How To Start Writing Compilers Without a Ph.D.

revival/fnuque

Approach

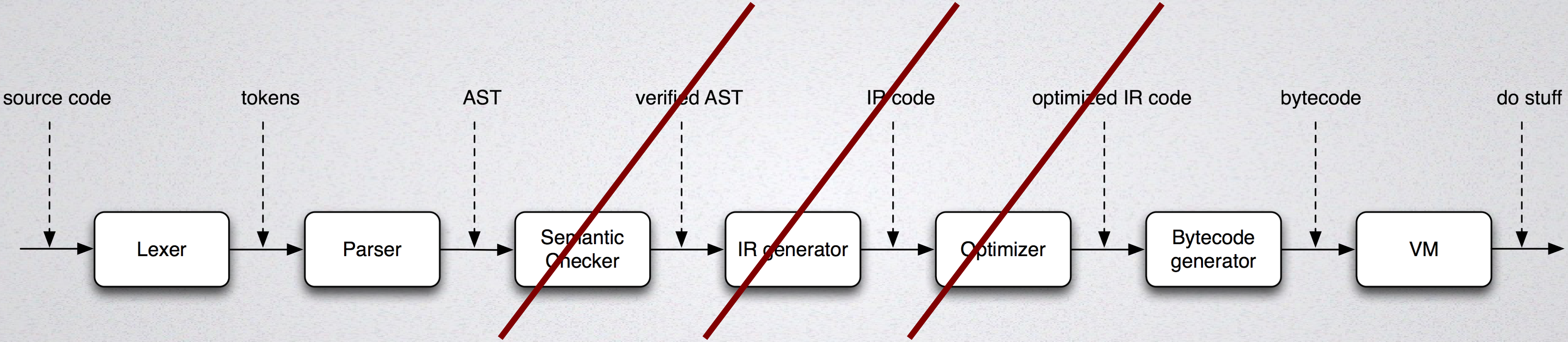
- Low-level & practical
- Looking at source
- Specific implementation

http://github.com/revivalizer/compilertalk

Outline

- Preliminaries
- Lesson 1: Addition
- Lesson 2: Complex expressions
- Lesson 3: Fixing associativity
- Lesson 4: Function calls
- Lesson 5: Variables
- Lesson 6: Control flow
- Questions?

Compiler structure

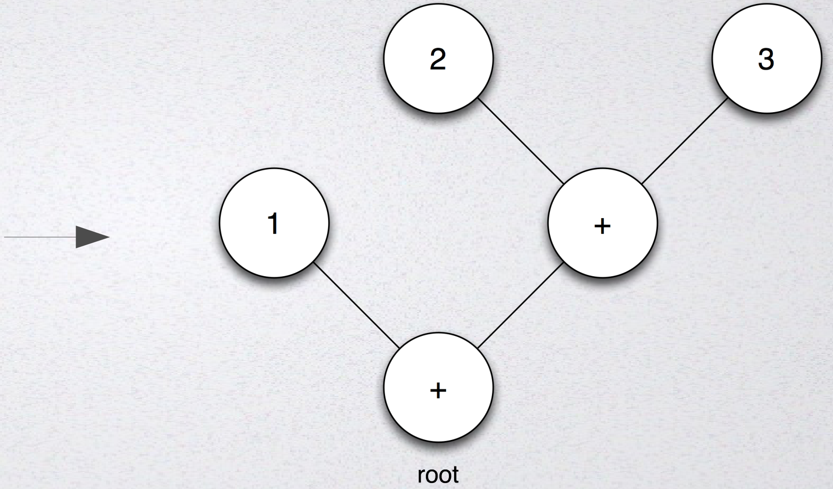


Lesson 1: Addition

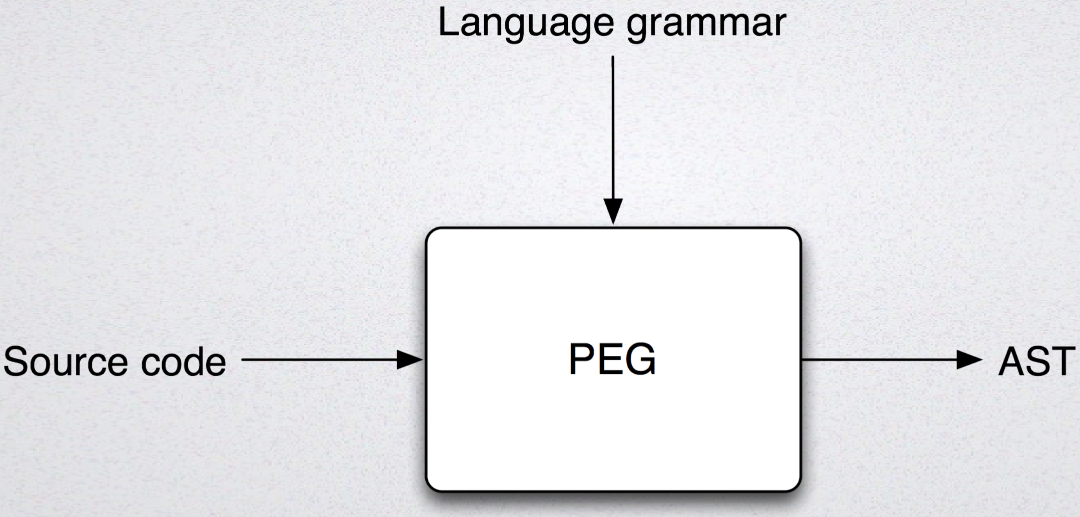
"1+2+3"

Lesson 1: Addition

"1+2+3"



PEGs



Grammar

“1+2+3”

Rules

- Integers
- Plus operator
- Whitespace

PEGs

- Define rule: $r_1 \leftarrow e$
- String match: 'string' / [a-z]
- Rule reference: r
- Sequence: $e_1\ e_2$
- Zero-or-more: e*
- One-or-more: e+
- Optional: e?
- Ordered choice: e_1 / e_2

Formal rules

“1+2+3”

```
ws ← %s+
integer ← '-'? [0-9]+ ws?
plus_operator ← '+' ws?
plus_expr <- integer plus_operator integer
```

Formal rules

“1+2+3”

```
ws ← %s+
integer ← '-'? [0-9]+ ws?
plus_operator ← '+' ws?
plus_expr <- integer plus_operator plus_expr
```

Formal rules

“1+2+3”

```
ws ← %s+
integer ← '-'? [0-9]+ ws?
plus_operator ← '+' ws?
plus_expr <- integer plus_operator plus_expr
           / integer
```

Captures (1)

```
root          ← ws? plus_expr
plus_expr     ← integer
                plus_operator
                plus_expr
              / integer
plus_operator ← '+' ws?
integer       ← '-'? [0-9]+ ws?
ws            ← %s+
```

Captures (2)

```
root          ← ws? (plus_expr) → {}
plus_expr     ← (integer
                 plus_operator
                 plus_expr) → {}
               / integer
plus_operator ← ('+' ws?)  → {}
integer       ← ('-'? [0-9]+ ws?) → {}
ws            ← %s+
```

Captures (3)

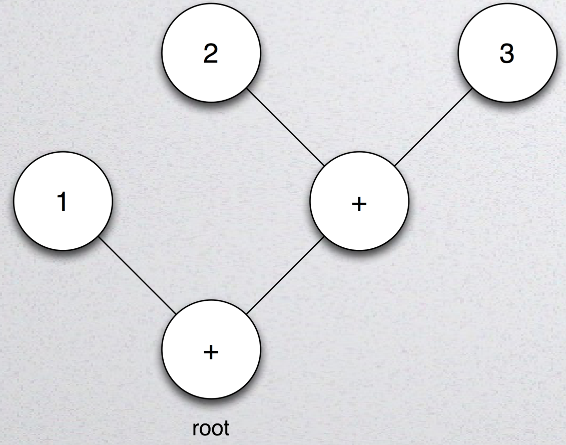
```
Root          ← ws? ({:root: plus_expr:}) → {}
plus_expr     ← ({:operand1: integer:}
                 {:operator: plus_operator:}
                 {:operand2: plus_expr:}) → {}
               / integer
plus_operator ← ({:type: '+':} ws?)  → {}
integer       ← ({:value: '-'? [0-9]+:} ws?) → {}
ws            ← %s+
```

Captures (4)

```
root          ← ws? ({:root: plus_expr:}) → {}
plus_expr     ← ({:tag: ''->'binary_op':}
                 {:operand1: integer:}
                 {:operator: plus_operator:}
                 {:operand2: plus_expr:}) → {}
                / integer
plus_operator ← ({:type: '+':} ws?)  → {}
integer       ← ({:tag: '' -> 'literal_int':} {:value: '-'? [0-9]+:} ws?) → {}
ws            ← %s+
```

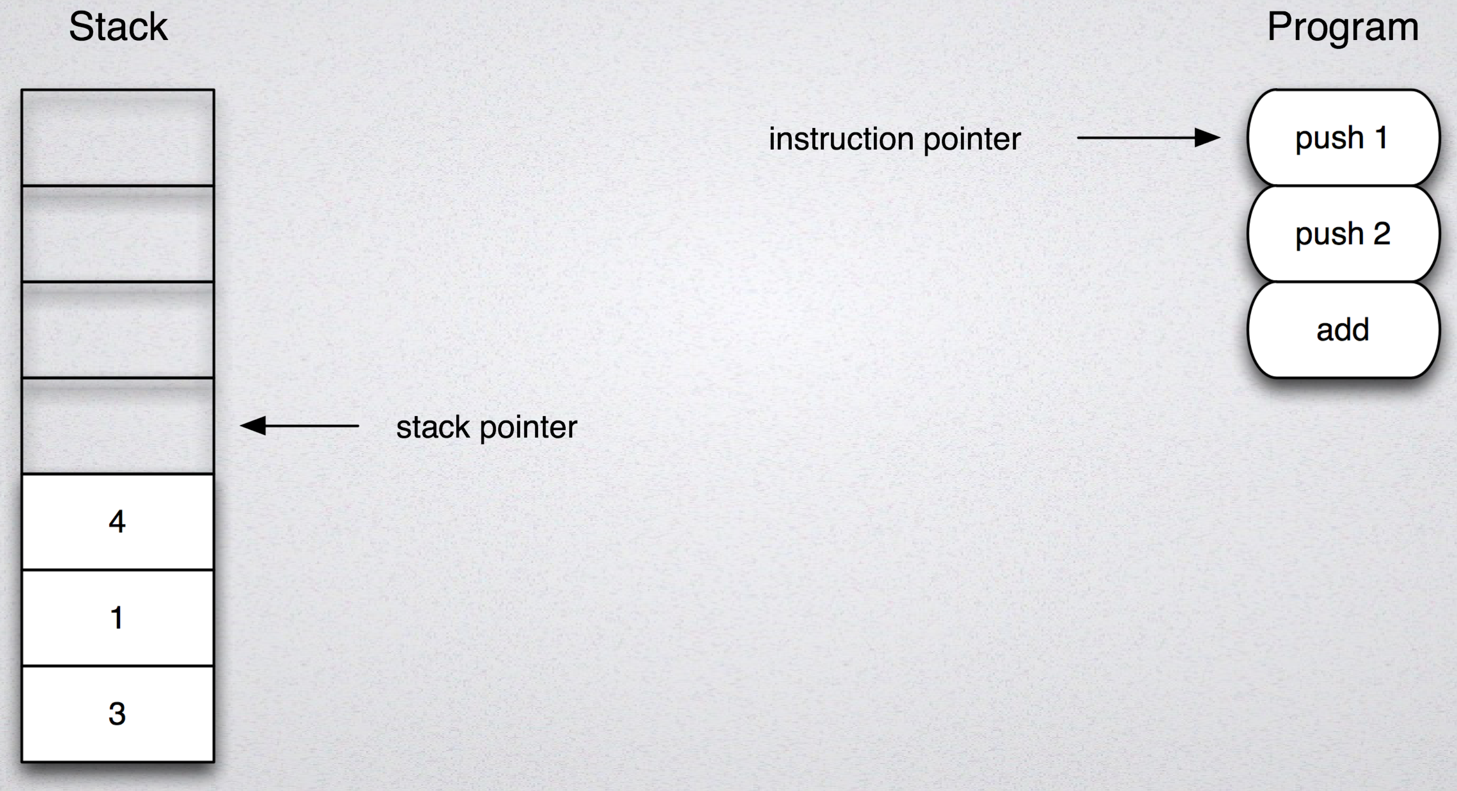
Resulting AST

"1+2+3"

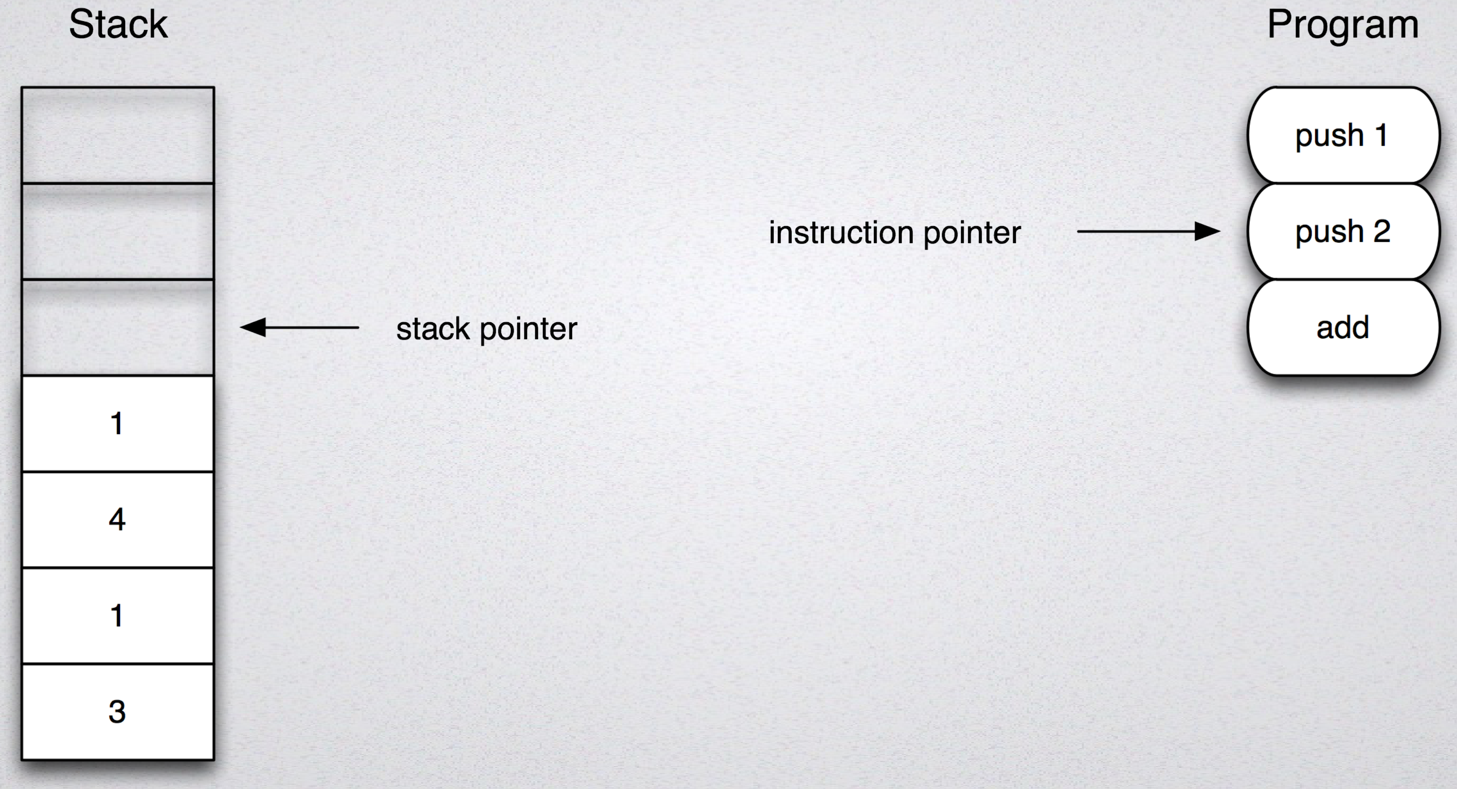


```
root = {
    operand2 = {
        operand2 = {
            value = "3",
            tag = "literal_int"},
        operator = {
            type = "+"},
        operand1 = {
            value = "2",
            tag = "literal_int"},
        tag = "binary_op"},
    operator = {
        type = "+"},
    operand1 = {
        value = "1",
        tag = "literal_int"},
    tag = "binary_op"},
```

Stackbased VM (1)



Stackbased VM (2)



Stack

1

4

1

3

stack pointer

instruction pointer

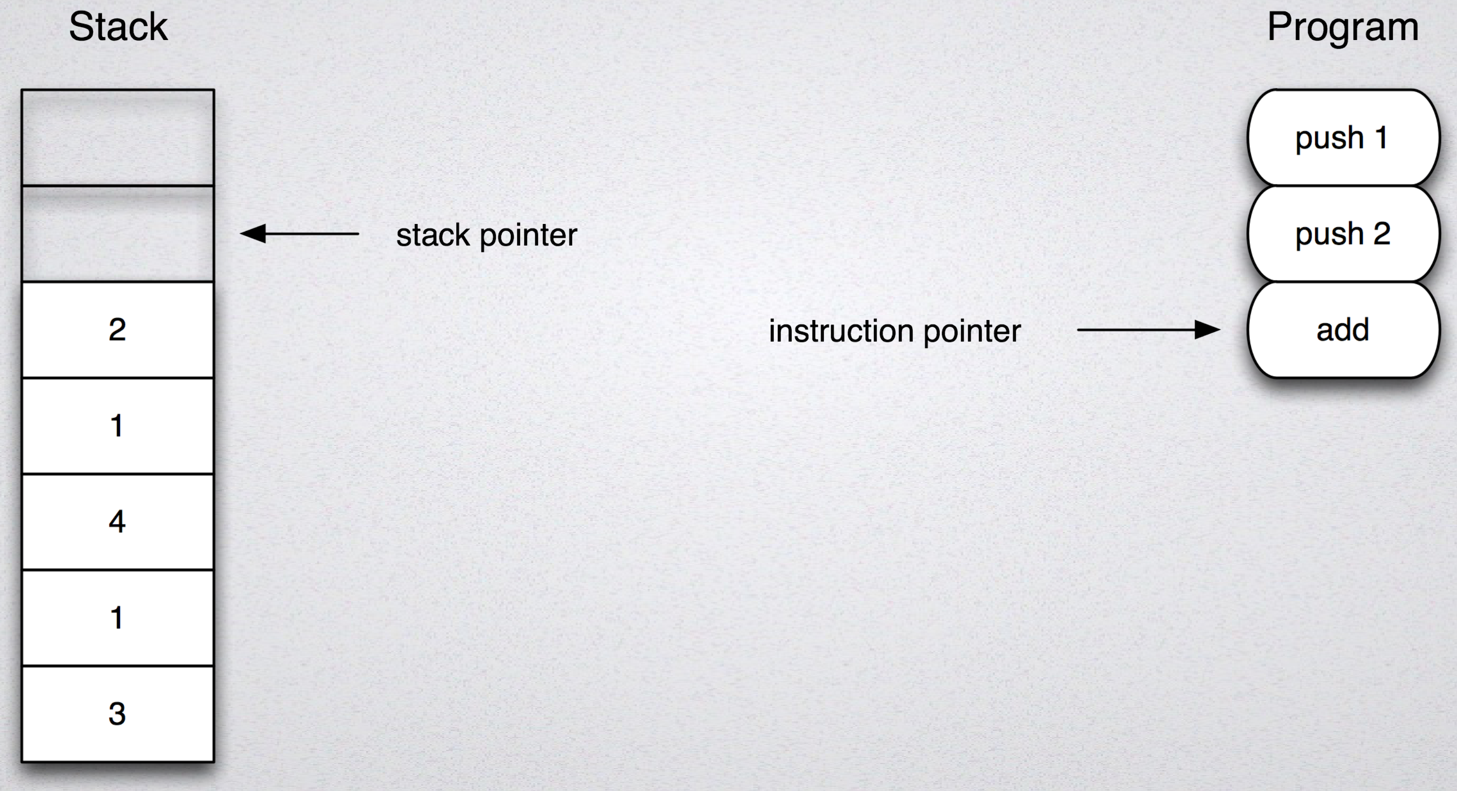
Program

push 1

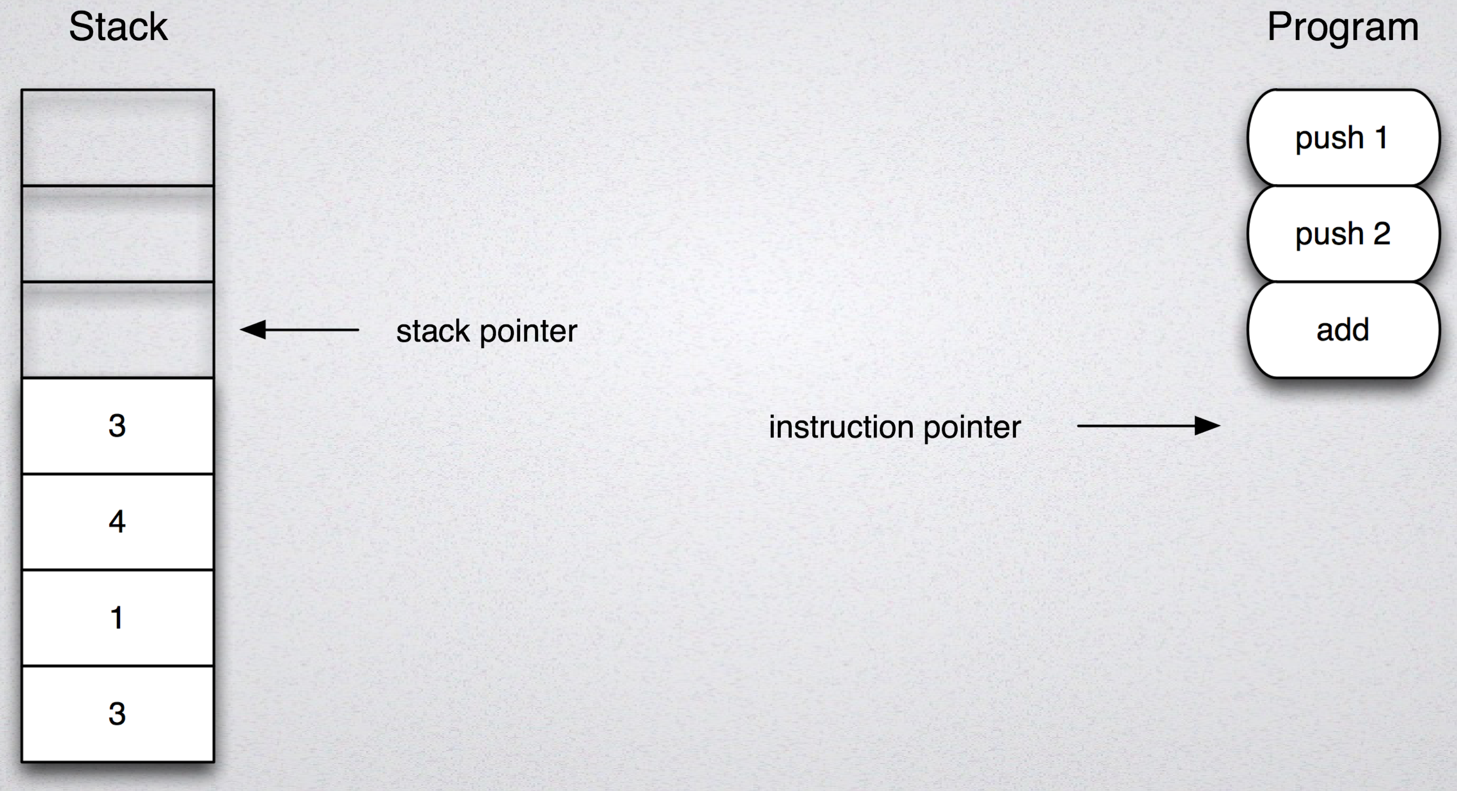
push 2

add

Stackbased VM (3)



Stackbased VM (4)



VM instructions

```
typedef unsigned short opcode_t;
typedef unsigned short argument_t;

kOpReturn   = 0x00,
kOpPush     = 0x10,
kOpAdd      = 0x20,
```

VM context

```
typedef struct
{
   opcode_t*   bytecode;
   constant_t* constants;
}  vm_program;

float vm_stack[1000];
```

VM

```
void vm_run(vm_program* program, opcode_t ip, constant_t* stack)
{
    while (1) {
             opcode_t   op     = program->bytecode[ip];
             argument_t op_arg = program->bytecode[ip+1];

             switch (op) {
                 case kOpReturn:
                     return;
                 case kOpPush:
                     stack[0] = program->constants[op_arg];
                     stack++; ip+=2;
                     break;
                 case kOpAdd:
                     stack[-2] = stack[-2] + stack[-1];
                     stack--; ip+=1;
                     break;
             }
    }
}
```

Bytecode generation

```
function generate_bytecode(node, program)
    -- Generate bytecode for this node
    if (node.tag=="literal_int") then
        node.constant_index = program.constants:get_id(tonumber(node.value))

        program.bytecode:insert(kOpPush)
        program.bytecode:insert(node.constant_index-1)
    elseif (node.tag=="binary_op" and node.operator.type=='+') then
        generate_bytecode(node.operand1, program)
        generate_bytecode(node.operand2, program)
        program.bytecode:insert(kOpAdd)
    end
end
```

Summary

1) Determined language features

2) Wrote PEG grammar

3) Defined instructions/opcodes

4) Wrote bytecode generator

5) Wrote simple VM

Status

kOpPush, kOpAdd

literal_int, binary_op (+)

“1+2+3”

“1   + 2”

“4 + 567 + 9 + 1”

Lesson 2: Complex expressions

Operators

```
Arithmetic : +  -  *  /  %
Relational : ==  !=  <  <=  >  >=
Logical    : &&  ||
Unary      : !  +  -
```

Precedence

1+2*3 = ?

1+(2*3) = 7

Precedence in C

Precedence	Operator	Description
3	+ - !	Unary plus and minus Logical unary not
5	* / %	Mul, div, modulo
6	+ -	Add, subtract
8	< <= > >=	Relational operators
9	== !=	Equality operators
13	&&	Logical and
14	||	Logical or

Operators

```
not_operator                         <- '!'  ws?
plus_operator                        <- '+'  ws?
minus_operator                       <- '-'  ws?
multiplication_operator              <- '*'  ws?
division_operator                    <- '/'  ws?
modulo_operator                      <- '%'  ws?
addition_operator                    <- '+'  ws?
subtraction_operator                 <- '-'  ws?
less_than_or_equal_operator          <- '<=' ws?
greater_than_or_equal_operator       <- '>=' ws?
less_than_operator                   <- '<'  ws?
greater_than_operator                <- '>'  ws?
equality_operator                    <- '==' ws?
inequality_operator                  <- '!=' ws?
logical_and_operator                 <- '&&' ws?
logical_or_operator                  <- '||' ws?
```

Operators

```
additive_operators   <- addition_operator / subtraction_operator
multitive_operators  <- multiplication_operator /
                        division_operator /
                        modulo_operator
equality_operators   <- equality_operator / inequality_operator
relational_operators <- less_than_or_equal_operator /
                        greater_than_or_equal_operator /
                        less_than_operator /
                        greater_than_operator
unary_operators      <- not_operator / plus_operator / minus_operator
```

Expression matching chain

```
                        Previously
plus_expr <- integer plus_operator plus_expr / integer
```

```
expression  <- additive
additive    <- multitive additive_operators additive   / multitive
multitive   <- primary   multitive_operators multitive / primary
primary     <- integer / open_parens expression close_parens
```

1*2+3*4

Expression matching chain

```
expression  <- logical_or
logical_or  <- logical_and   logical_or_operator    logical_or   / logical_and
logical_and <- equality      logical_and_operator   logical_and  / equality
equality    <- relational    equality_operators     equality     / relational
relational  <- additive      relational_operators   relational   / additive
additive    <- multitive     additive_operators     additive     / multitive
multitive   <- unary         multitive_operators    multitive    / unary
unary       <-               unary_operators        unary        / primary
primary     <- integer / open_parens expression close_parens
```

Compiler and VM changes

```
if (node.tag=="literal_int") then
    program.bytecode:insert(kOpPush)
    bytecode:insert(node.constant_index-1)
elseif (node.tag=="binary_op") then
    generate_bytecode(node.operand1)
    generate_bytecode(node.operand2)
    program.bytecode:insert(binary_opcodes[node.operator.type])
elseif (node.tag=="unary_op") then
    generate_bytecode(node.operand)
    program.bytecode:insert(unary_opcodes[node.operator.type])
end
```

Status

kOpPush, kOpAdd, **kOpAdd, kOpSubtract, kOpMultiply, kOpDivide, kOpModulo, kOpEqual, kOpNotEqual, kOpLessThan, kOpLessThanOrEqual, kOpGreaterThan, kOpGreaterThanOrEqual, kOpLogicalAnd, kOpLogicalOr, kOpNot, kOpPlus, kOpMinus**

literal_int, **binary_op, unary_op**

“1*2/3-4%5”

“1<2 || 56/19>=3”

“!((3+4>3%(56))”

Lesson 3: Fixing associativity

Associativity

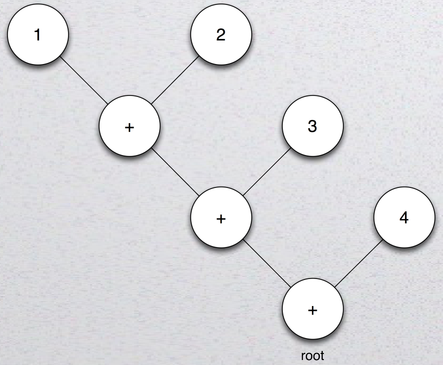
1+2+3+4

6-3-2-1 = 0

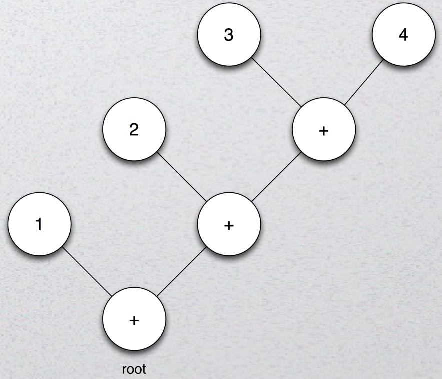
6-(3-(2-1)) = 4

whoops

Left associative : ((1+2)+3)+4



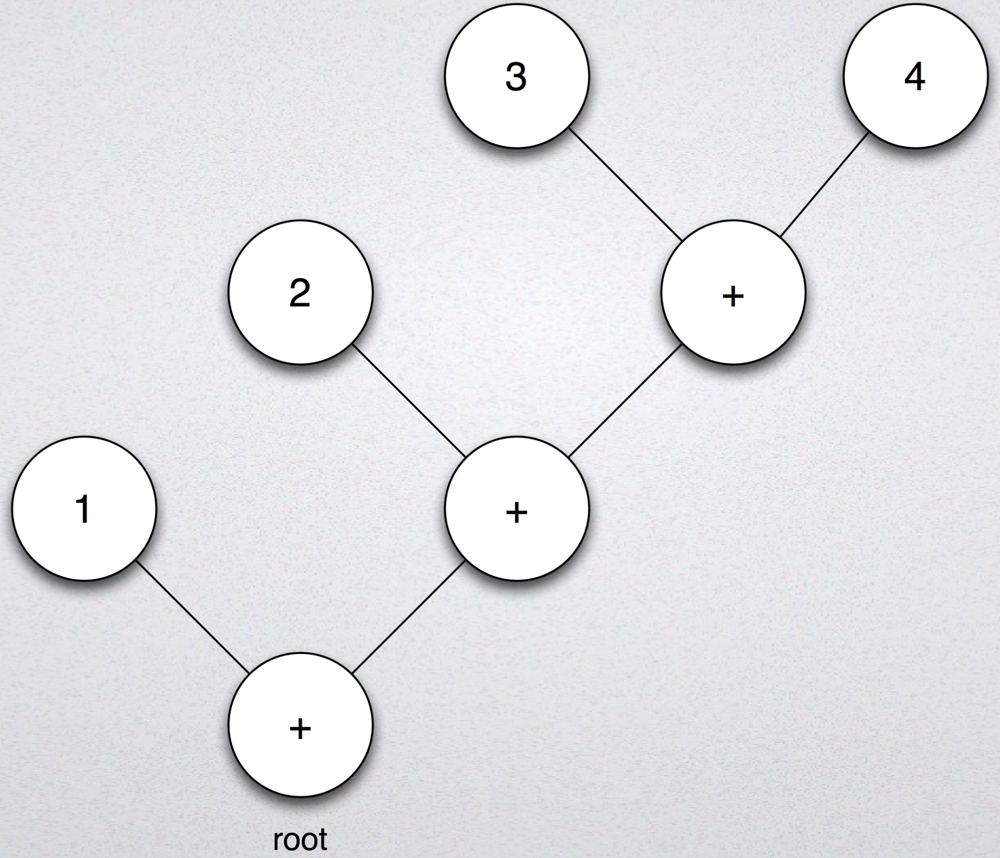
Right associative: 1+(2+(3+4))



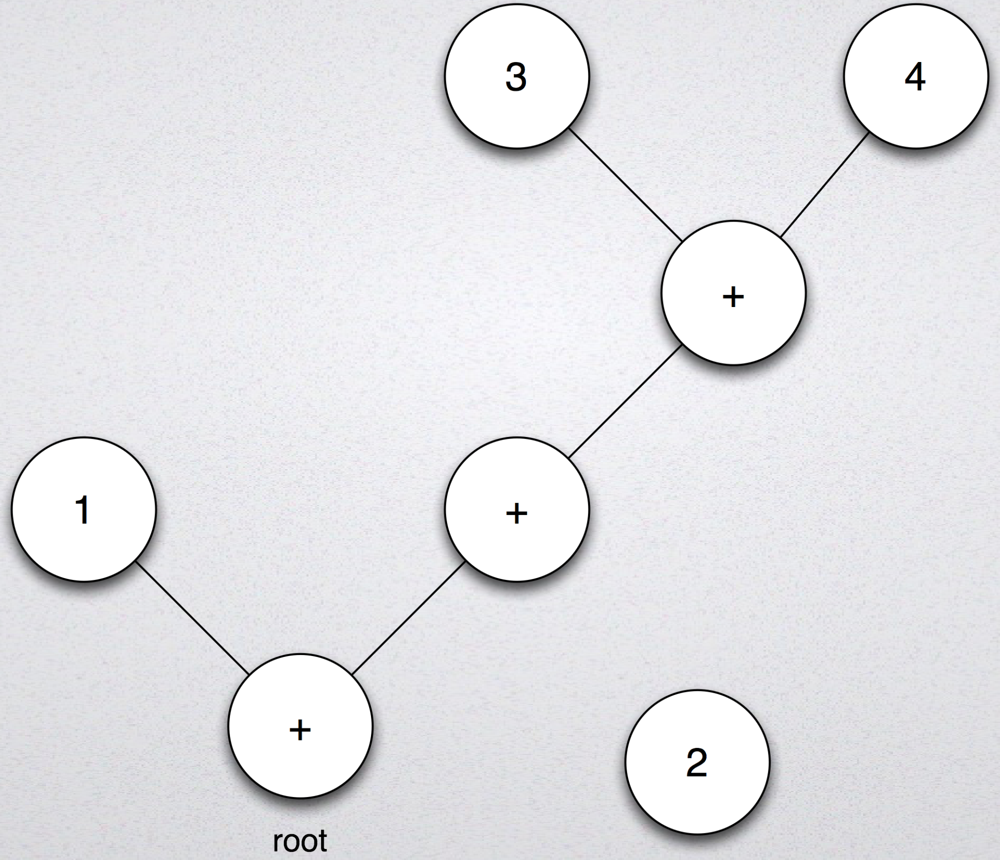
Capture precedence!

```
addition_operator <- ({:precedence: '' -> '6':}
                      {:assoc: '' -> 'left':}
                      {:type: '+':} ws?) -> {}
```

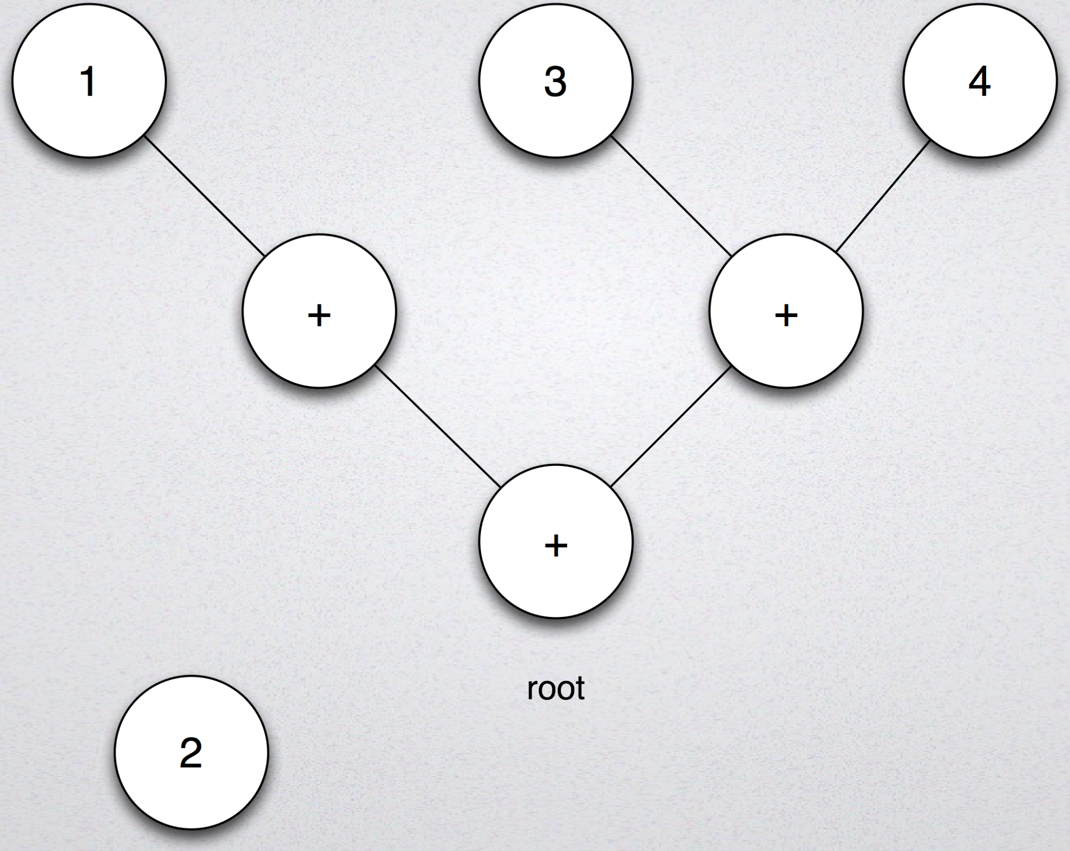
Associativity fix (I)



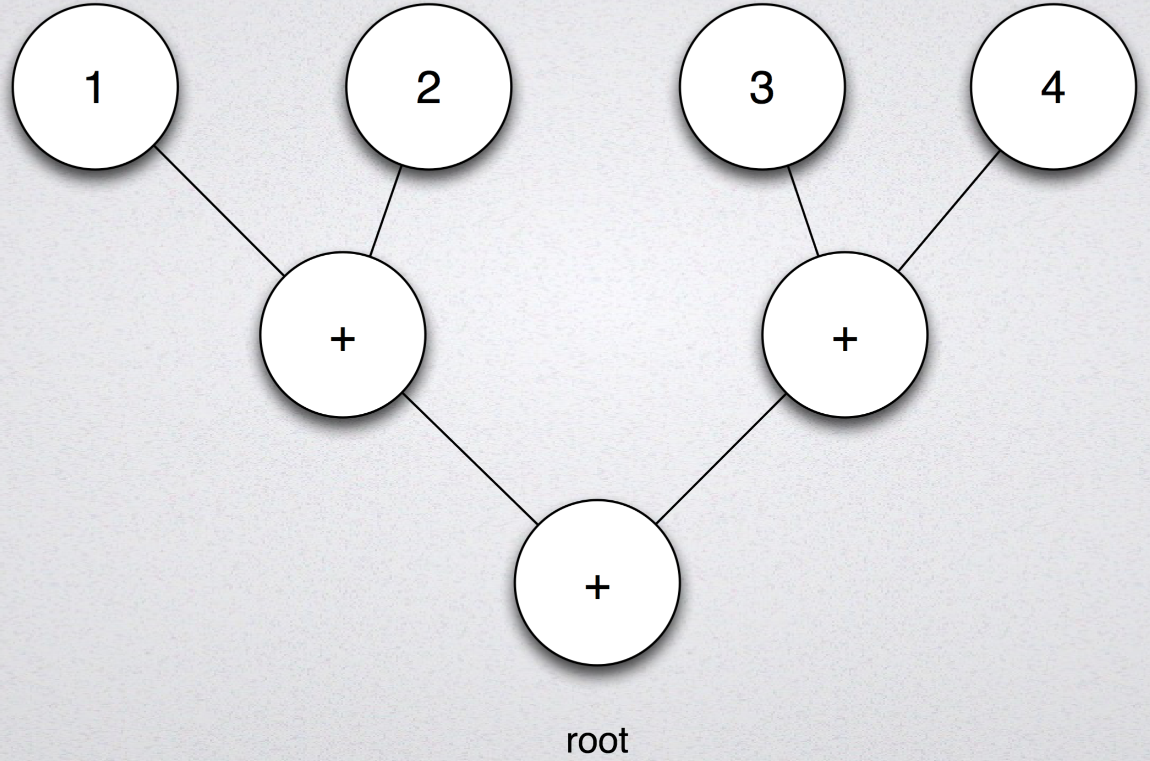
Associativity fix (2)



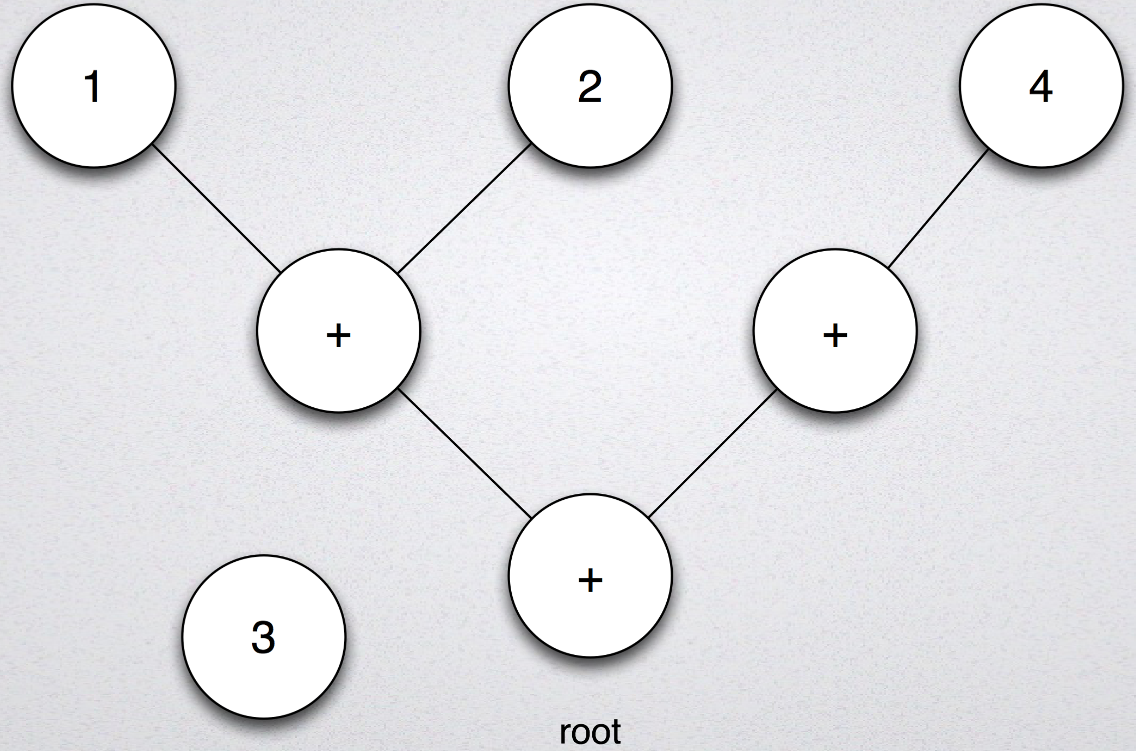
Associativity fix (3)



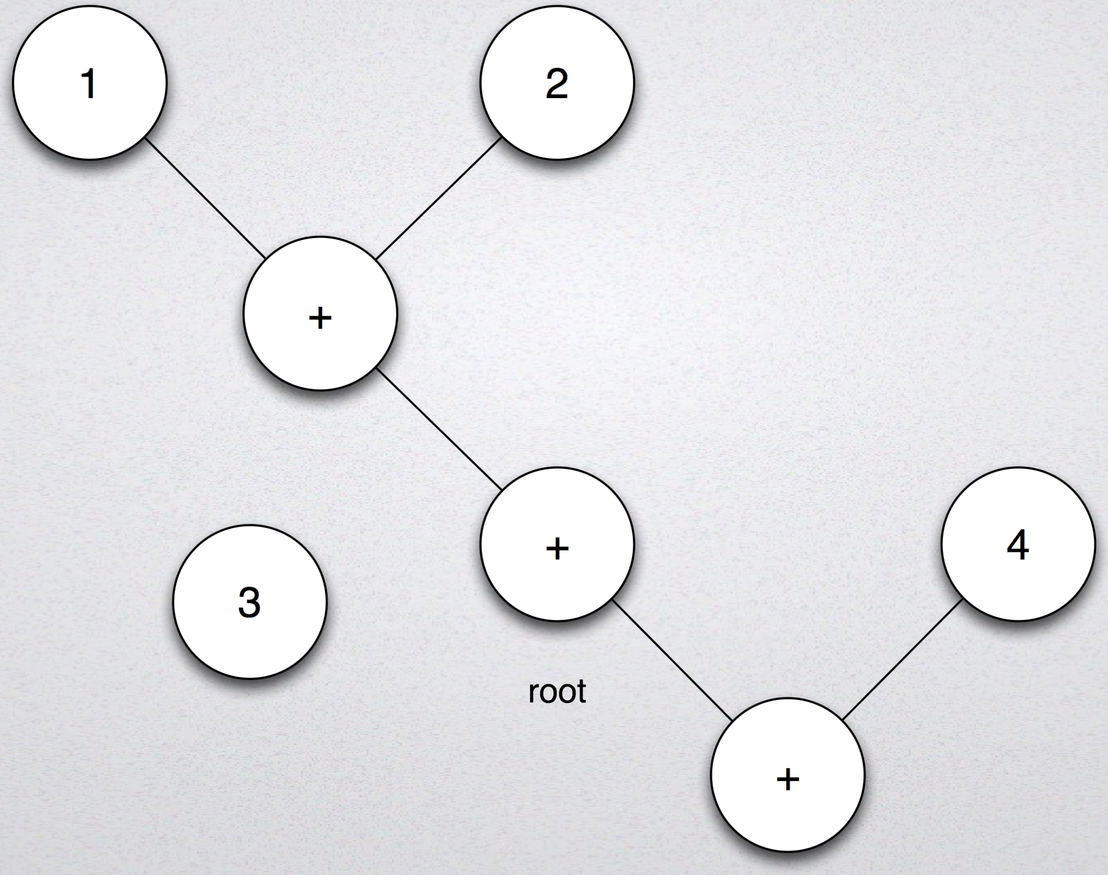
Associativity fix (4)



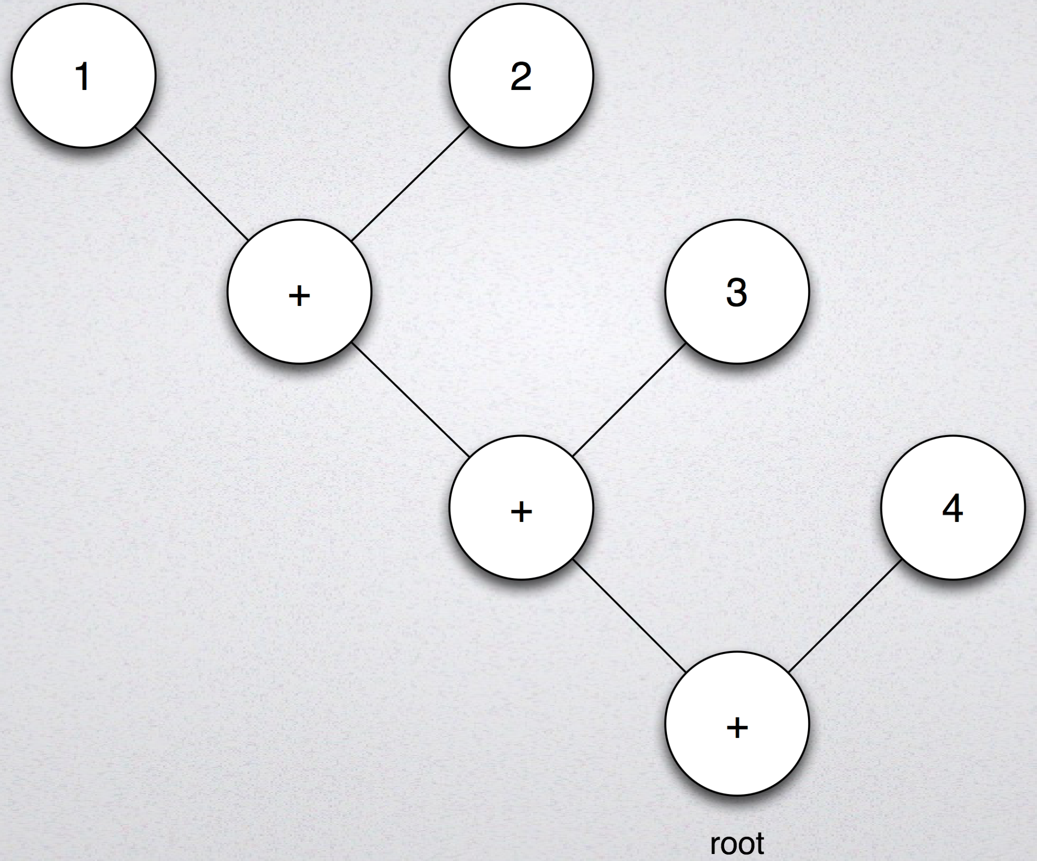
Associativity fix (5)



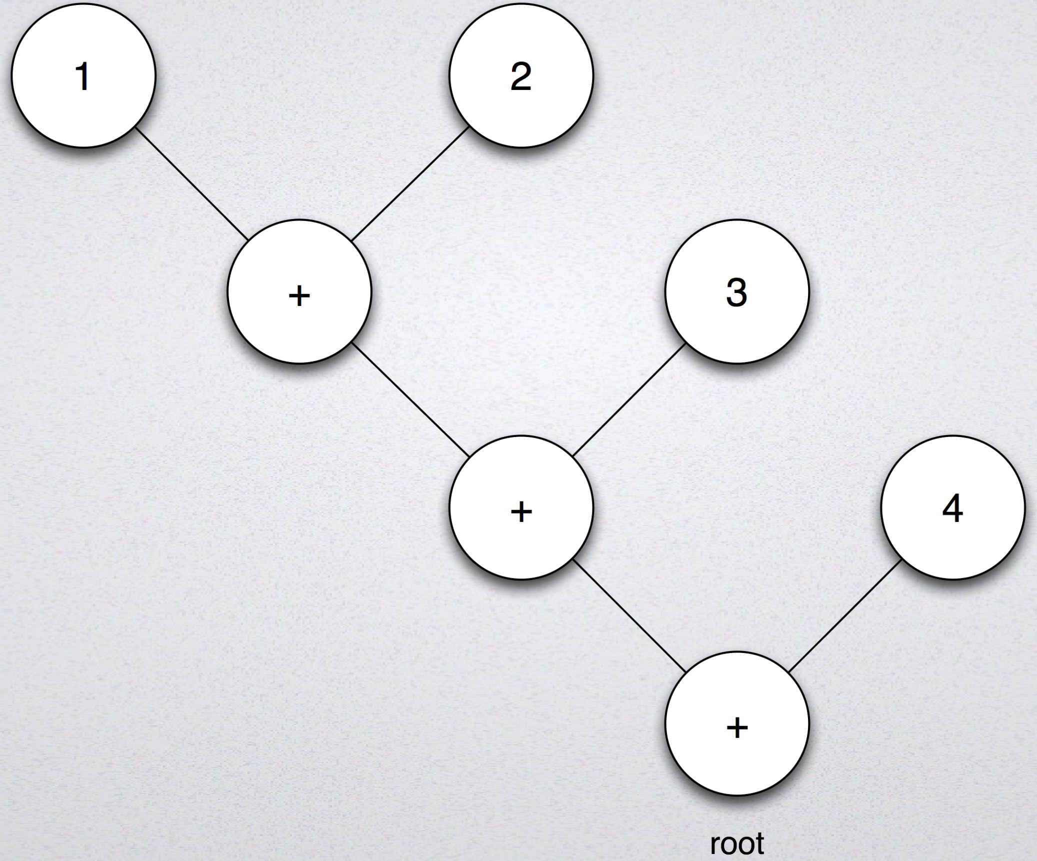
Associativity fix (6)



Associativity fix (7)



Associativity fix (7)



Lesson 4: Function calls

Function prototypes (1)

```
void print(number);
number sin(number);
number cos(number);
```

Function prototypes (2)

```
functions = {
    ["print"] = {
            ["id"] = 0,
            ["arguments"] = {[1] = {["type"] = "number"}},
            ["result"] = "void"},
    ["sin"] = {
            ["id"] = 1,
            ["arguments"] = {[1] = {["type"] = "number"}},
            ["result"] = "number"},
    ["cos"] = {
            ["id"] = 2,
            ["arguments"] = {[1] = {["type"] = "number"}},
            ["result"] = "number"}},
```

Statements

```
start                      <-  ws? statement_list
statement_list             <-  statement*
statement                  <-  function_call_statement
function_call_statement    <-  function_call semicolon
```

Function calls

```
function_call <- identifier open_parens function_call_arguments? close_parens
function_call_arguments <- expression (',' ws? expression)*

identifier    <- [a-zA-Z] [a-zA-Z0-9_]* ws?

primary     <- integer / function_call / open_parens expression close_parens
```

Statements bytecode generation

```
function generate_bytecode
    […]
    elseif (node.tag=="statement_list") then
            for i,statement in ipairs(node) do
                generate_bytecode(statement, program)
            end
    elseif (node.tag=="function_call") then
            for i,arg in ipairs(node.arguments) do
                generate_bytecode(arg, program)
            end

            program.bytecode:insert(kOpCallFunc)
            program.bytecode:insert(functions[node.identifier].id)
    elseif (node.tag=="function_call_statement") then
            generate_bytecode(node.function_call, program)

            if (functions[node.function_call.identifier].result=="number") then
                program.bytecode:insert(kOpPop)
            end
    end
end
```

VM changes

```
case kOpPop:
{
    stack--;
    ip+=1;
    break;
}
```

```
case kOpCallFunc:
{
    switch (op_arg)
    {
            case kFuncPrint:
                printf("%f\n", stack[-1]);
                stack--;
                break;
            case kFuncSin:
                stack[-1] = sinf(stack[-1]);
                break;
            case kFuncCos:
                stack[-1] = cosf(stack[-1]);
                break;
    }

    ip+=2;
    break;
}
```

Status

kOpPush, kOpAdd, kOpAdd, kOpSubtract, kOpMultiply, kOpDivide, kOpModulo, kOpEqual, kOpNotEqual, kOpLessThan, kOpLessThanOrEqual, kOpGreaterThan, kOpGreaterThanOrEqual, kOpLogicalAnd, kOpLogicalOr, kOpNot, kOpPlus, kOpMinus, **kOpFuncCall, kOpPop**

literal_int, binary_op, unary_op, **statement_list, function_call_statement, function_call**

```
“print(1+2);
print(cos(31415/10000));
print(sin(2)*cos(3)<0);”
```

Lesson 5: Variables

VM implications

Only global variables

VM context: bytecode, stack, set of constants

+ variables

New opcodes

```
case kOpPushVar:
    stack[0] = variables[op_arg];
    stack++;
    ip+=2;
    break;

case kOpPopVar:
    variables[op_arg] = stack[-1];
    stack--;
    ip+=2;
    break;
```

Variable parsing

```
statement         <- function_call_statement / assign_statement

assign_statement  <- identifier '=' ws? expression semicolon

primary           <- integer /
                     function_call /
                     variable /
                     open_parens expression close_parens

variable          <- identifier
```

Variable compilation

```
function generate_bytecode(node, program)
    [...]
    elseif (node.tag=="variable") then
            node.variable_index = program.variables:get_id(node.identifier)

            program.bytecode:insert(kOpPushVar)
            program.bytecode:insert(node.variable_index-1)
    elseif (node.tag=="assign_statement") then
            generate_bytecode(node.expression, program)

            node.variable_index = program.variables:get_id(node.identifier)

            program.bytecode:insert(kOpPopVar)
            program.bytecode:insert(node.variable_index-1)
    [...]
```

Status

kOpPush, kOpAdd, kOpAdd, kOpSubtract, kOpMultiply, kOpDivide, kOpModulo, kOpEqual, kOpNotEqual, kOpLessThan, kOpLessThanOrEqual, kOpGreaterThan, kOpGreaterThanOrEqual, kOpLogicalAnd, kOpLogicalOr, kOpNot, kOpPlus, kOpMinus, kOpFuncCall, kOpPop, **kOpPushVar, kOpPopVar**

literal_int, binary_op, unary_op, statement_list, function_call_statement, function_call, **assign_statement, variable**

```
a = cos(19*b);
print(a*100);
y = sin(angle)*radius;
```

Lesson 6: Conditionals and loops

Control flow statements

if

if else

while

Control flow parsing

```
statement_list <- statement*
statement      <- if_else_statement / if_statement / while_statement /
                  function_call_statement / assign_statement
block          <- open_brace statement_list close_brace / statement



if_statement      <- 'if' ws? open_parens expression close_parens block
if_else_statement <- 'if' ws? open_parens expression close_parens block
                        'else' ws? block
while_statement   <- 'while' ws? open_parens expression close_parens block
```

Labels

- Jump destinations
- program.labels → label soup
- Clean up later
- **create_label**(program)
- program.bytecode:insert(**create_label_ref**(label))
- Layer of indirection in bytecode: {tag="label_ref", id=label.id}
- Passes after bytecode generation
  - Generate sorted, distinct label list
  - Replace label references in bytecode with fixed up label ids

Label list

- Label list exported with bytecode, along with constants
- List of bytecode adresses
- VM context
  - bytecode
  - constants
  - labels
  - stack
  - variables

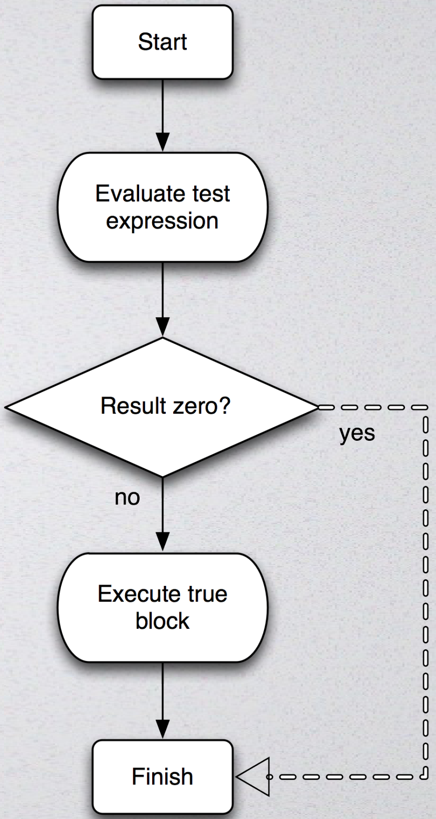
Control flow VM changes

```
case kOpJump:
   ip = program->labels[op_arg];
   break;
```

```
case kOpJumpEqual:
   if (stack[-1]==0.f)
          ip = program->labels[op_arg];
   else
          ip += 2;
   stack--;
   break;
```

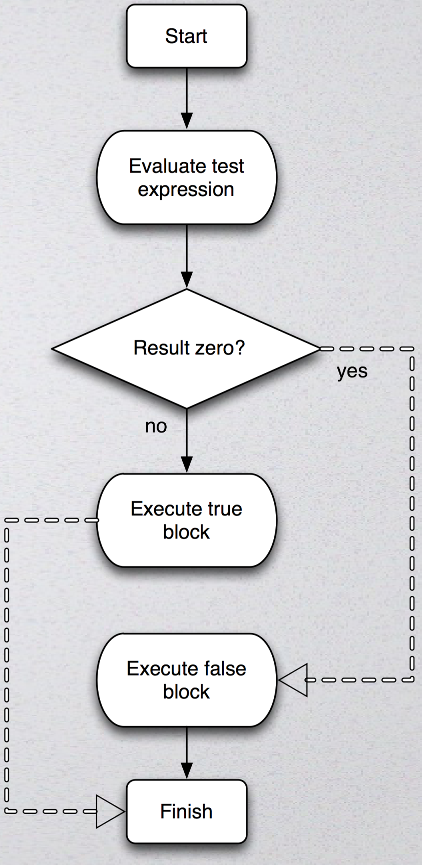
If compilation

```
function generate_bytecode(node, program)
    [...]
    elseif (node.tag=="if_statement") then
        node.label_finish = create_label(program)

        generate_bytecode(node.expression, program)

        program.bytecode:insert(kOpJumpEqual)
        program.bytecode:insert(
           create_label_ref(node.label_finish))

        generate_bytecode(node.block, program)

        node.label_finish.address = #program.bytecode
    [...]
```



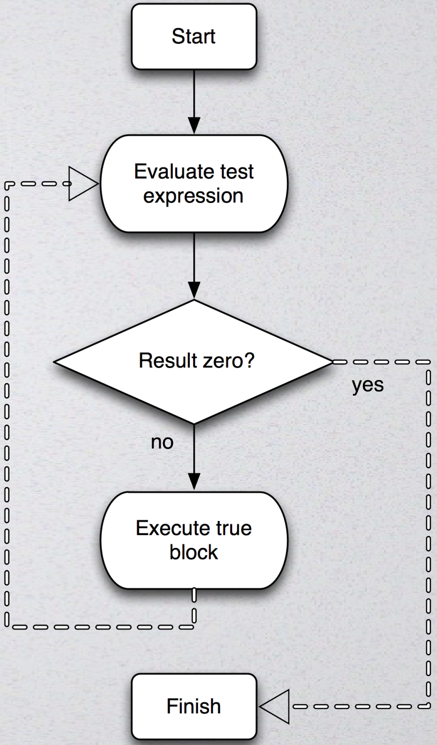
If-Else compilation

```
function generate_bytecode(node, program)
    [...]
        node.label_else = create_label(program)
        node.label_finish = create_label(program)

        generate_bytecode(node.expression, program)

        program.bytecode:insert(kOpJumpEqual)
        program.bytecode:insert(create_label_ref(node.label_else))

        generate_bytecode(node.if_block, program)

        program.bytecode:insert(kOpJump)
        program.bytecode:insert(create_label_ref(node.label_finish))

        node.label_else.address = #program.bytecode

        generate_bytecode(node.else_block, program)

        node.label_finish.address = #program.bytecode
    [...]
```



While compilation

```
function generate_bytecode(node, program)
    [...]
        node.label_test = create_label(program)
        node.label_finish = create_label(program)

        node.label_test.address = #program.bytecode
        generate_bytecode(node.expression, program)

        program.bytecode:insert(kOpJumpEqual)
        program.bytecode:insert(
            create_label_ref(node.label_finish))

        generate_bytecode(node.block, program)
        program.bytecode:insert(kOpJump)
        program.bytecode:insert(
            create_label_ref(node.label_test))

        node.label_finish.address = #program.bytecode
    [...]
```



Sorry about the example language.

Hopefully you already have some ideas for extending it.

Summary

- Writing compilers = fun
- Trading compilation speed → power = good
- Writing your own parser and lexer = stupid (possible instructive)
- Using C++ to write a compiler = silly

Questions!