

# **CSC2/455 Software Analysis and Improvement Intermediate Representations (IRs)**

---

Sreepathi Pai

January 22, 2024

URCS

# Outline

Introduction

Midend

Miscellaneous

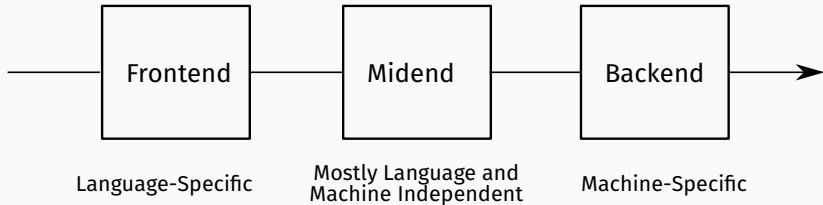
# Outline

Introduction

Midend

Miscellaneous

# Classic Compiler Architecture



Recommended reading: Chris Lattner, LLVM, The Architecture of Open Source Applications

# Outline

Introduction

**Midend**

Miscellaneous

## What does the midend do?

- Mostly language and machine independent analyses
- Majority of analyses run in this stage
- Multiple *intermediate representations* used
  - Starts from abstract syntax tree
  - Usually stops before instruction scheduling/register allocation
  - Examples: AST, CFG, DDG, PDG, etc.
- There is no *one* Intermediate Representation (IR)
  - although people have tried ...

## Mid-end structure

- Organized as a set of *passes*
- Each pass usually performs one task
  - Some specific analysis of the IR
  - Some transformation of the IR
- Input to each pass is the IR and output is also the IR
  - And usually analysis results, etc.

## Running example

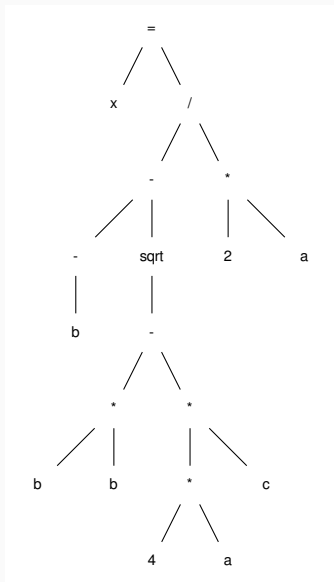
$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

In Python (assume `math.sqrt` is `sqrt`):

```
x = (-b - sqrt(b*b - 4 * a * c)) / (2*a)
```



# Abstract Syntax Tree



# AST as a list

```
ast = ['=',  
      ['x',  
       ['/',  
        ['-',  
         ['neg', 'b'],  
         ['sqrt',  
          ['-',  
           ['*', 'b', 'b'],  
           ['*',  
            ['*', '4', 'a'],  
            'c']  
          ]  
        ]  
       ]  
      ],  
      ['*', '2', 'a']  
     ]  
    ]  
   ]
```

- What does this remind you of?

## Slightly less LISPy

```
class Node(object):  
    operator = None  
    left = None  
    right = None
```

- Actual ASTs are not binary trees!
  - Will usually have list of descendants instead of left and right
  - descendants may be more specific, while may have condition and body

# Linear Forms

- ASTs imply treewalking
  - Works best when manipulating source code
  - e.g. Source-to-source compilers
  - Or when control flow is not important
- Other lower level forms are “closer to machine”
  - Stack machines
  - 3 address code

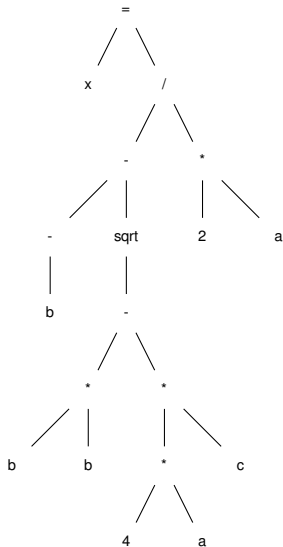
# Stack Machine

```
push a
push 2
mul
push c
push a
push 4
mul
mul
push b
push b
mul
sub
sqrt
push b
neg
sub
div
pop x
```

How do you produce stack machine code from an AST?

# Generating Stack Machine Code

```
push a
push 2
mul
push c
push a
push 4
mul
mul
push b
push b
mul
sub
sqrt
push b
neg
sub
div
pop x
```



# Stack machines

- Compact in size
  - Operands are implicit – on top of stack.
- Easy to execute
- BUT, fixed order of execution
  - Bad for parallelism
- Hard to analyze
- Nevertheless, widely used:
  - Java bytecode
  - Python bytecode
  - WebAssembly

## 3 address code

- 3 “addresses”
  - Two source operands
  - One destination operand
  - One operation
- Addresses are actually *names* generated by compiler
  - Or refer directly to variables



## Our example in 3-address code

```
t1 <- -b  
  
t2 <- b * b  
t3 <- 4 * a  
t4 <- t3 * c  
t5 <- t2 - t4  
t6 <- sqrt(t5)  
  
t7 <- t1 - t6  
t8 <- 2 * a  
t9 <- t7 / t8  
  
x <- t9
```

(Here ' $\leftarrow$ ' is  $\leftarrow$ , signifying assignment)

How do we produce 3-address code from the AST?

# Producing 3-address code

```
t1 <- -b
```

```
t2 <- b * b
```

```
t3 <- 4 * a
```

```
t4 <- t3 * c
```

```
t5 <- t2 - t4
```

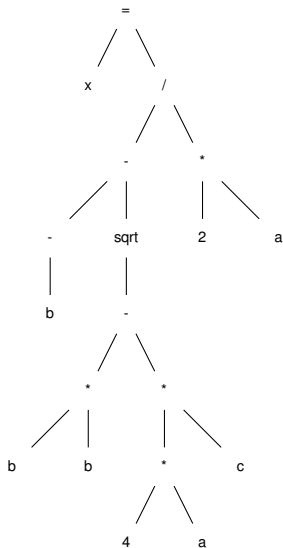
```
t6 <- sqrt(t5)
```

```
t7 <- t1 - t6
```

```
t8 <- 2 * a
```

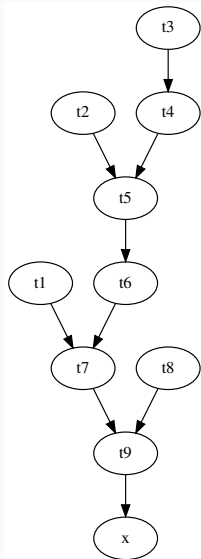
```
t9 <- t7 / t8
```

```
x <- t9
```



# Data Dependence Graphs (DDGs)

- DDGs track “data flow” as an acyclic graph
- Strict (partial) order in which operations must be performed
  - Can’t use a value that has not been calculated yet!
- But multiple orders may be allowed!
  - Topological sort
- Will revisit DDGs when we discuss instruction scheduling



# DDG Example

```
t1 <- -b
```

```
t2 <- b * b
```

```
t3 <- 4 * a
```

```
t4 <- t3 * c
```

```
t5 <- t2 - t4
```

```
t6 <- sqrt(t5)
```

```
t7 <- t1 - t6
```

```
t8 <- 2 * a
```

```
t9 <- t7 / t8
```

```
x <- t9
```

# DDG Example, reordered by dependence

```
# group 1
t1 <- -b
t2 <- b * b
t3 <- 4 * a
t8 <- 2 * a

# must wait for t3
t4 <- t3 * c

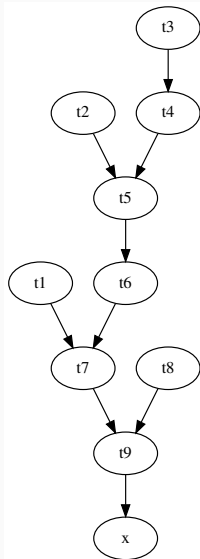
# must wait for t4 and t2
t5 <- t2 - t4

# ...
t6 <- sqrt(t5)

t7 <- t1 - t6

t9 <- t7 / t8

x <- t9
```

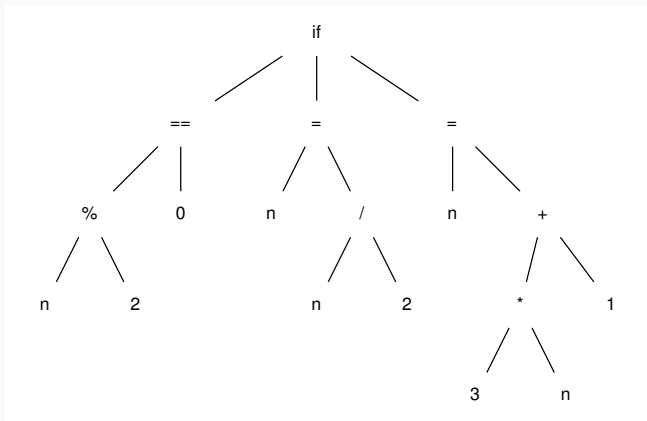


## Control structures in 3-address code

What should the 3 address code for the code below look like?

```
if n % 2 == 0:  
    n = n / 2  
else:  
    n = 3 * n + 1
```

## The AST for if



- An if AST node has a condition, true-code, and false-code

## 3-Address Code for if

```
t1 <- n % 2
tc <- t1 == 0
if (tc == 0) goto L1
```

```
t2 <- n / 2
n <- t2
goto L2
```

L1:

```
t3 <- 3 * n
t4 <- t3 + 1
n <- t4
```

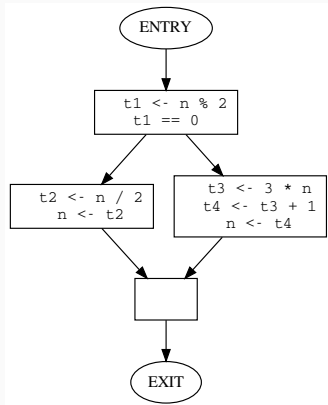
L2:

- 3-address code can contain:
  - conditional branches, usually just a comparison to zero
  - unconditional branches
  - labels



# Control Flow Graphs (CFGs)

- “Hybrid” representation
  - Linear code + Graph structure
- Each node in the CFG is a “basic block”
  - Linear code
  - Single entry, single exit
  - “Straight-line code”
- Most common form for analysis



## In this course

- We will write midend + some bits of a backend
- Input language: C
- Output language: C in 3-address form
  - Not assembly (maybe extra credit?)
- Using Python library `pyparser`

## For assignments, make sure to review

- Basic data structures
  - lists
  - trees
  - graphs
- Basic data structure traversals
  - Infix, prefix, postfix
  - Depth-first, breadth-first
- And how to implement them in Python
  - Using Python standard libraries is fine

# Outline

Introduction

Midend

Miscellaneous

# Not-so-classic 'Compiler' Architectures

What is this code from TensorFlow doing?

```
a = tf.constant(2)
b = tf.constant(3)

with tf.Session() as sess:
    print("a=2, b=3")
    print("Addition with constants: %i" % sess.run(a+b))
    print("Multiplication with constants: %i" % sess.run(a*b))
```

[https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/1\\_Introduction/basic\\_operations.py](https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/1_Introduction/basic_operations.py)

# Metaprogramming

- TensorFlow API builds a graph
  - directed, acyclic
  - similar to the DDG
  - very common technique
- When `sess.run` is called, graph is compiled and executed
- Advantages:
  - No syntax, no parsing!
- Disadvantages:
  - ?

# What about functional programming languages?

- General problem is to represent control transfers as function calls
- Continuation Passing Style
  - Each basic block is a function that accepts a *continuation*, another function
  - At end of each basic block/function, the continuation is called
- Later we'll learn about SSA which is easy to transform into this style
  - See Appel (1998), SSA is functional programming\*
- Another representation is *A-normal Form*
  - Cormac, Sabry, Duba, Felleisen (1993), The Essence of Compiling with Continuations
  - Interesting usage: McMahan et al. (2017), An Architecture Supporting Formal and Compositional Binary Analysis

# References

- Chapter 5 of Cooper and Turczon
  - Up to 5.4 in this lecture, but we will ultimately study the whole chapter
- Note that \* next to paper citations means that the paper is in the syllabus for 4xx students