Code generation: tree walking

- want to start looking at ways to generate code corresponding to statements expressed in an abstract syntax tree (AST)
- will start with handling simple expressions and primitive operators, then expand to consider function calls, more complex types and operations
- will use the idea of a tree-walking function: something that traverses the AST, outputting code in the target language
- will use a set of fictional assembly language instructions for our target language

Example: x + y * z

- AST would have + as the root, x as left subtree, * as root of right subtree, leaves of * would be y and z
- post-order traversal of tree gives bottom-up evaluation, would process nodes in sequence x y z * +
- Suppose initially register Ra has address of activation record (AR) none of the variables are in registers yet, register locations are stored as offsets from start of AR
- Need to load each item's offset into a register, then use that to load item value, then perform computations
- (and using virtual registers to hold intermediate values)

Ex: x + y * z continued

- (making up different forms of load instructions here) sequence x y z * +, generated code might look like

```
loada x,r1 // loads offset of x into r1
load ra,r1,r2 // from addr AR+offset, load x content into r2
loada y,r3
load ra,r3,r4 // so now y in r4
loada z,r5
load ra,r5,r6 // so now z in r6
mult r4,r6,r7 // r7 = y * z
add  r2,r7,r8 // r8 = x + (y * z)
```

Tree-walking routine

- need a tree-walking routine to perform the AST traversal and output the desired code
- values are getting stored in registers, so routine needs local variables to store the register names to use, and a routine to look up the next available register name
- for now, assume AST nodes are just binary operators, numeric literals, or variable names
- For now, assume we have an output function that generates correct code for an individual operator/arg list

walk(node n)

```
locals: res, tmp1, tmp2

if operator(n)
    tmp1 = walk(left(n))
    tmp2 = walk(right(n))
    res = getNextReg()
    output(op(n), tmp1, tmp2, res)
```

```
else if number(n)
    res = getNextReg()
    output(loadI, value(n), nil, res)

else if identifier(n)
    tmp1 = baseaddress(n)
    tmp2 = offset(n)
    res = getNextReg()
    output(load, tmp1, tmp2, res)
```

- output(op, arg1, arg2, destreg)
  - use a switch that looks at operand type (e.g. add, mult, load, loada, etc) and generates the right line(s) of assembler, embedding the provided arguments and destination register
- walk would need to be expanded, cases for ternary ops, unary ops (left and right associative), etc

Loading from registers

- if x,y,z were already in registers then the pairs of load instructions would be irrelevant (and possibly incorrect, e.g. if x was in a register and that register value had changed since x was loaded)
- tree walk might add a call to a lookup function to see if a specific storage location was already loaded
- access to some locations might require other instructions (e.g. access to global variables might first require loading base address of global space)

Access type

- pass-by-value params:
  - as per variables
- pass-by-reference params:
  - if already in register use that
  - otherwise
    - load its offset into one register
    - use that plus the AR register to load the parameter value (e.g. the pointer to the actual variable to work on)
    - use THAT address to load the actual desired content

Register counts

- after we traverse one subtree, we need to use a register to store its result while traversing the other subtree
- to reduce total count of registers used, it's best to first traverse the subtree that requires fewer registers
- e.g. Suppose left tree requires 5, right tree requires 3
  - if we do left tree first, we use 5 during its traversal and 4 during right's traversal
  - if we do right tree first, we use 3 during its traversal and 6 during left's traversal

Optimizations

- we'd like our compiler to be smarter than just to follow the raw precedence rules verbatim
- e.g. a + b – c + a + b
- ideally, recognize the (a + b) replication, at least reuse that intermediate result, and possibly even optimize further with a bit shift, e.g. implement as ((a+b) <<2) - c)
- order of ops and limits of floating point precision can also have an impact, e.g. x1 + x2 + ... xn, adding from smallest to largest can give different results than largest to smallest