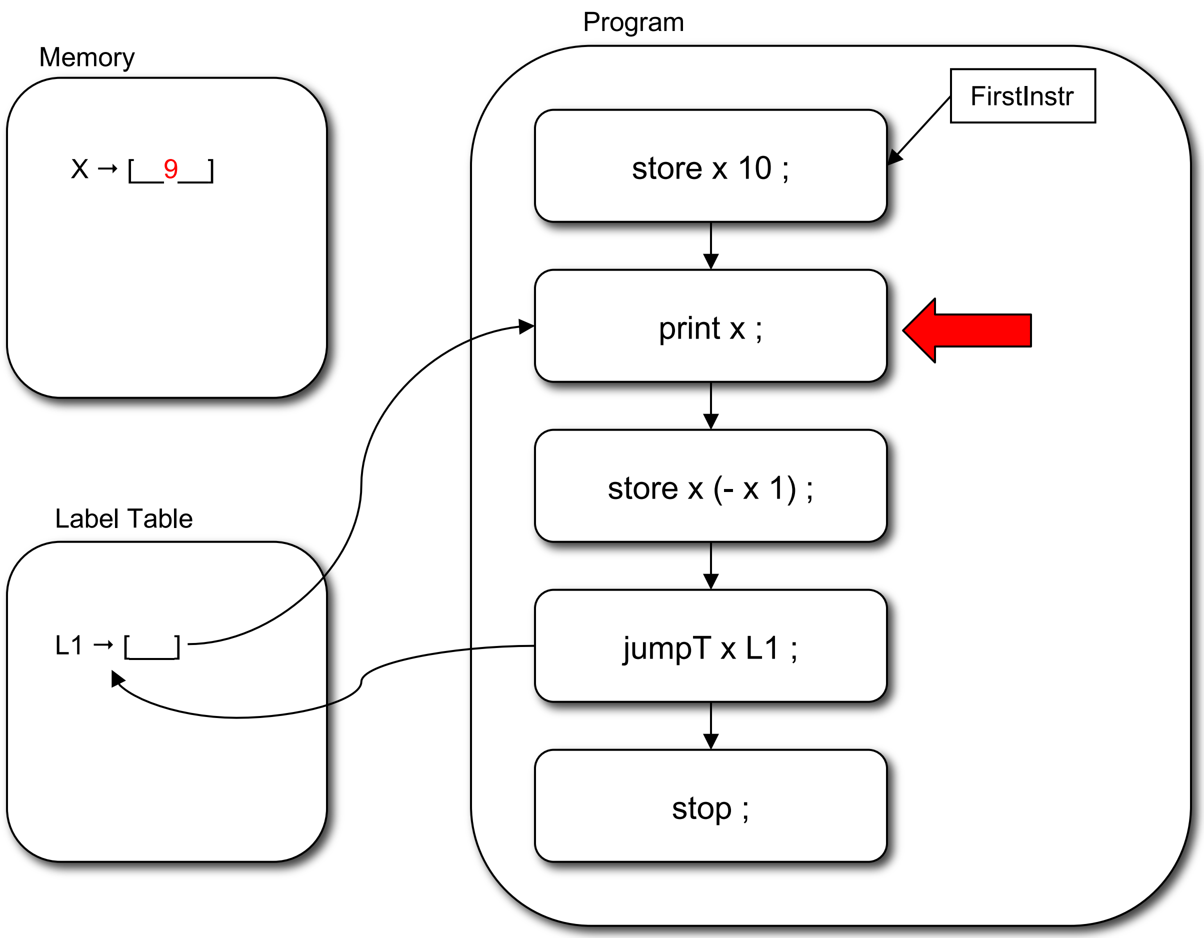
Running the Program

10



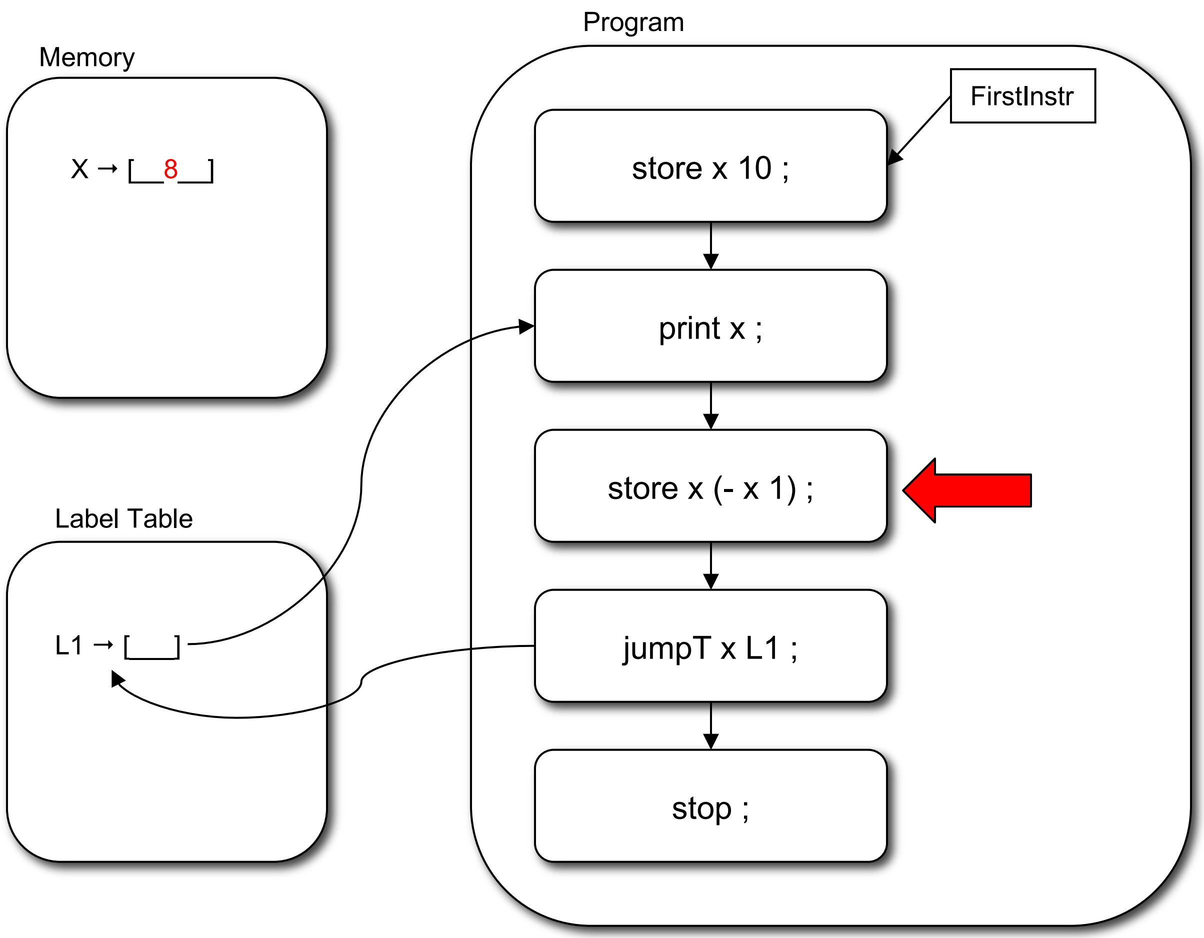
Running the Program

10 9



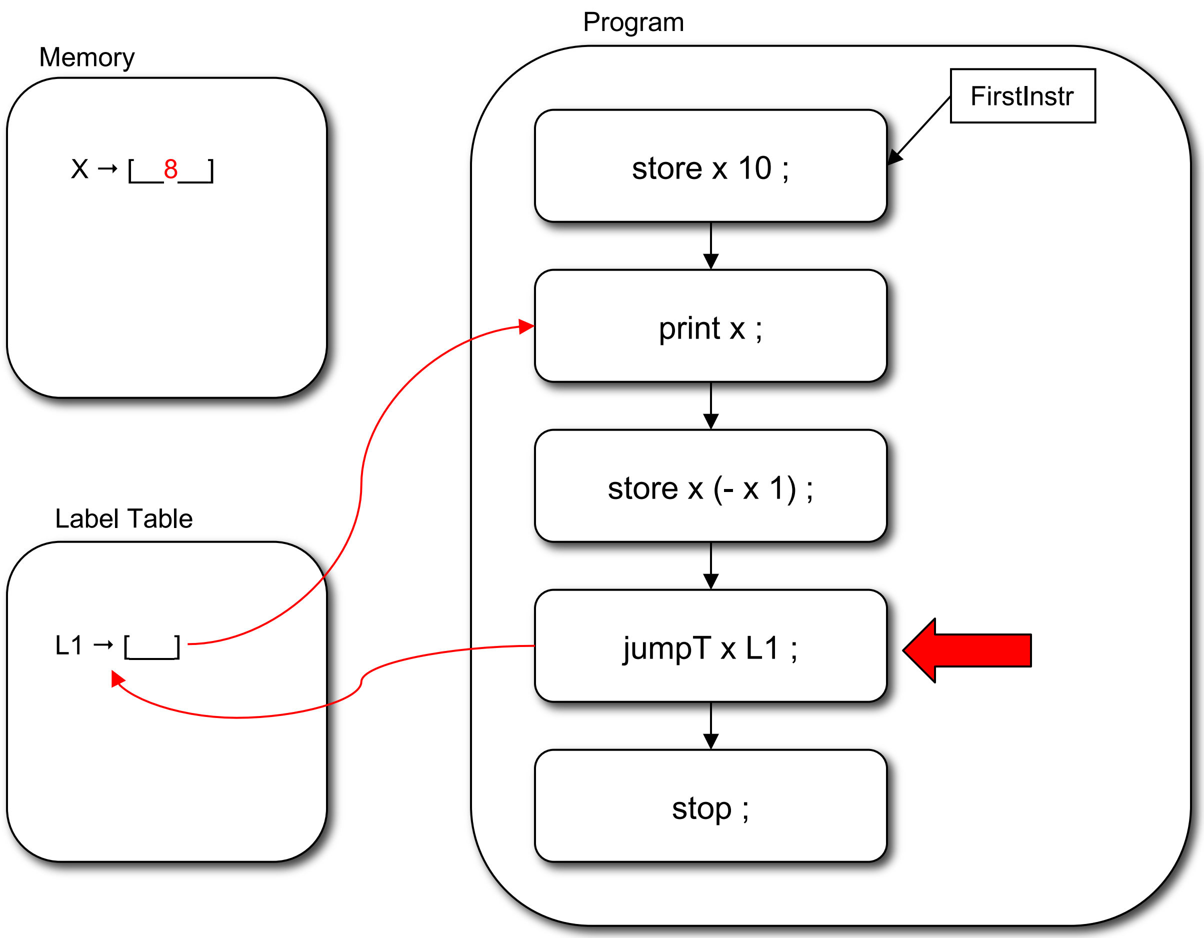
Running the Program

10 9



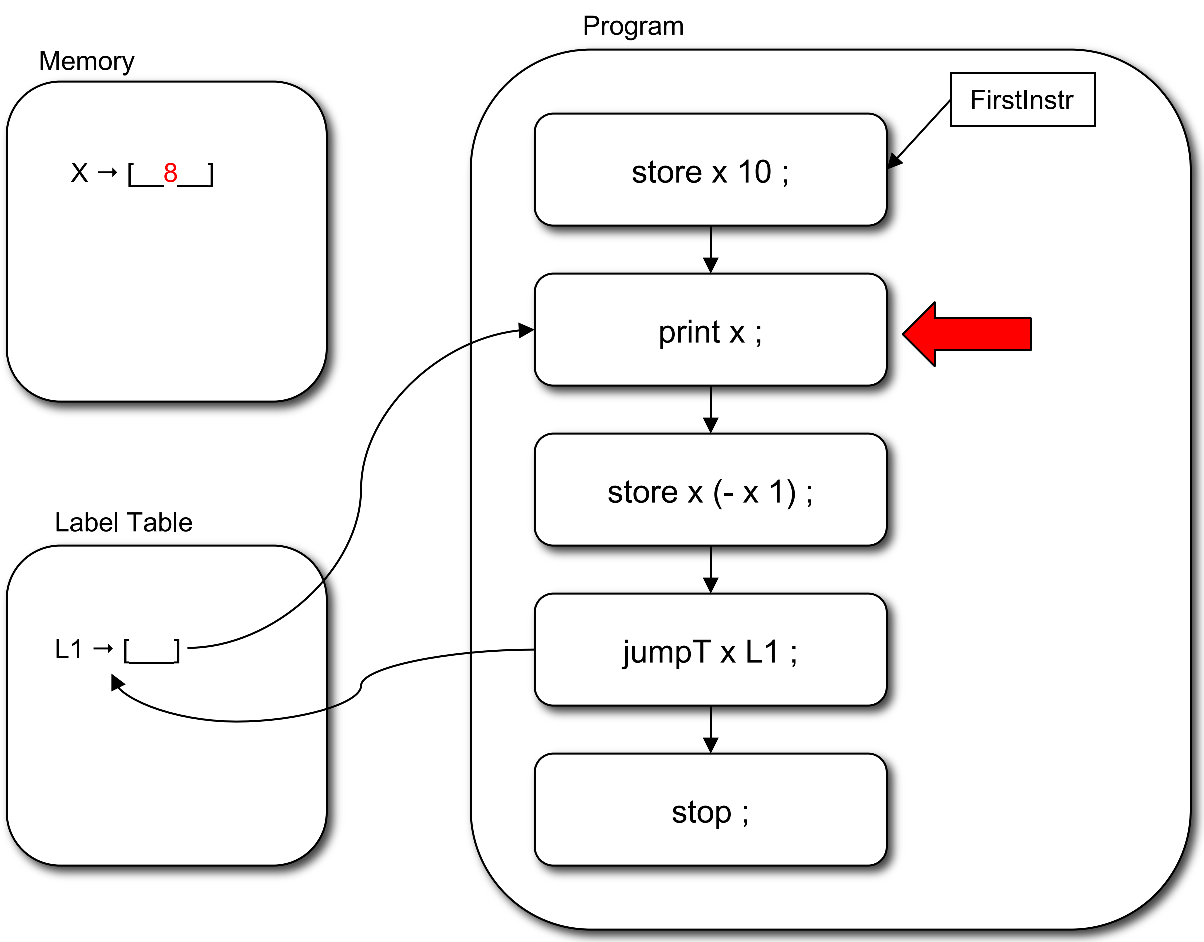
Running the Program

10 9



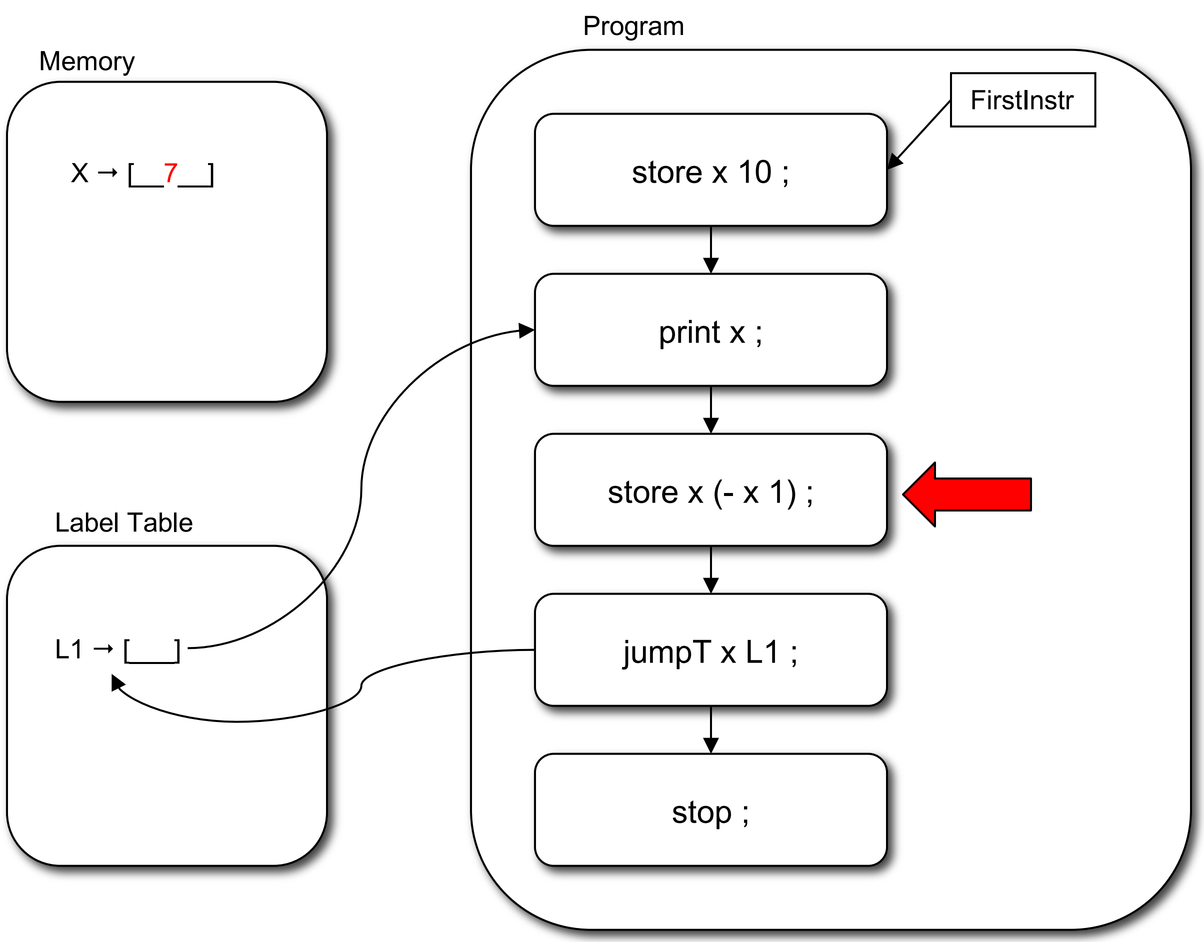
Running the Program

10 9 8



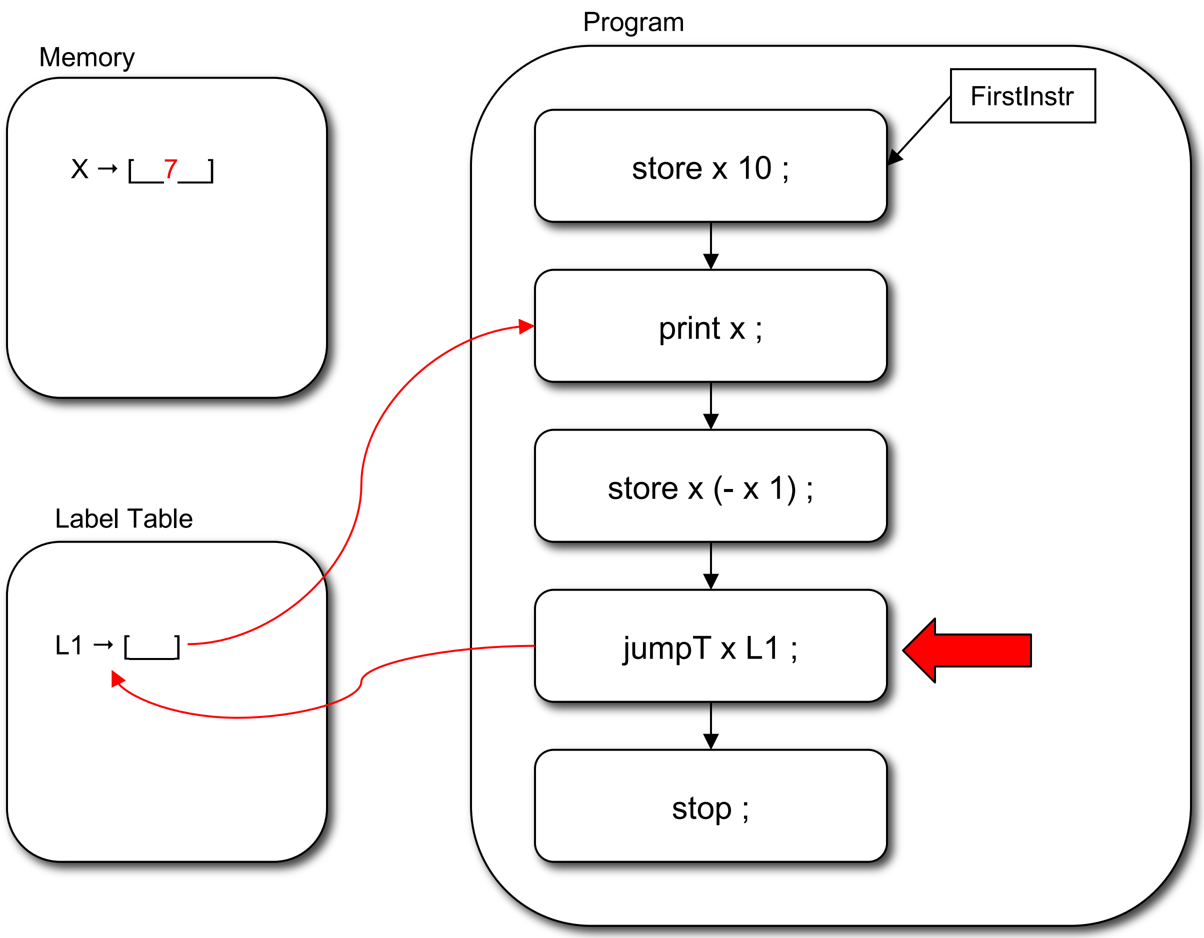
Running the Program

10 9 8



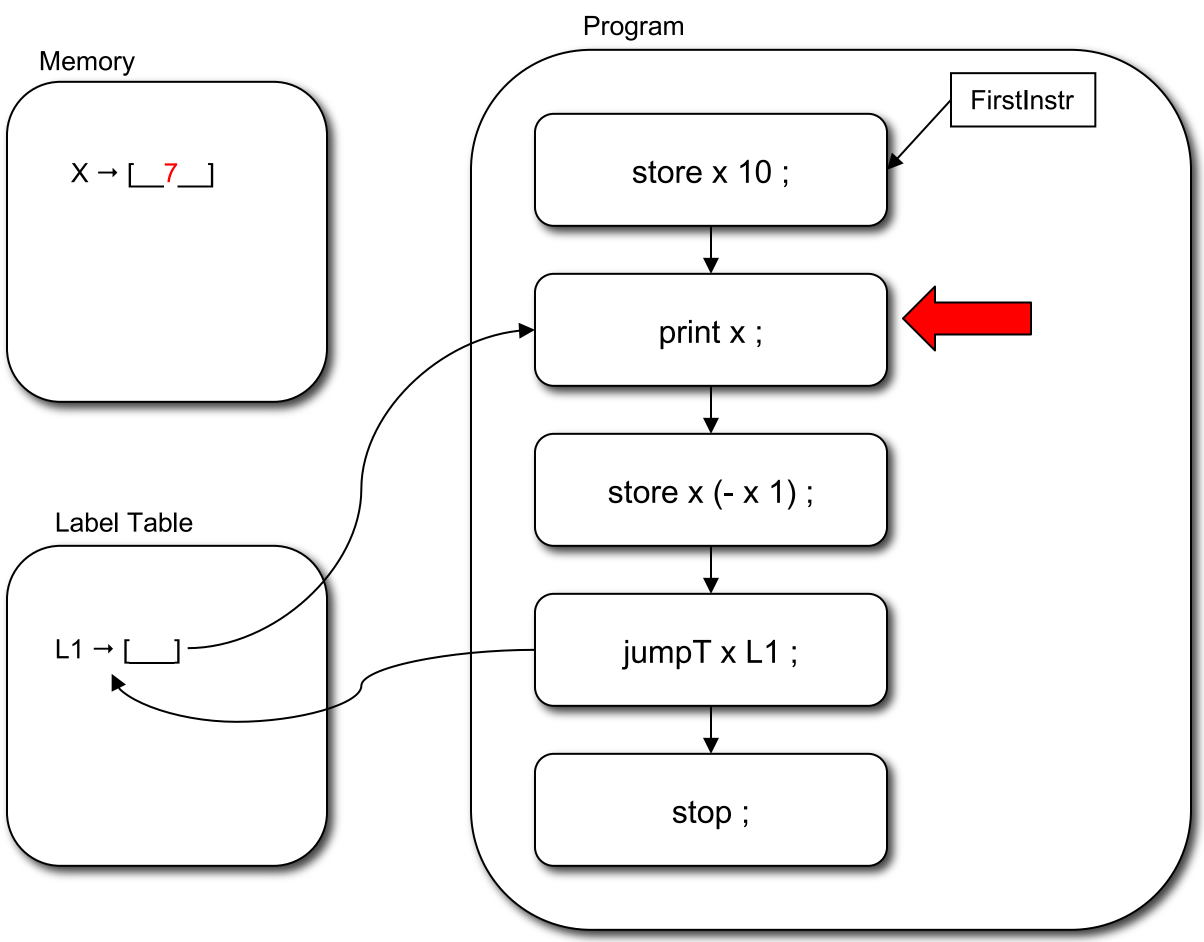
Running the Program

10 9 8



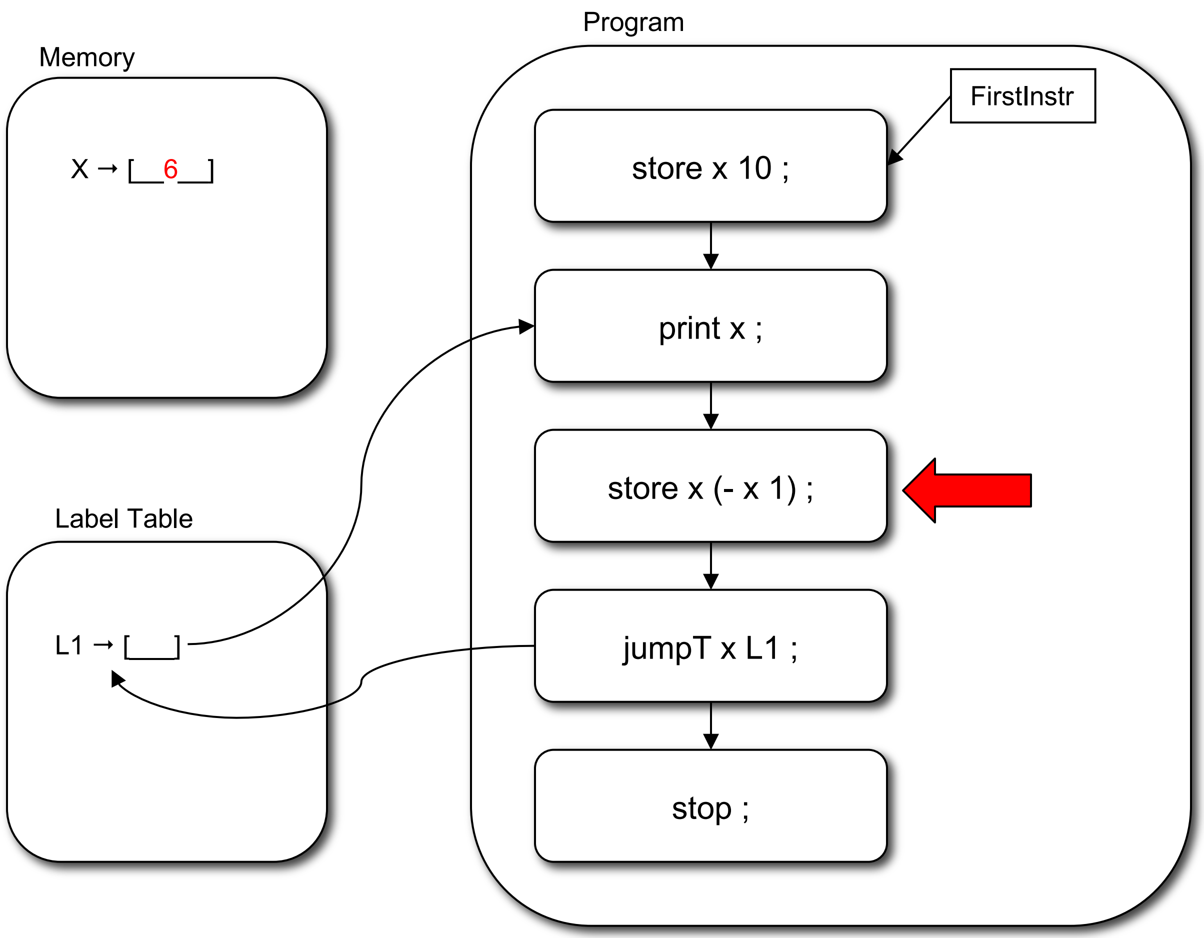
Running the Program

10 9 8



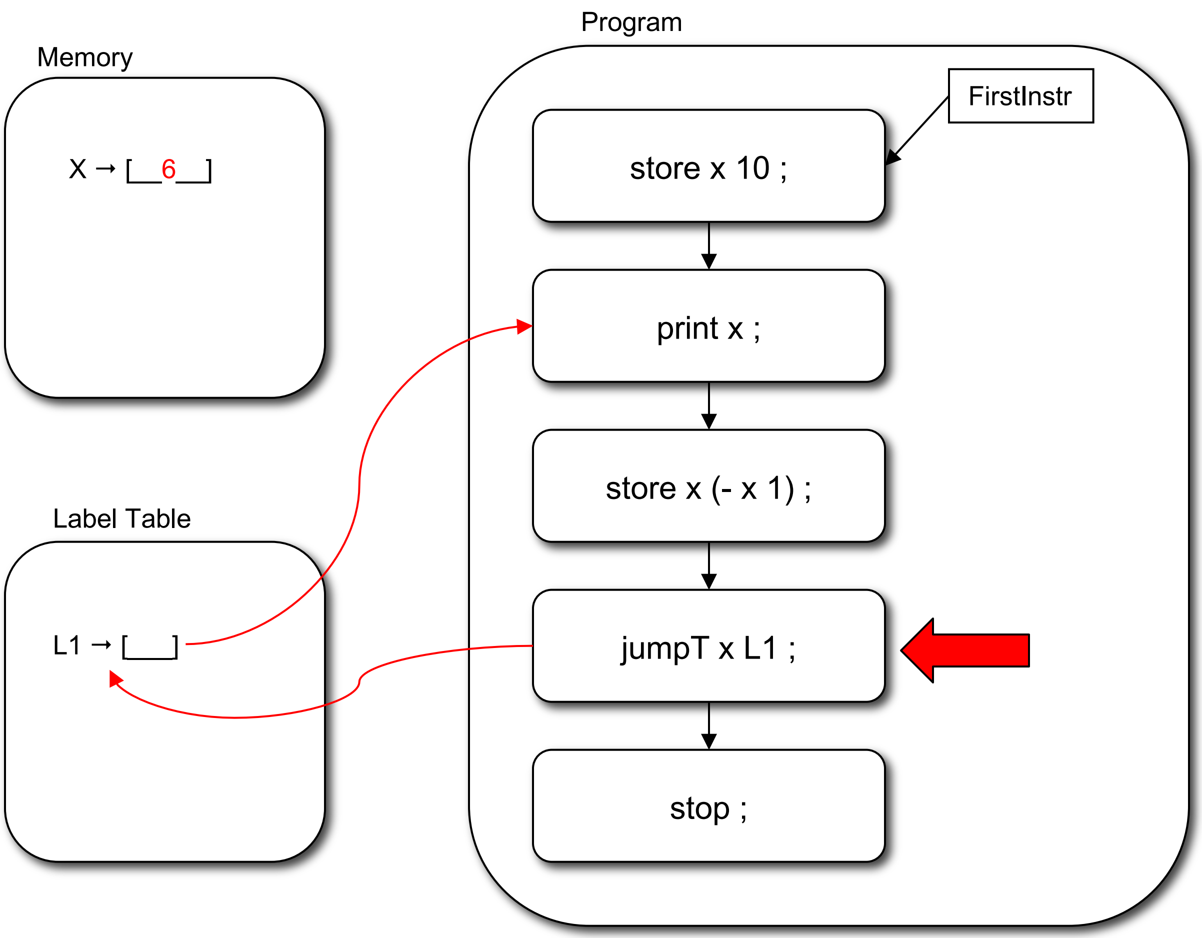
Running the Program

10 9 8 7



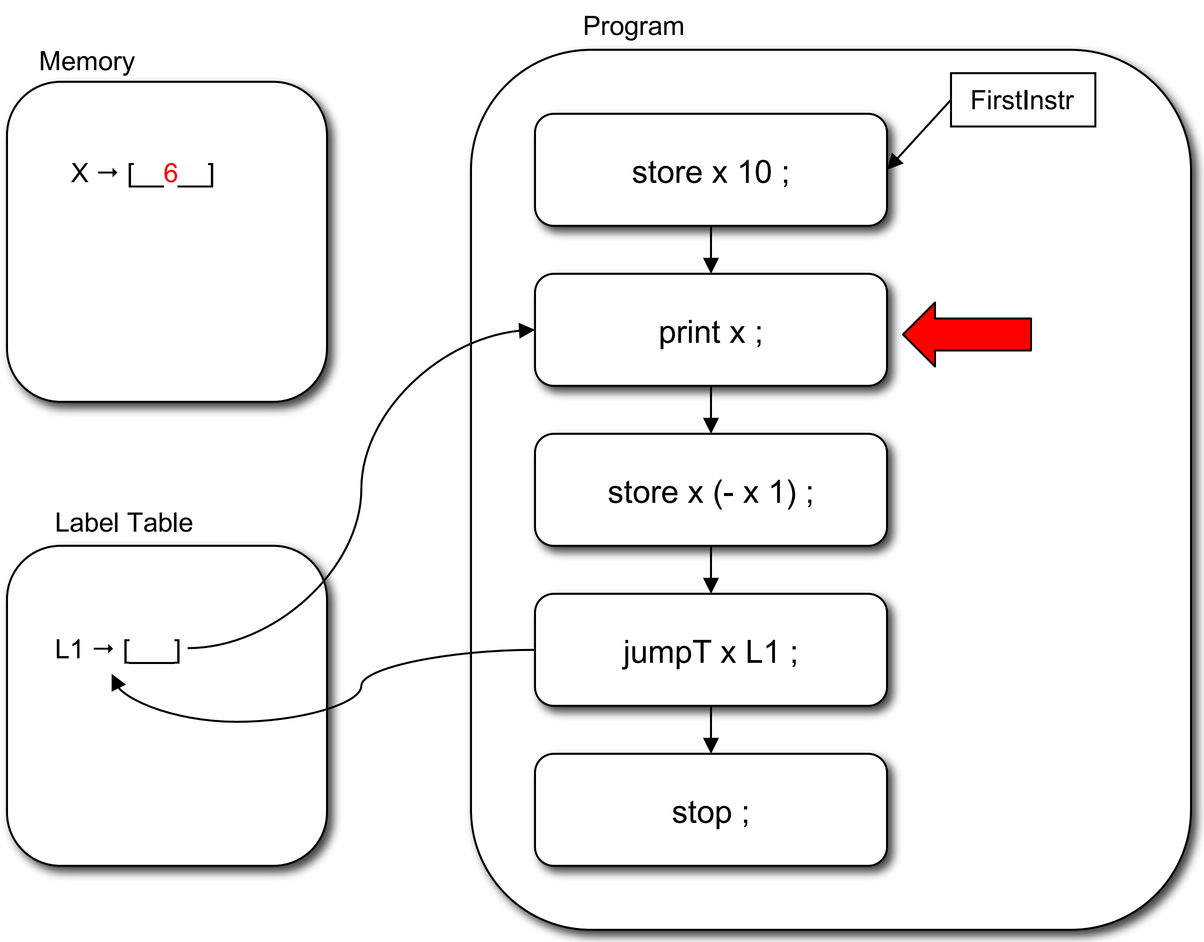
Running the Program

10 9 8 7



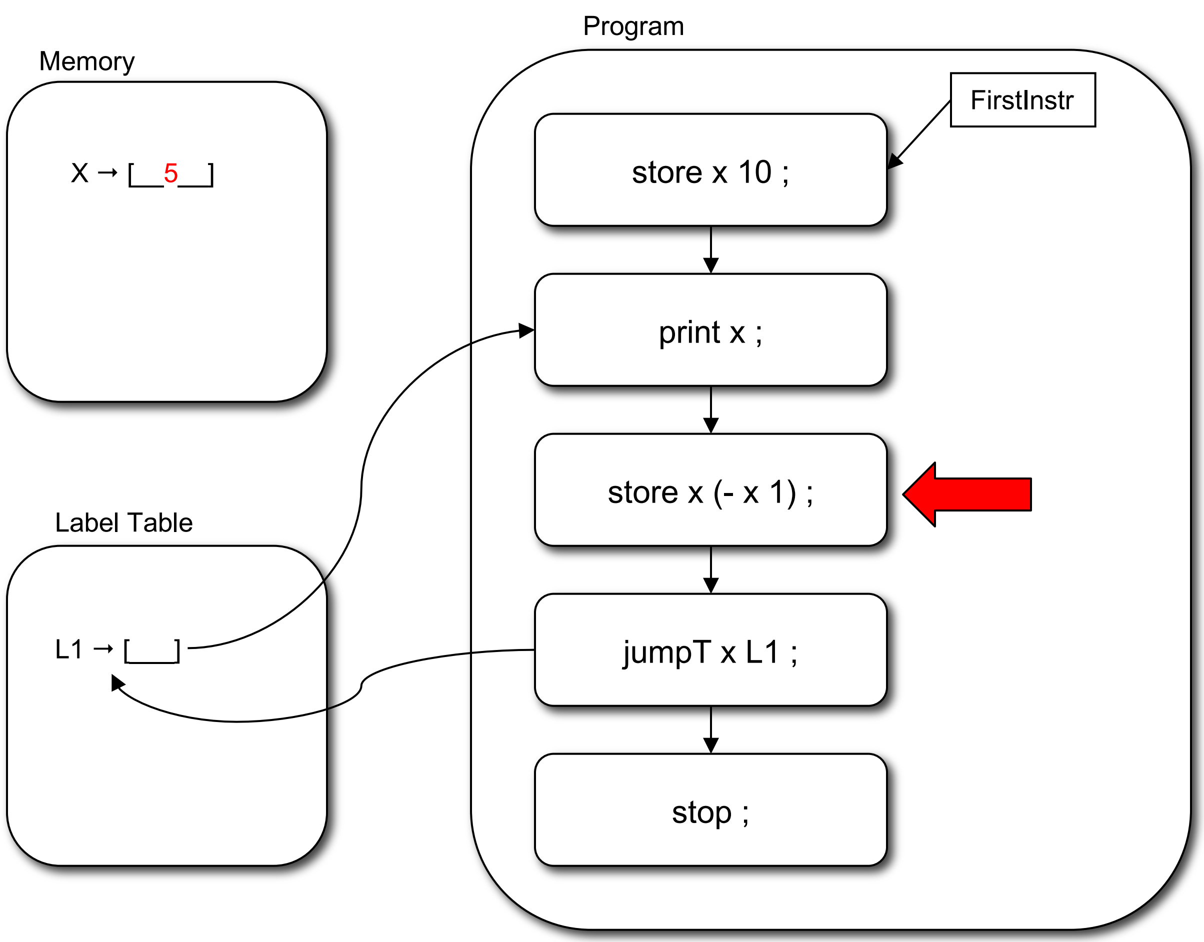
Running the Program

10 9 8 7 6



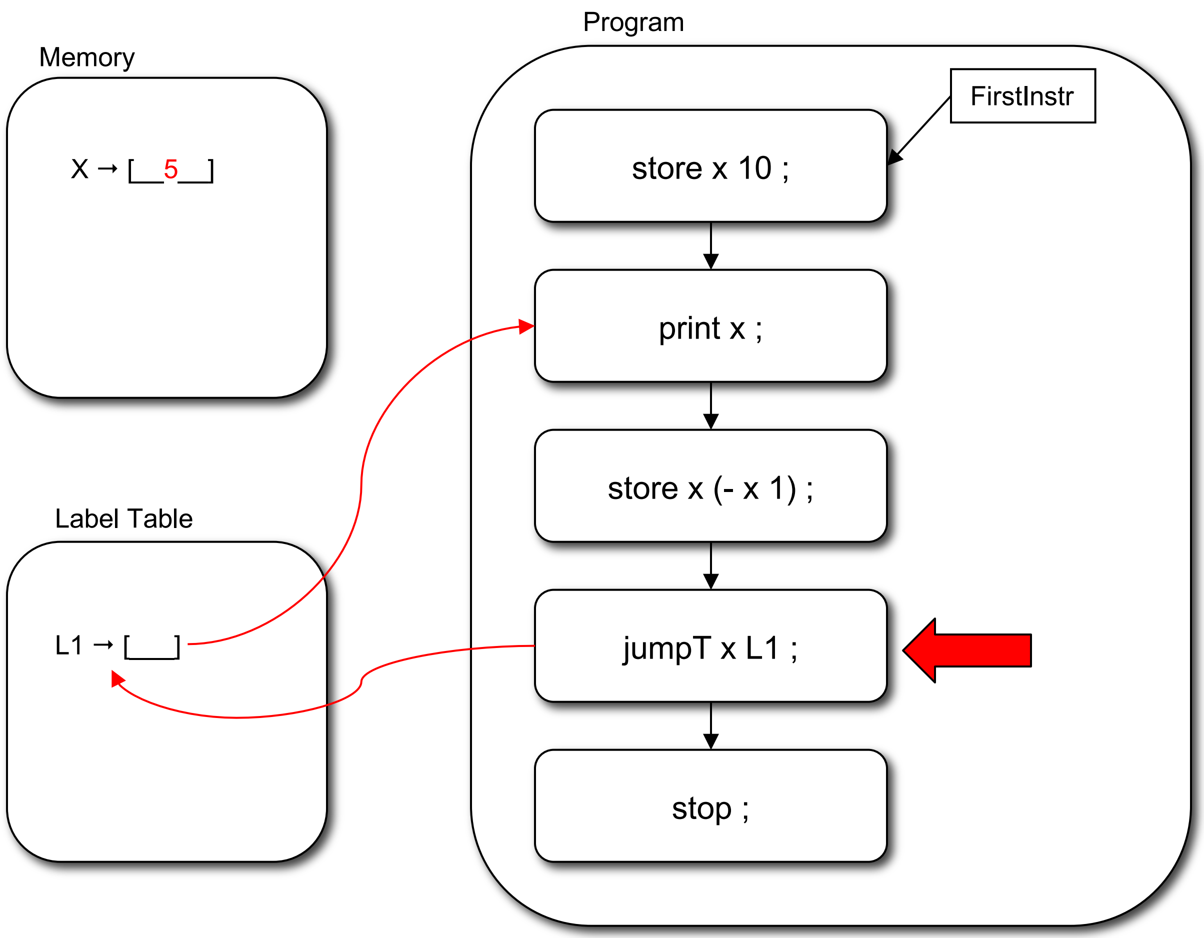
Running the Program

10 9 8 7 6



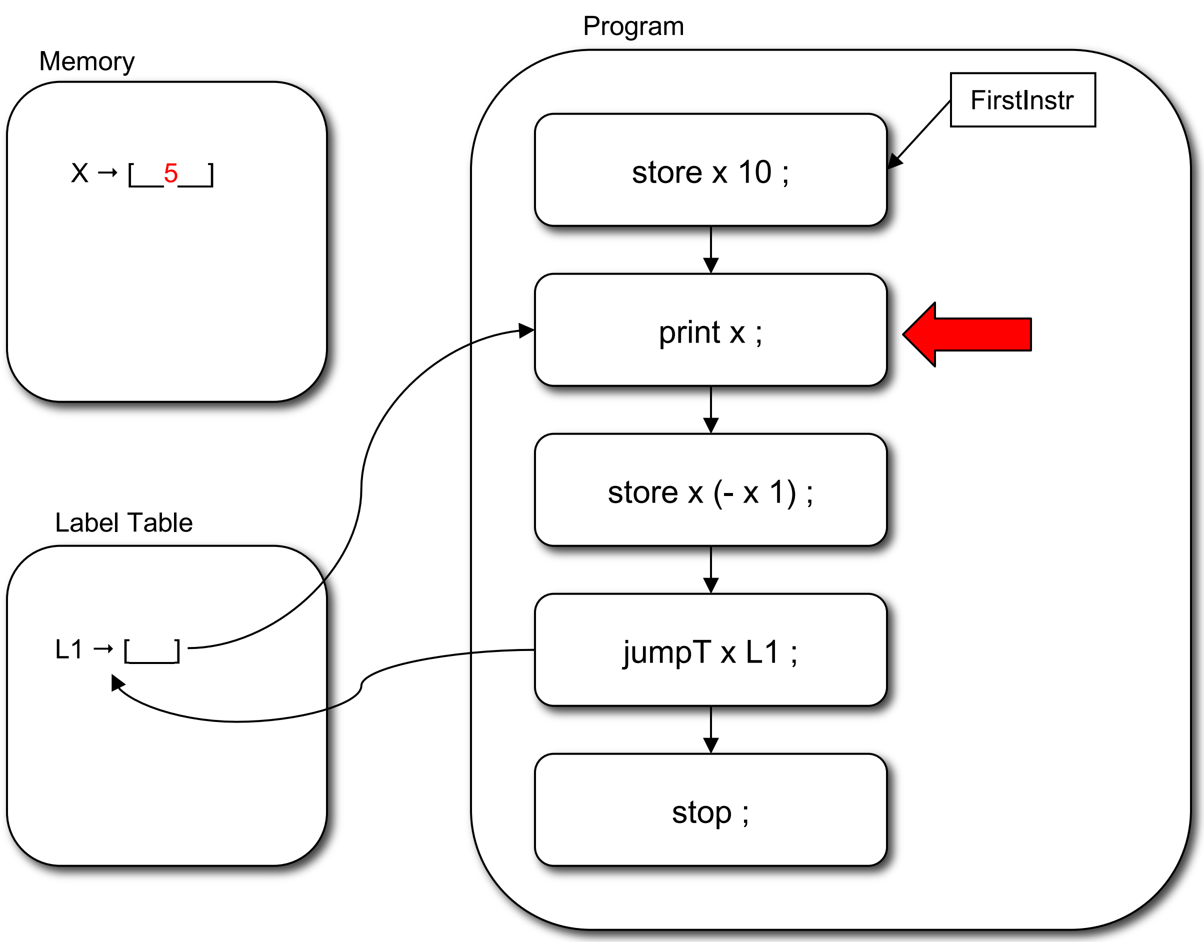
Running the Program

10 9 8 7 6



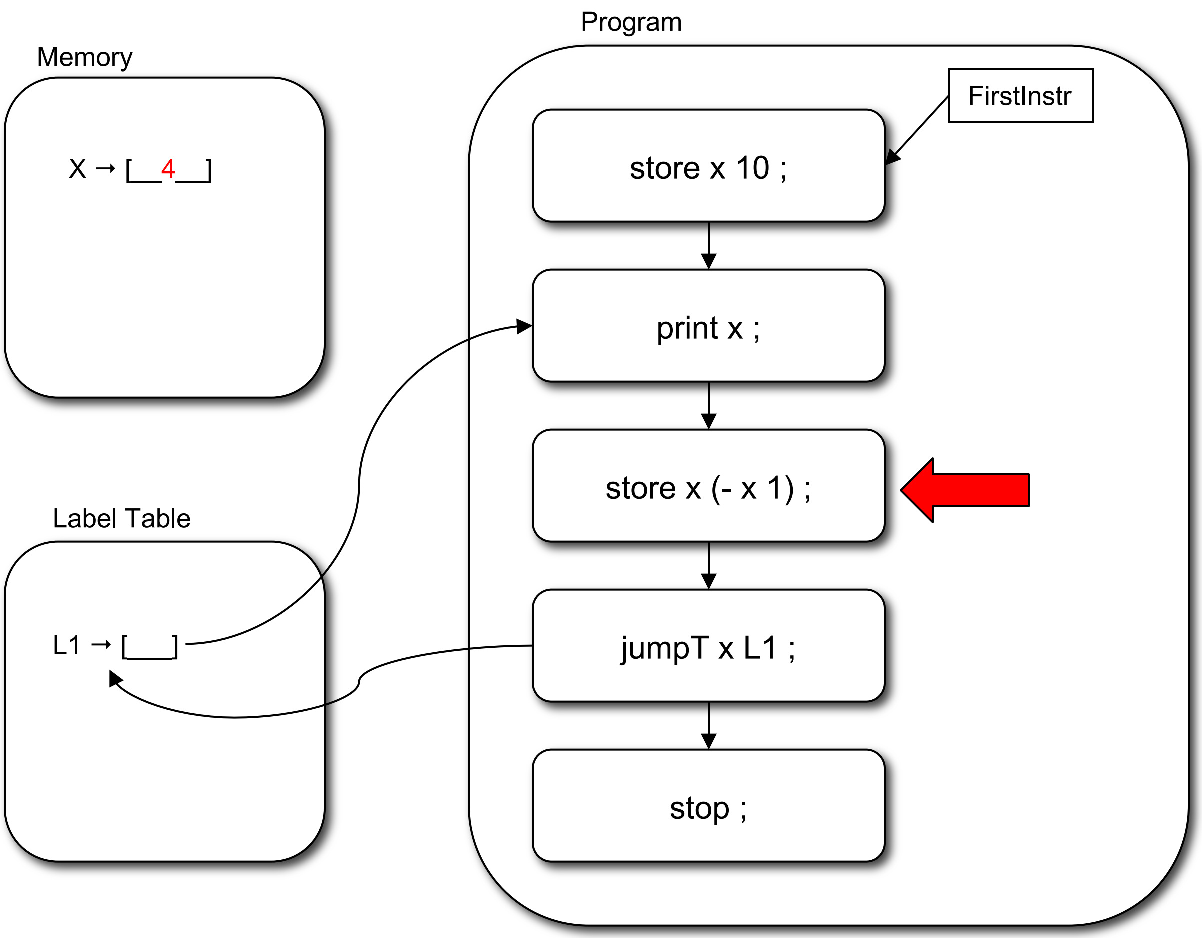
Running the Program

10 9 8 7 6 5



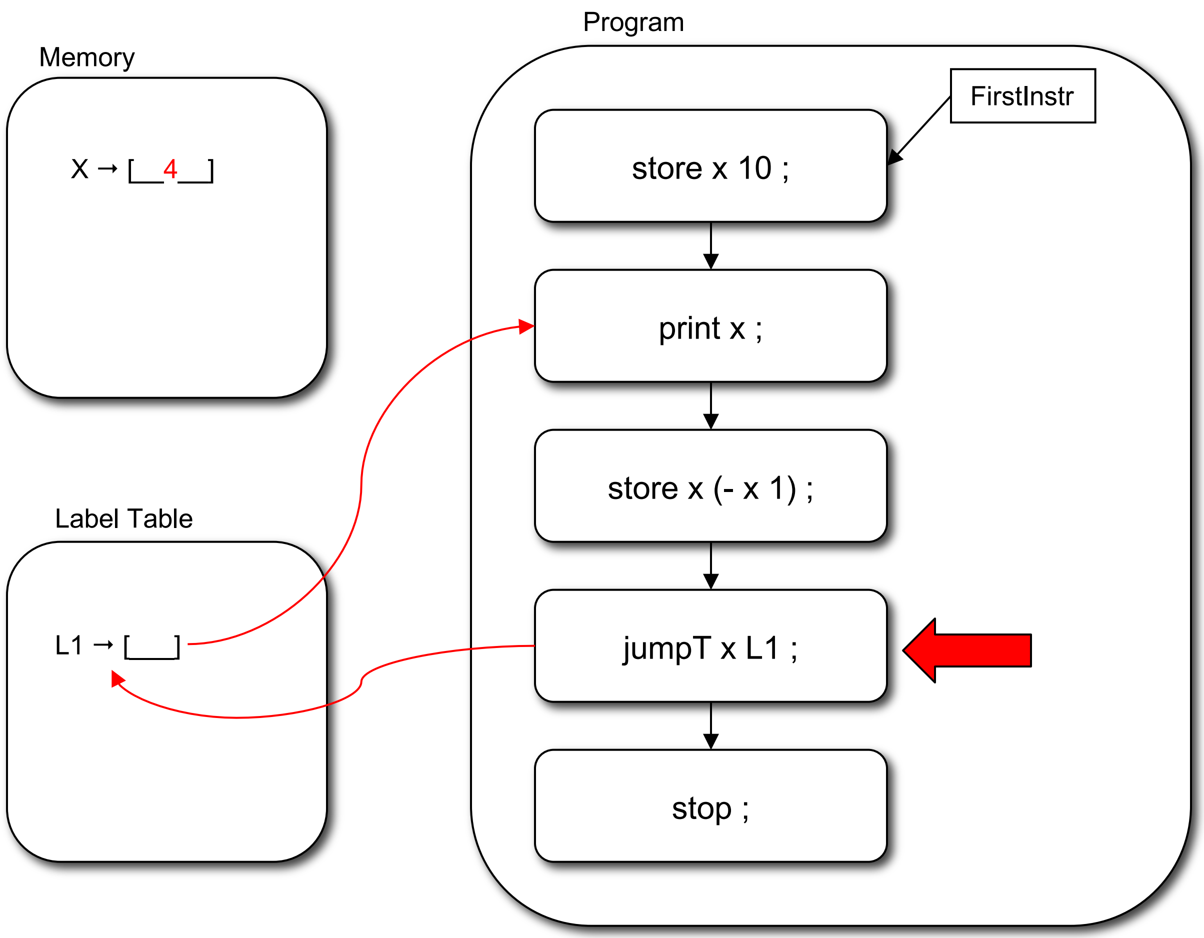
Running the Program

10 9 8 7 6 5



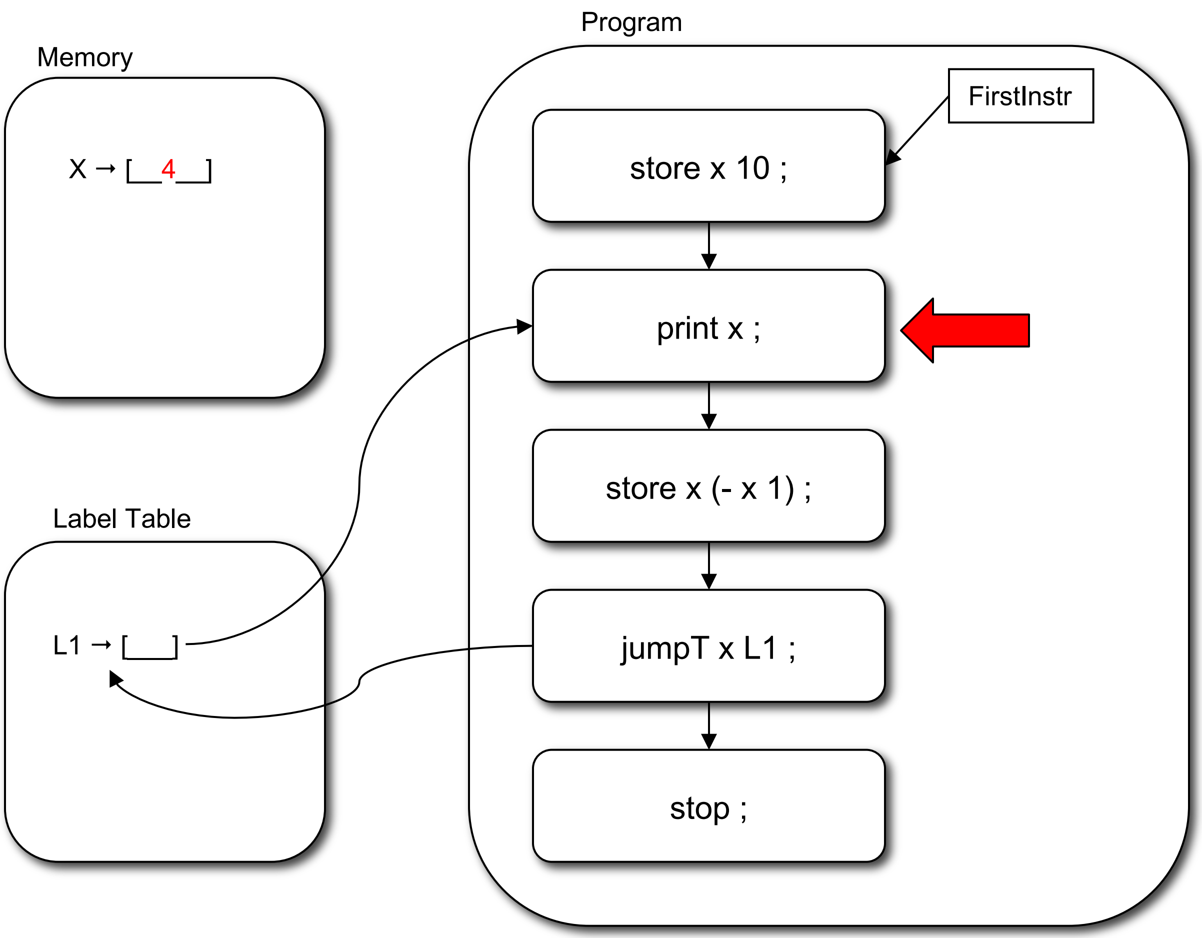
Running the Program

10 9 8 7 6 5



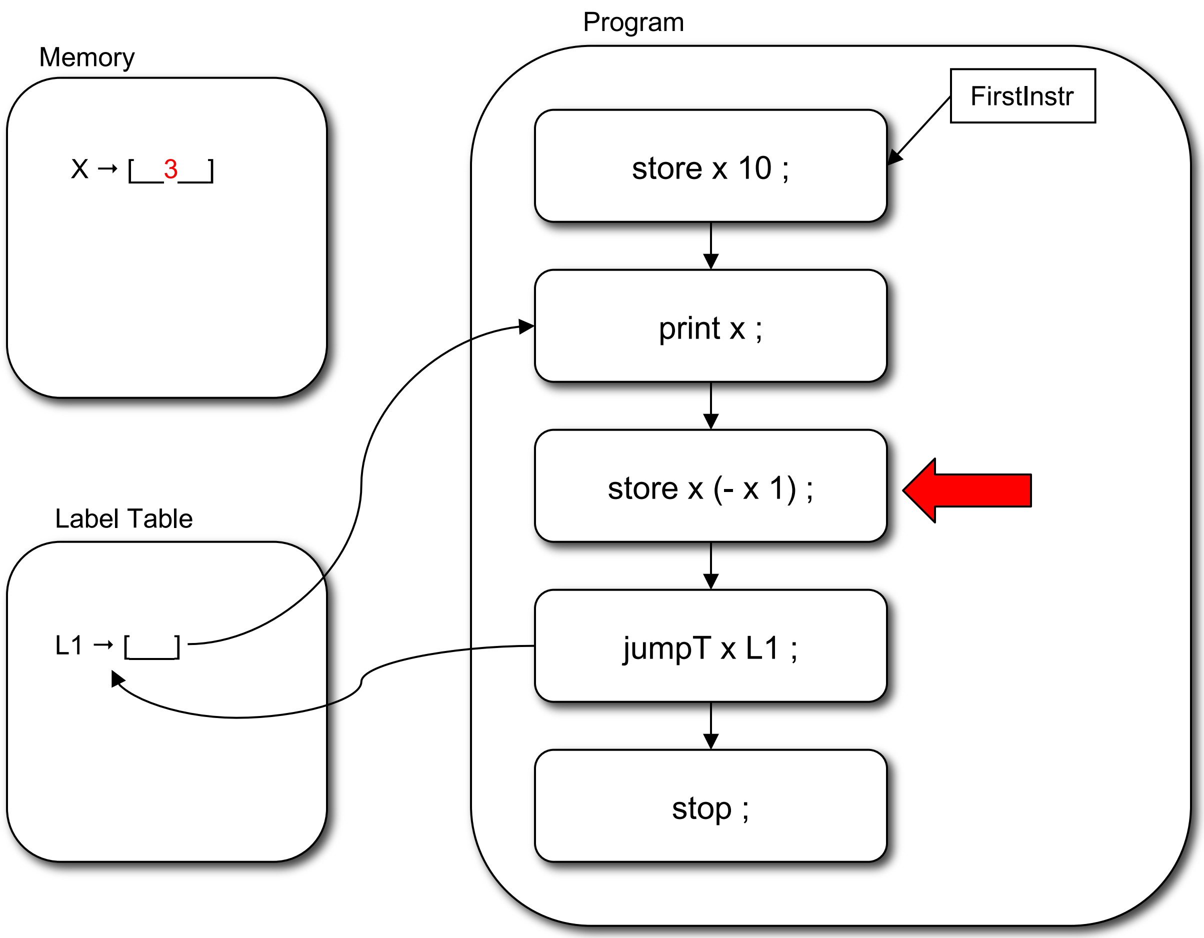
Running the Program

10 9 8 7 6 5 4



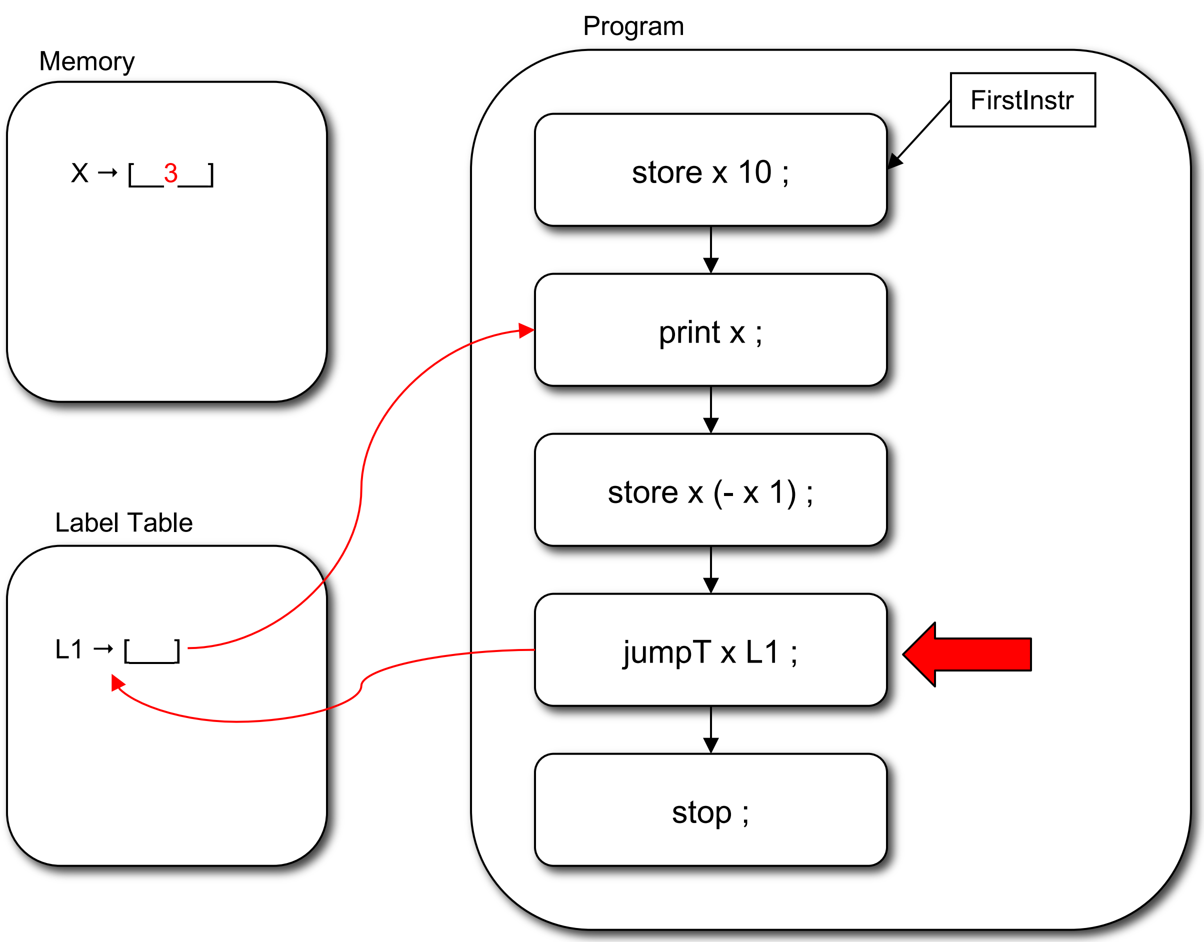
Running the Program

10 9 8 7 6 5 4



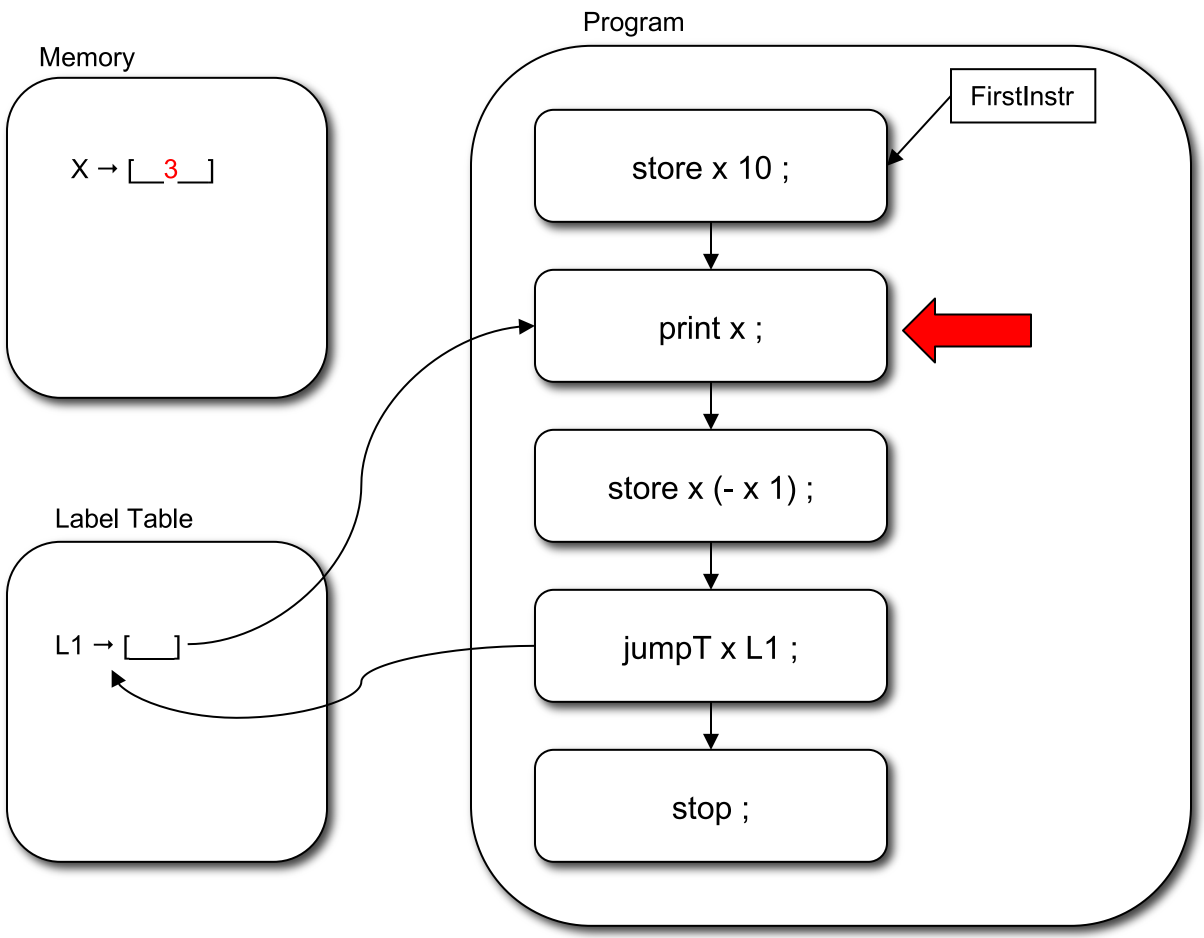
Running the Program

10 9 8 7 6 5 4



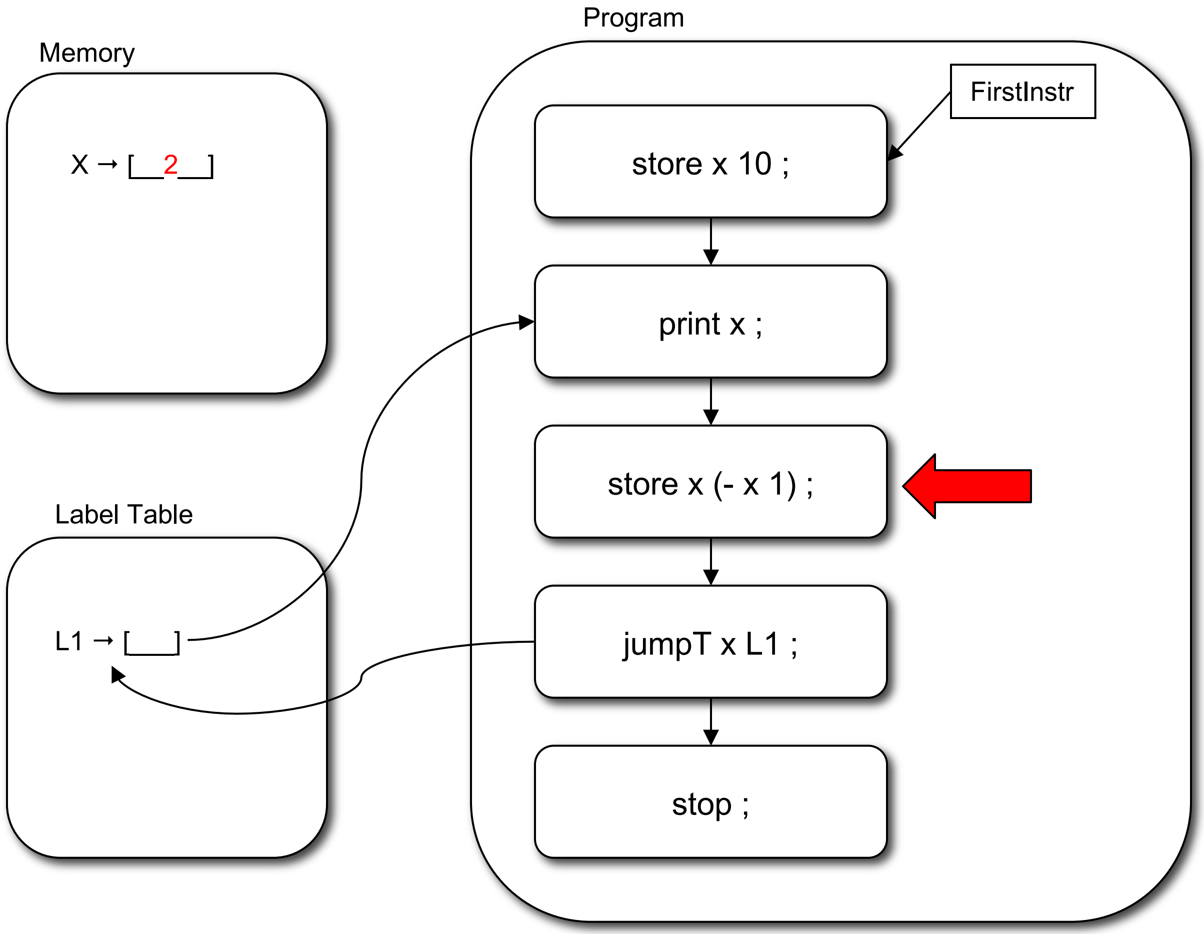
Running the Program

10 9 8 7 6 5 4 3



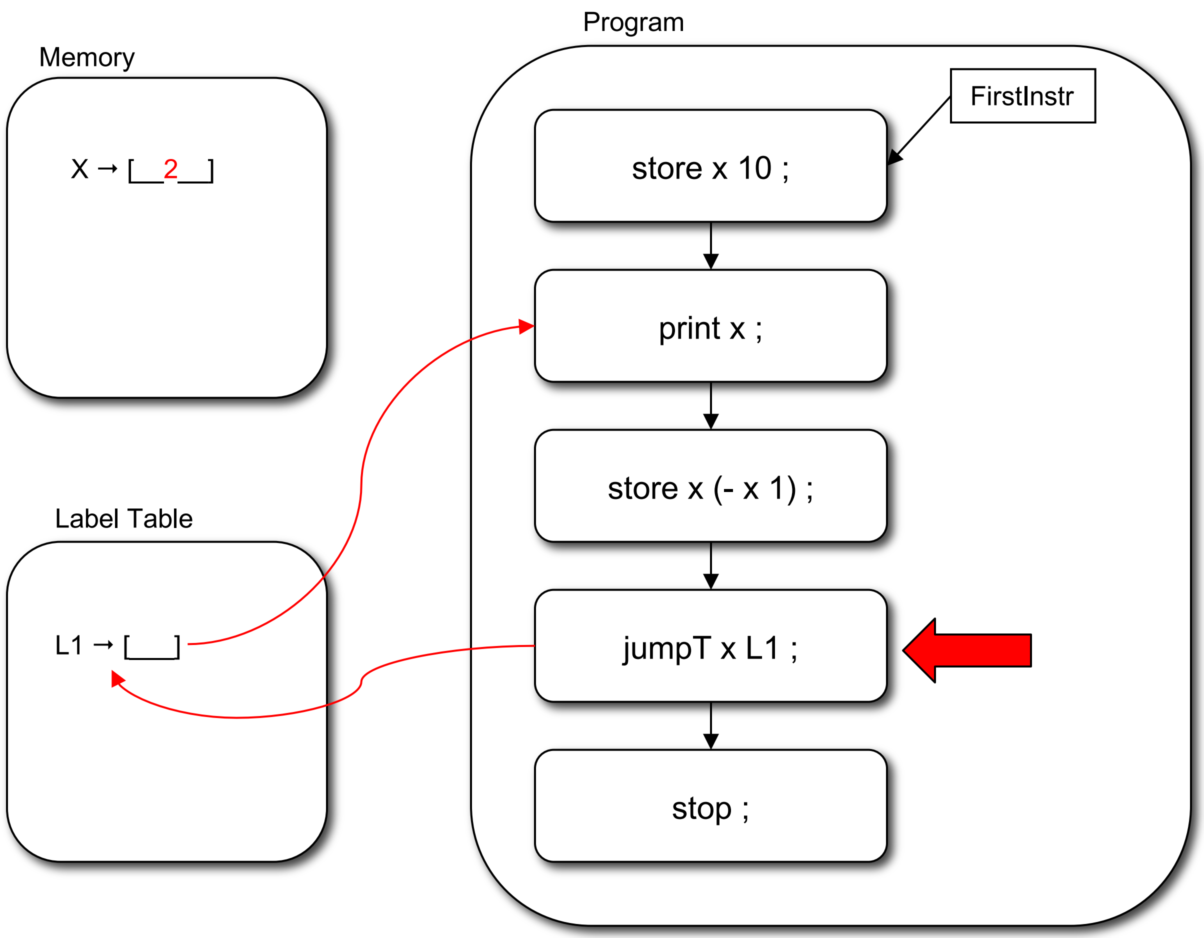
Running the Program

10 9 8 7 6 5 4 3



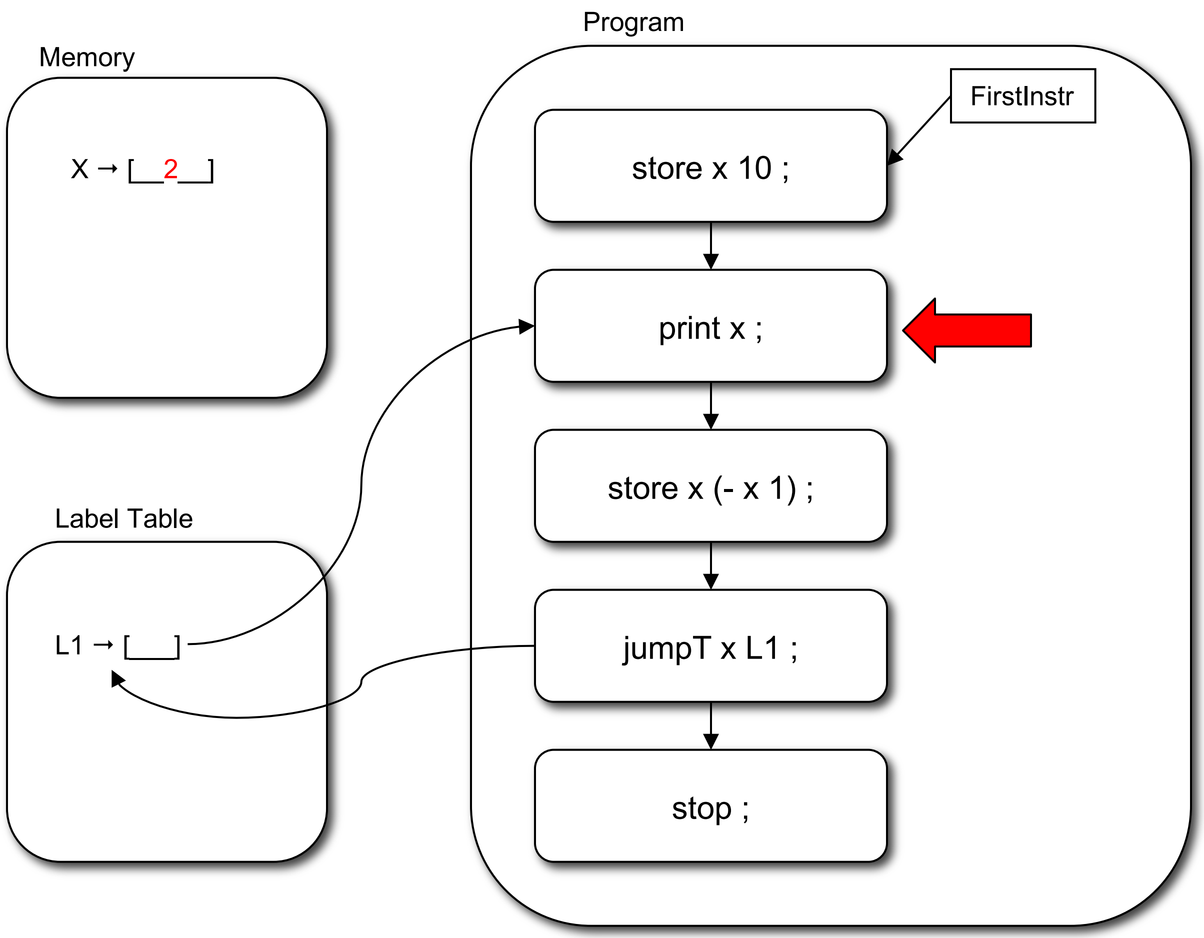
Running the Program

10 9 8 7 6 5 4 3



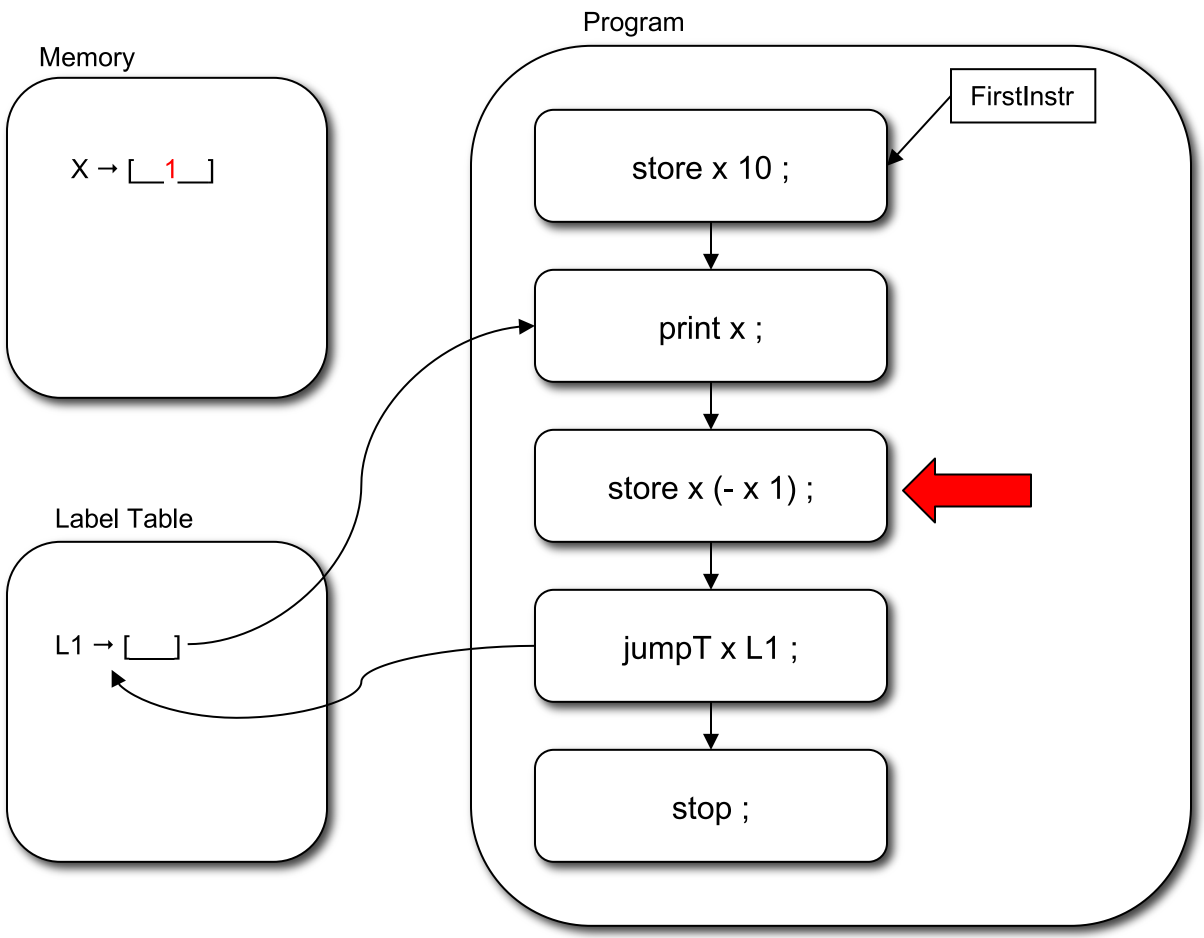
Running the Program

10 9 8 7 6 5 4 3 2



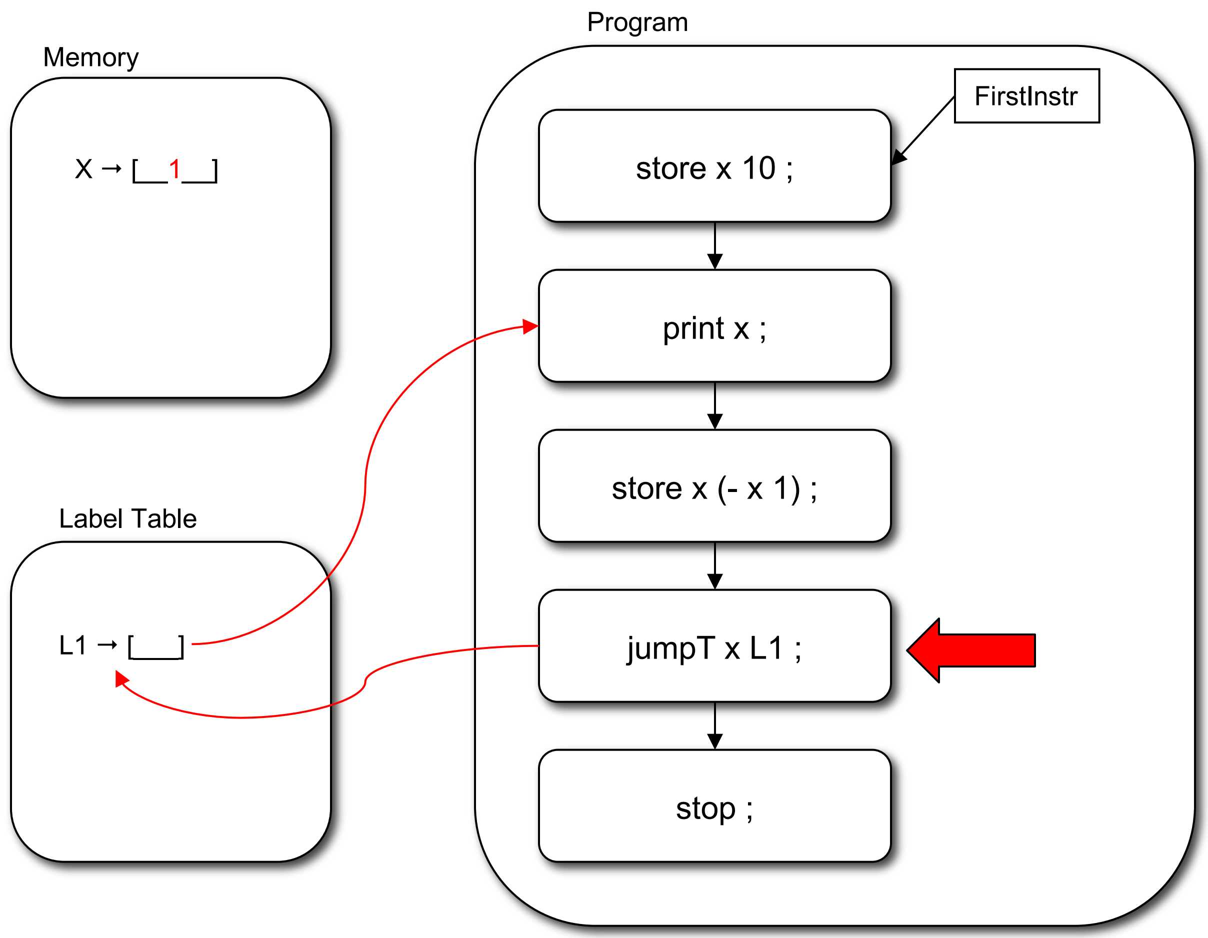
Running the Program

10 9 8 7 6 5 4 3 2



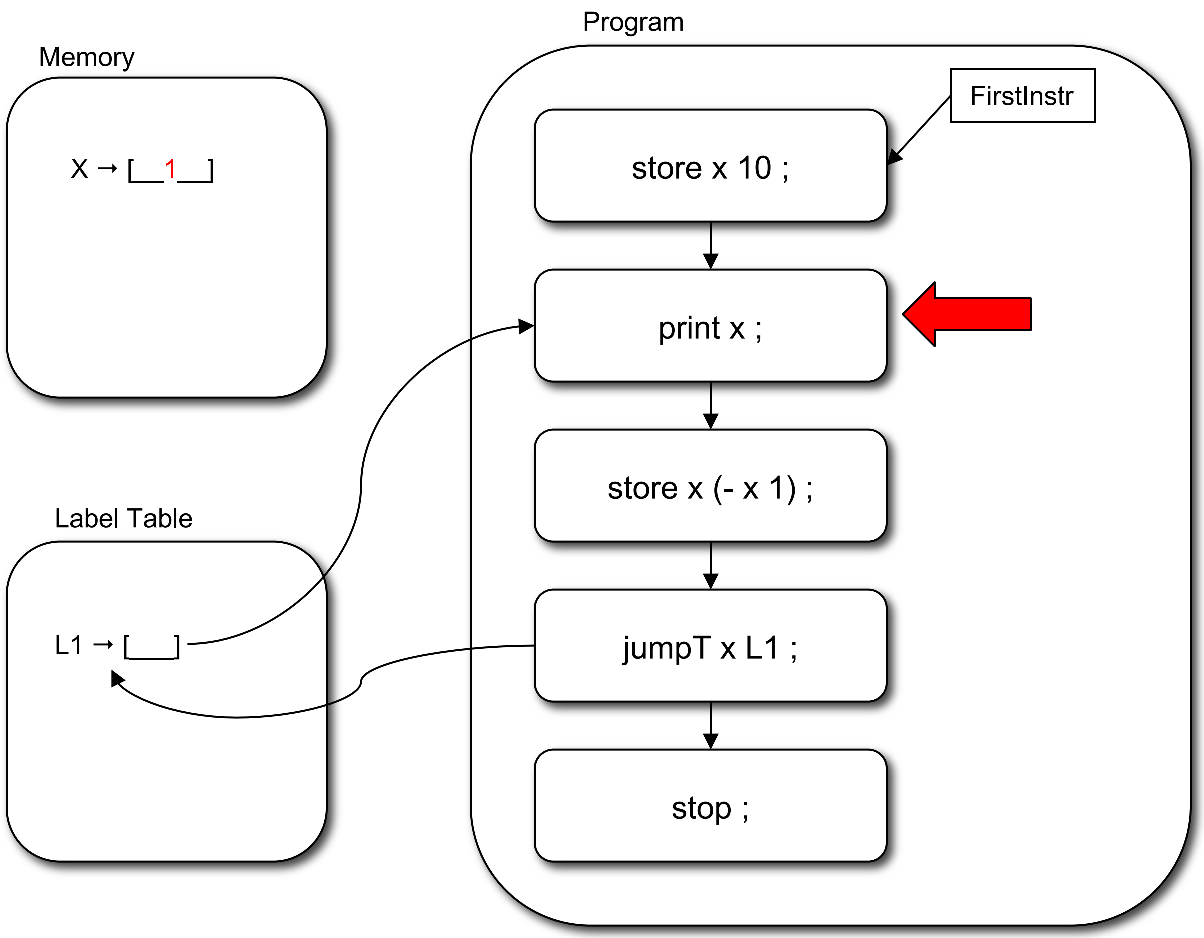
Running the Program

10 9 8 7 6 5 4 3 2



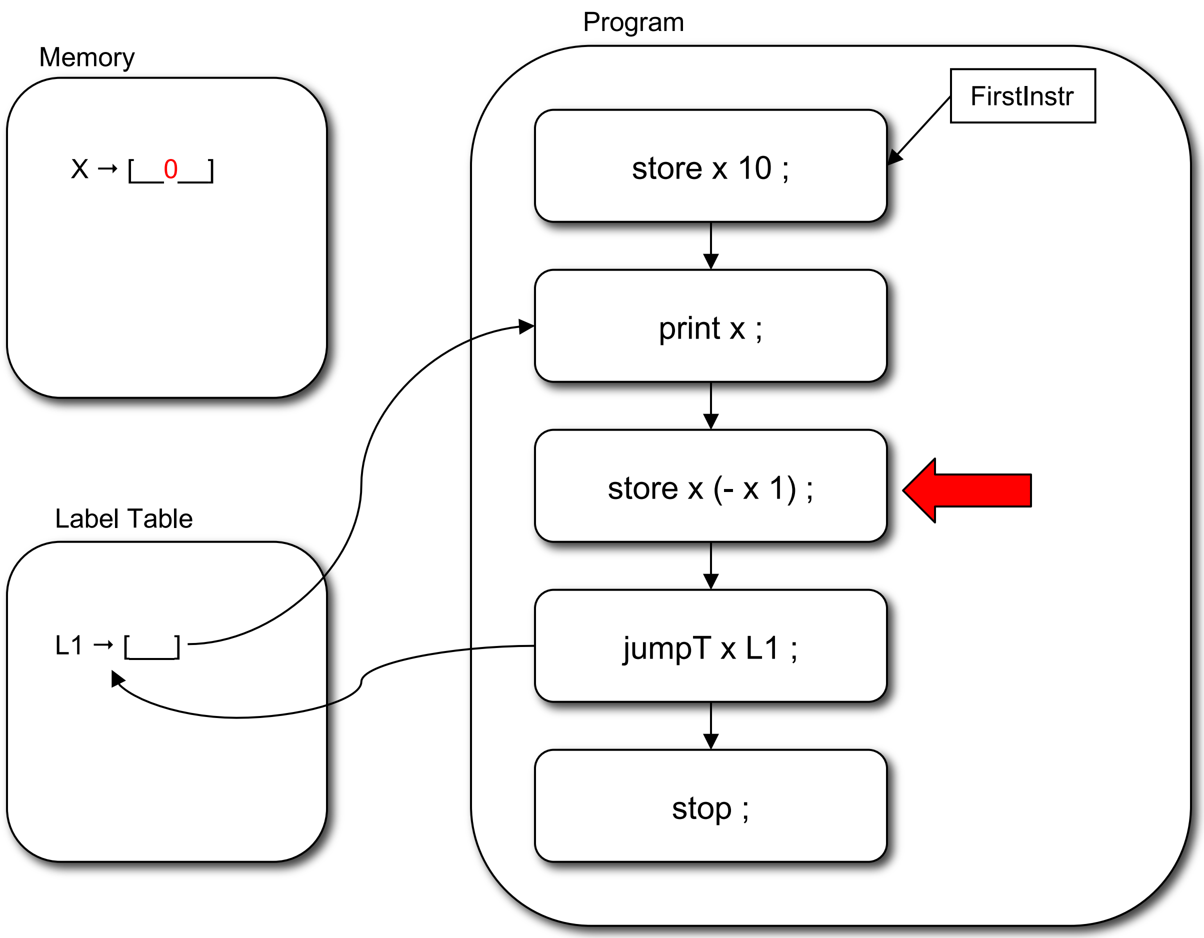
Running the Program

10 9 8 7 6 5 4 3 2 1



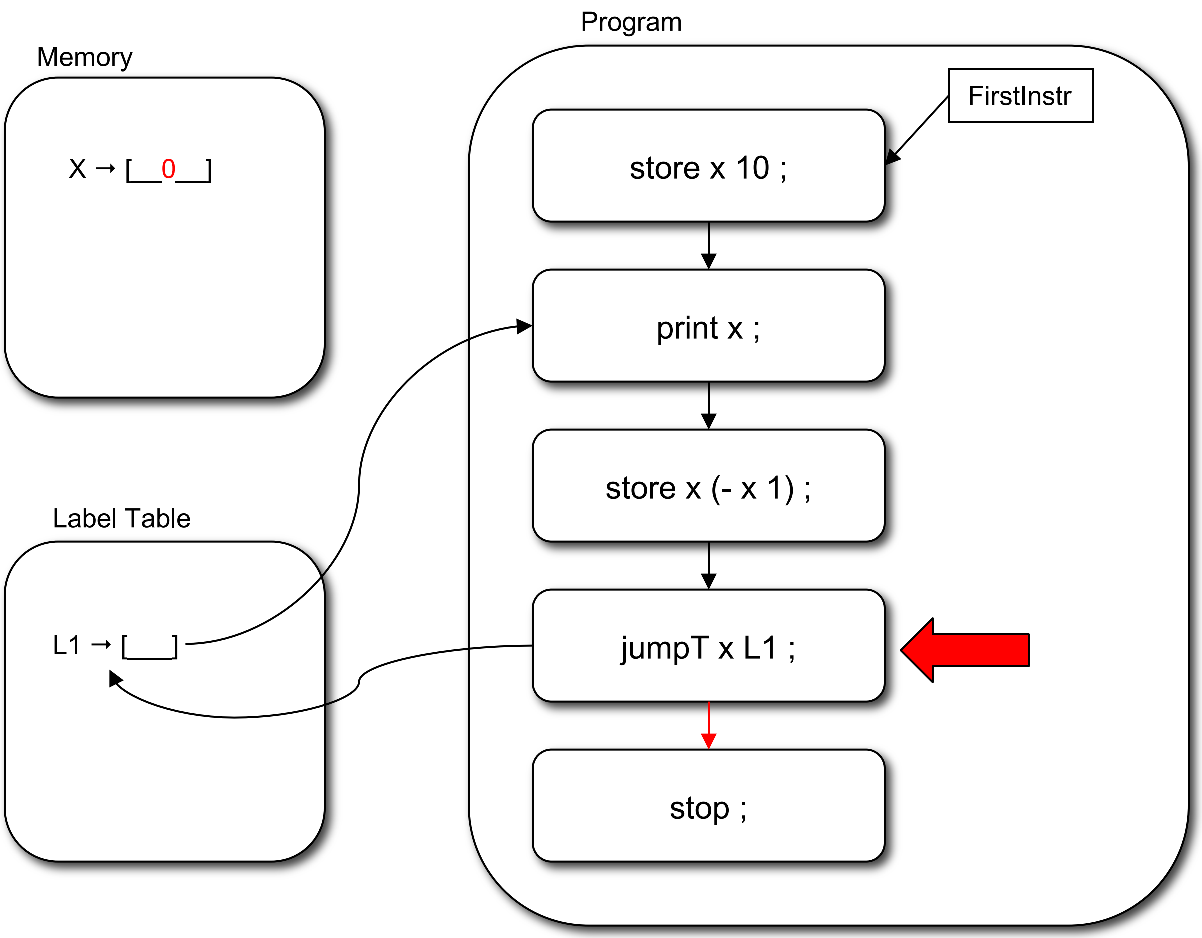
Running the Program

10 9 8 7 6 5 4 3 2 1



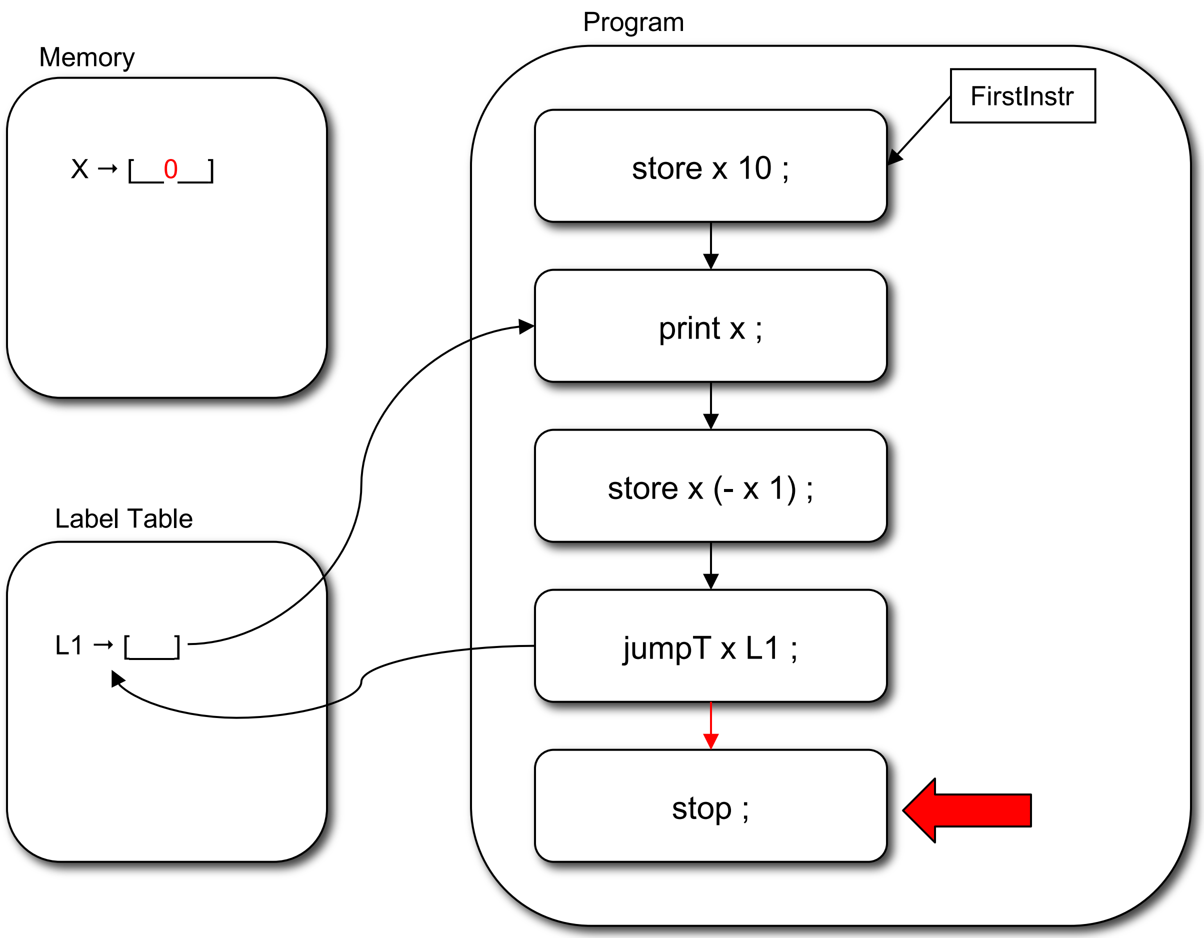
Running the Program

10 9 8 7 6 5 4 3 2 1



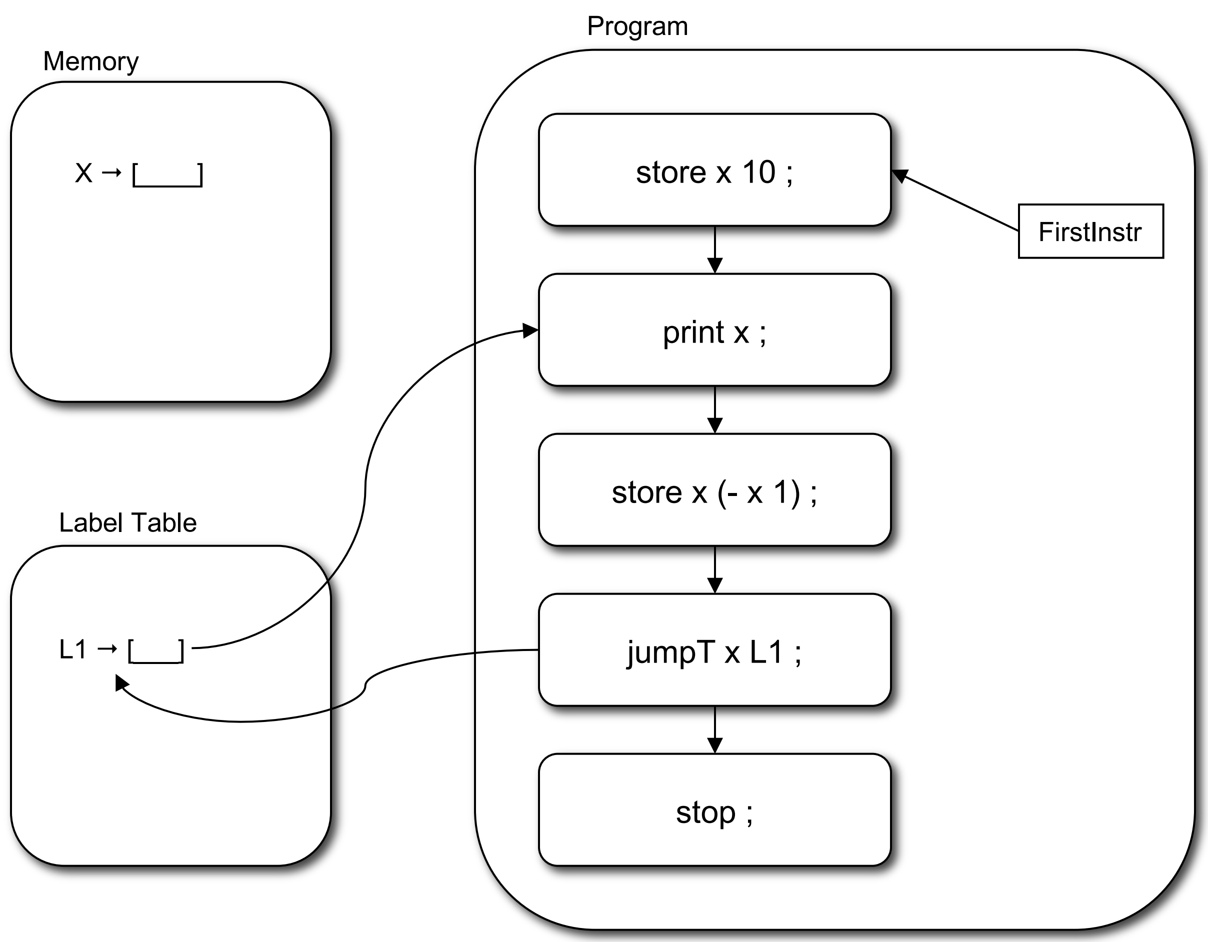
Running the Program

10 9 8 7 6 5 4 3 2 1



Implementation

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



Implementation: State

```
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: Lexer

```
token_specs = [
#   type:          value:
    ('PRINT',      r'print'),
    ('STORE',      r'store'),
    ('INPUT',      r'input'),
    ('JUMPT',      r'jumpt'),
    ('JUMPF',      r'jumpf'),
    ('JUMP',       r'jump'),
    ('STOP',       r'stop'),
    ('NOOP',       r'noop'),
    ('NUMBER',     r'[0-9]+'),
    ('NAME',       r'[a-zA-Z][a-zA-Z0-9_]*'),
    ('ADD',        r'\+'),
    ('SUB',        r'-'),
    ('MUL',        r'\*'),
    ('DIV',        r'/'),
    ('NOT',        r'!'),
    ('EQ',         r'=='),
    ('LE',         r'=<'),
    ('LPAREN',     r'\('),
    ('RPAREN',     r'\)'),
    ('SEMI',       r';'),
    ('COLON',      r':'),
    ('COMMENT',    r'#.*'),
    ('WHITESPACE', r'[ \t\n]+'),
    ('UNKNOWN',    r'.'),
]
```

Implementation: Extended Grammar

```
instr_list : ({NAME,PRINT,STORE,INPUT,JUMPT,JUMPF,JUMP,STOP,NOOP}
                   labeled_instr)*

labeled_instr : {NAME} label_def instr
              | {PRINT,STORE,INPUT,JUMPT,JUMPF,JUMP,STOP,NOOP} instr

label_def : {NAME} label COLON

instr : {PRINT} PRINT exp SEMI
      | {STORE} STORE var exp SEMI
      | {INPUT} INPUT var SEMI
      | {JUMPT} JUMPT exp label SEMI
      | {JUMPF} JUMPF exp label SEMI
      | {JUMP} JUMP label SEMI
      | {STOP} STOP SEMI
      | {NOOP} NOOP SEMI

exp : {ADD} ADD exp exp
    | {SUB} SUB exp ({ADD,SUB,MUL,DIV,NOT,EQ,LE,LPAREN,NAME,NUMBER} exp)?
    | {MUL} MUL exp exp
    | {DIV} DIV exp exp
    | {NOT} NOT exp
    | {EQ} EQ exp exp
    | {LE} LE exp exp
    | {LPAREN} LPAREN exp RPAREN
    | {NAME} var
    | {NUMBER} num

label : {NAME} NAME
var : {NAME} NAME
num : {NUMBER} NUMBER
```

Implementation: Parser

- The parser has code that will construct/fill in the IR

exp1bytecode_interp_fe.py

Implementation: Parser

```
# lookahead sets for parser
exp_lookahead = ['ADD','SUB','MUL','DIV','NOT','EQ','LE','LPAREN','NAME','NUMBER']
instr_lookahead = ['PRINT','STORE','INPUT','JUMPT','JUMPF','JUMP','STOP','NOOP']
labeled_instr_lookahead = instr_lookahead + ['NAME']
```

```
# instr_list : ({NAME,PRINT,STORE,JUMPT,JUMPF,JUMP,STOP,NOOP} labeled_instr)*
def instr_list(stream):
  while stream.pointer().type in labeled_instr_lookahead:
    labeled_instr(stream)
  return None
```

Implementation: Parser

```
# labeled_instr : {NAME} label_def instr
#               | {PRINT,STORE,JUMPT,JUMPF,JUMP,STOP,NOOP} instr
def labeled_instr(stream):
    token = stream.pointer()
    if token.type in ['NAME']:
        l = label_def(stream)
        i = instr(stream)
        state.label_table[l] = state.instr_ix
        state.program.append(i)
        state.instr_ix += 1
        return None
    elif token.type in instr_lookahead:
        i = instr(stream)
        state.program.append(i)
        state.instr_ix += 1
        return None
    else:
        raise SyntaxError("labeled_instr: syntax error at {}"
                          .format(token.value))
```

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

Implementation: Parser

```
def instr(stream):
    token = stream.pointer()
    if token.type in ['PRINT']:
        stream.match('PRINT')
        e = exp(stream)
        stream.match('SEMI')
        return ('PRINT', e)
    elif token.type in ['STORE']:
        stream.match('STORE')
        v = var(stream)
        e = exp(stream)
        stream.match('SEMI')
        return ('STORE', v, e)
...
    elif token.type in ['JUMPT']:
        stream.match('JUMPT')
        e = exp(stream)
        l = label(stream)
        stream.match('SEMI')
        return ('JUMPT', e , l)
...
    elif token.type in ['NOOP']:
        stream.match('NOOP')
        stream.match('SEMI')
        return ('NOOP',)
    else:
        raise SyntaxError("instr: syntax error at {}"
                          .format(token.value))
```

Instruction Tuples!

Implementation: Parser

```
def exp(stream):
    token = stream.pointer()
    if token.type in ['ADD']:
        stream.match('ADD')
        e1 = exp(stream)
        e2 = exp(stream)
        return ('ADD', e1, e2)
    elif token.type in ['SUB']:
        stream.match('SUB')
        e1 = exp(stream)
        if stream.pointer().type in exp_lookahead:
            e2 = exp(stream)
            return ('SUB', e1, e2)
        else:
            return ('UMINUS', e1)
    elif token.type in ['MUL']:
        stream.match('MUL')
        e1 = exp(stream)
        e2 = exp(stream)
        return ('MUL', e1, e2)
...
    elif token.type in ['NAME']:
        v = var(stream)
        return ('NAME', v)
    elif token.type in ['NUMBER']:
        n = num(stream)
        return ('NUMBER', n)
    else:
        raise SyntaxError("exp: syntax error at {}"
                          .format(token.value))
```

Computing an expression tree!

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

```
def exp(stream):
    token = stream.pointer()
    if token.type in ['ADD']:
        stream.match('ADD')
        e1 = exp(stream)
        e2 = exp(stream)
        return ('ADD', e1, e2)
    elif token.type in ['SUB']:
        stream.match('SUB')
        e1 = exp(stream)
        if stream.pointer().type in exp_lookahead:
            e2 = exp(stream)
            return ('SUB', e1, e2)
        else:
            return ('UMINUS', e1)
    elif token.type in ['MUL']:
        stream.match('MUL')
        e1 = exp(stream)
        e2 = exp(stream)
        return ('MUL', e1, e2)
...
    elif token.type in ['NAME']:
        v = var(stream)
        return ('NAME', v)
    elif token.type in ['NUMBER']:
        n = num(stream)
        return ('NUMBER', n)
    else:
        raise SyntaxError("exp: syntax error at {}"
                          .format(token.value))
```

```
def num(stream):
    token = stream.pointer()
    if token.type in ['NUMBER']:
        stream.match('NUMBER')
        return token.value
    else:
        raise SyntaxError("num: syntax error at {}"
                          .format(token.value))
```

According to the parser the expression,

`* + 3 1 2`

gives rise to the term tree,

`('MUL',('ADD', ('NUMBER', 3), ('NUMBER', 1)), ('NUMBER', 2))`

Testing our Parser

```
lutz$ python3
Python 3.8.2 (default, Jun  8 2021, 11:59:35)
[Clang 12.0.5 (clang-1205.0.22.11)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from exp1bytecode_interp_state import state
>>> from exp1bytecode_interp_fe import parse
>>> import pprint
>>> pp = pprint.PrettyPrinter()
>>> input =\
... '''
... store x 10;
... L1:
... print x;
... store x - x 1;
... jumpt x L1;
... stop;
... '''
>>> parse(input)
>>> pp.pprint(state.program)
[('STORE', 'x', ('NUMBER', '10')),
 ('PRINT', ('NAME', 'x')),
 ('STORE', 'x', ('SUB', ('NAME', 'x'), ('NUMBER', '1'))),
 ('JUMPT', ('NAME', 'x'), 'L1'),
 ('STOP',)]
>>> pp.pprint(state.label_table)
{'L1': 1}
>>> 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 expression

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 NAME  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])
        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])
        else:
            state.instr_ix += 1

    elif type == 'JUMP':
        # JUMP label
        state.instr_ix = state.label_table.get(instr[1])

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

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

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

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

…
```

Integer division!

Interpreting Expressions

exp1bytecode_interp.py

```
…
  elif type == 'EQ':
    # '=' 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 == 'LE':
    # '<=' 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 == 'NOT':
    # '!' 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 int(node[1])

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

Representing Booleans as integers.

Top-level Function

exp1bytecode_interp.py

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

    try:
        state.initialize()  # initialize our abstract machine
        parse(input_stream) # build the IR
        interp_program()    # interpret the IR
    except Exception as e:
        print("error: "+str(e))
```

Running from Commandline

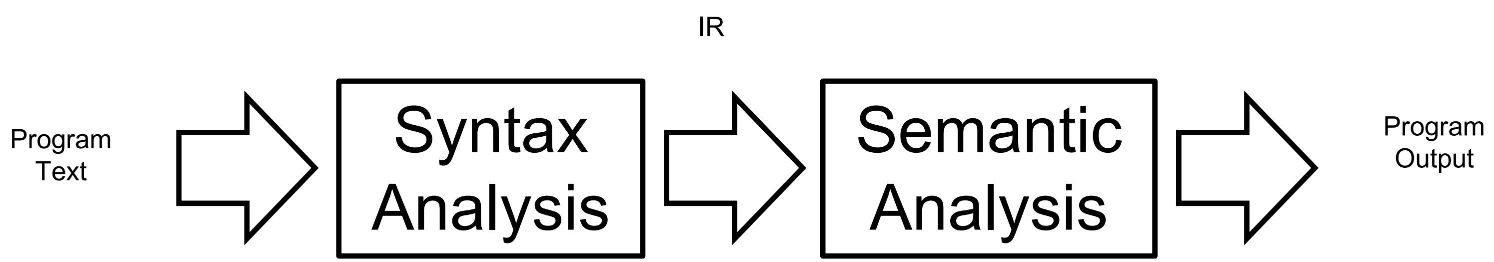
```
if __name__ == '__main__':
    import sys
    import os

    if len(sys.argv) == 1: # no args - read stdin
        char_stream = sys.stdin.read()
    else: # last arg is filename to open and read
        input_file = sys.argv[-1]
        if not os.path.isfile(input_file):
            print("unknown file {}".format(input_file))
            sys.exit(0)
        else:
            f = open(input_file, 'r')
            char_stream = f.read()
            f.close()

    interp(char_stream)
```

Testing our Interpreter

```
$ cat seq.txt
# print a sequence
   input x;
L1:
   print x;
   store x (- x 1);
   jumpt x L1;
   stop;
$ python3 exp1bytecode_interp.py seq.txt
Please enter a value for x: 3
3
2
1
$
```

Interpreter with IR

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



Reading

- Chap 4