# Compiler
# Code Generation

COMP360

*"Students who acquire large debts putting themselves through school are unlikely to think about changing society. When you trap people in a system of debt, they can't afford the time to think."*

Noam Chomsky

# Snowflake Parser

- A recursive descent parser for the Snowflake language is due by noon on Friday, February 17, 2017
- This parser should work with your Snowflake scanner
- The parser must determine if the input Snowflake program has proper syntax
- An appropriate error message (showing line and column) must be displayed if there is a syntax error

# Exam 1

- The first COMP360 exam will be on Friday, February 17, 2017
- You are allowed to have one 8½ by 11” page of notes

# Stages of a Compiler

- Source preprocessing
- Lexical Analysis (scanning)
- Syntactic Analysis (parsing)
- Semantic Analysis
- Optimization
- Code Generation
- Link to libraries

# Lexical Analysis

- Lexical Analysis or scanning reads the source code (or expanded source code)
- It removes all comments and white space
- The output of the scanner is a stream of tokens
- Tokens can be words, symbols or character strings
- A scanner can be a finite state automata

# Parsing

- Syntactic Analysis or parsing reads the stream of tokens created by the scanner
- It checks that the language syntax is correct
- The output of the Syntactic Analyzer is a parse tree
- The parser can be implemented by a context free grammar stack machine

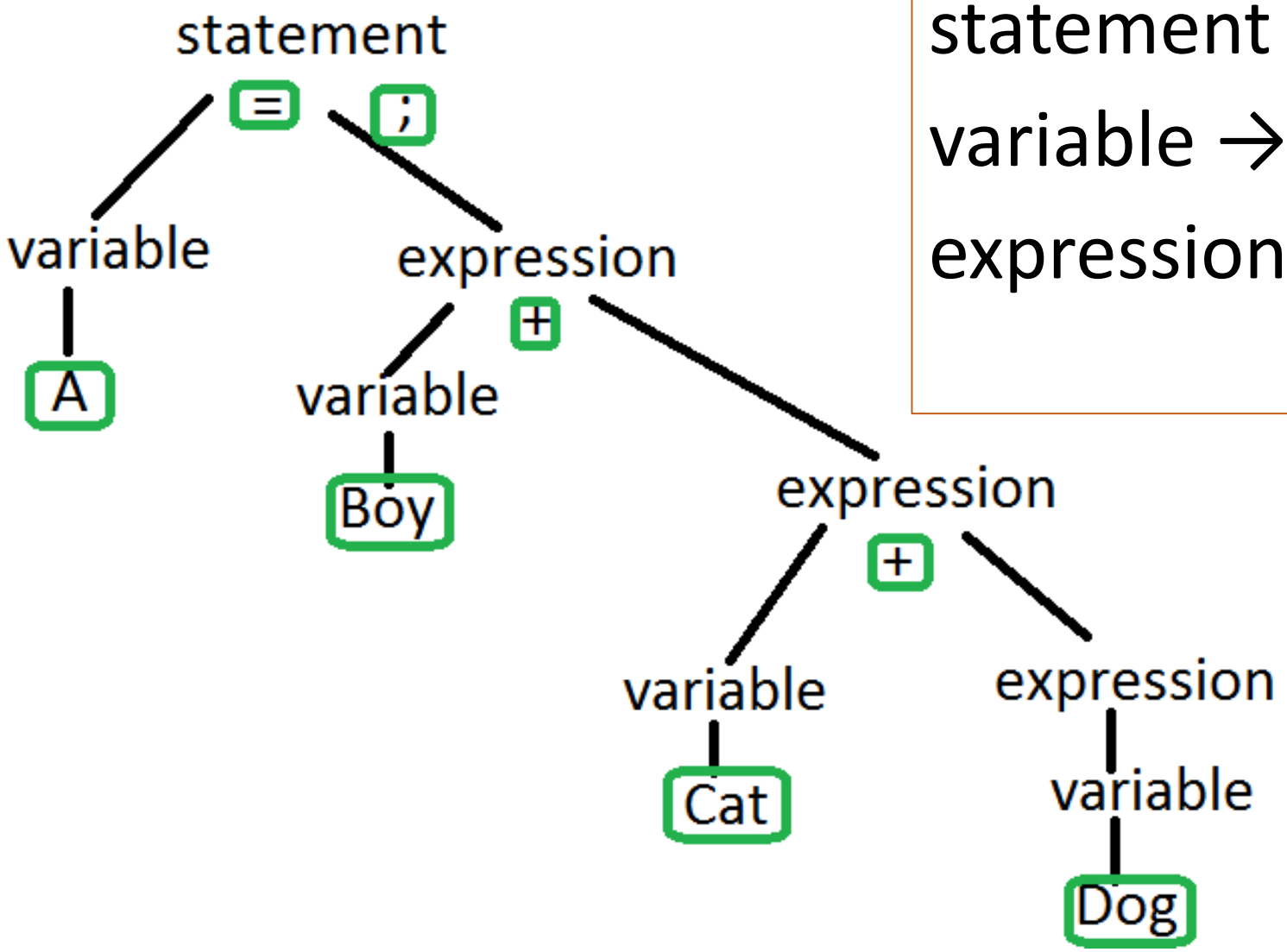
# Semantic Analysis

- The Semantic Analysis inputs the parse tree from the parser
- Semantic Analysis checks that the operations are valid for the given operands *(i.e. cannot divide a String)*
- This stage determines what the program is to do
- The output of the Semantic Analysis is an intermediate code. This is similar to assembler language, but may include higher level operations

# Creating a Parse Tree

- The goal of the parser is to create a parse tree whose leaves are the tokens from the lexical scanner when traversed left to right
- In addition to proving that a tree does or does not exist, the parser should actually build the tree
- The semantic analyzer will use the tree to create executable code

# Parsing




statement → variable **=** expression **;**

variable → *string of characters*

expression → variable **+** expression

| variable

# Semantic Analysis

- Semantic analysis will check
- Intermediate code defines a series of simple steps that will execute the program

| Operation | First Operand | Second Operand | Destination |
|---|---|---|---|
| add | Boy | Cat | temp1 |
| add | temp1 | Dog | temp2 |
| copy | temp2 | | A |

# Simple Machine Language

- Load register with B
- Add C to register
- Store register in Temp1
- Load register with Temp1
- Add D to register
- Store register in Temp2
- Load register with Temp2
- Store register in A

# Optimized Machine Language

- Load register with B
- Add C to register
- Store register in Temp1
- Load register with Temp1
- Add D to register
- Store register in Temp2
- Load register with Temp2
- Store register in A

# Action Rules

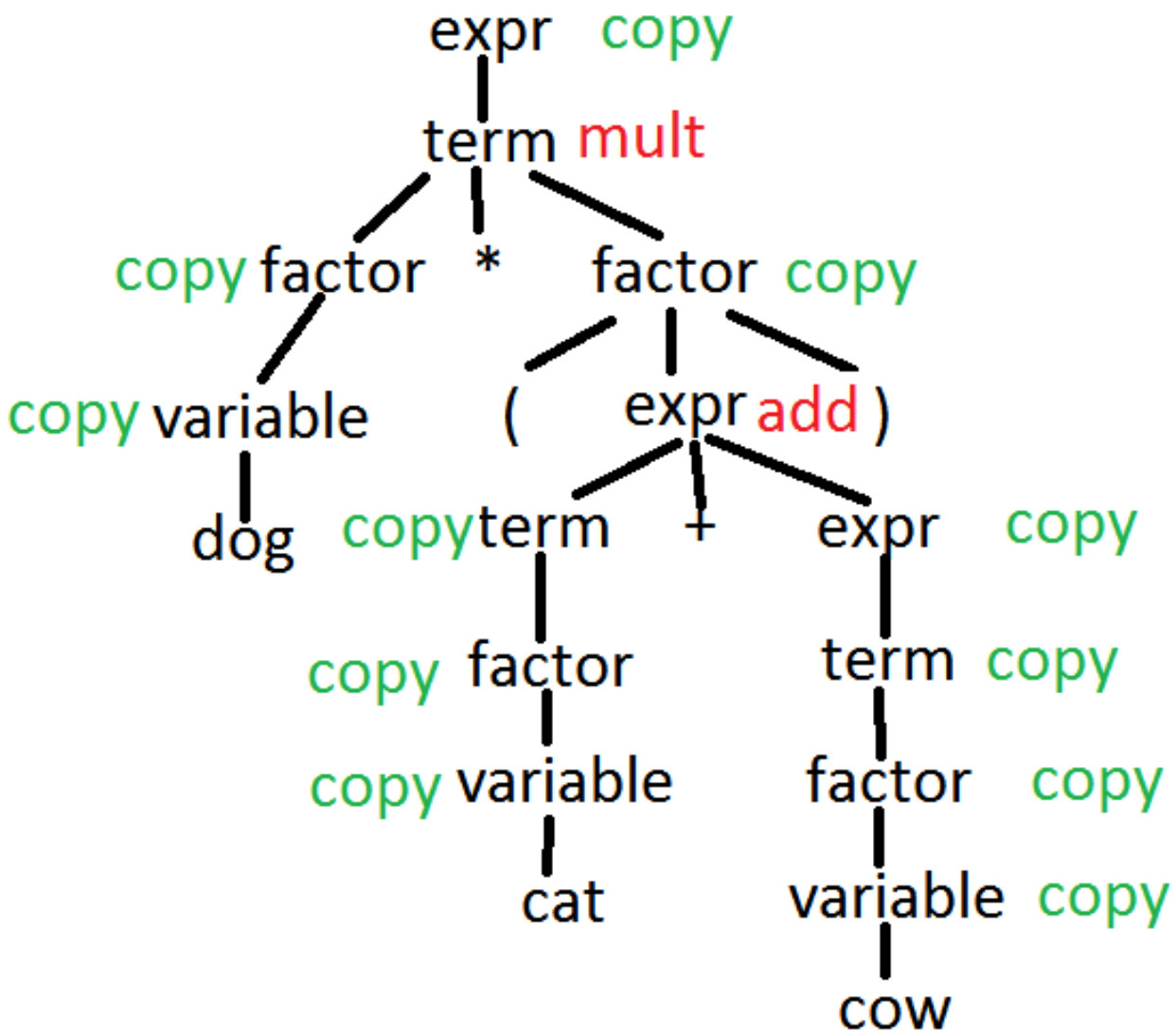
- Action rules are applied to the BNF productions to indicate what action needs to be taken to execute that portion of the program
- Action rules can either copy a value from a lower production in the tree or compute a function on the lower tree values
- Action rules can only involve lower items on the tree

# Example Grammar with Actions

| | | |
|---|---|---|
| 1. | expr → term | expr = term |
| 2. | expr → term **+** $expr_2$ | expr = sum( term, $expr_2$) |
| 3. | expr → term **-** $expr_2$ | expr = diff( term, $expr_2$) |
| 4. | term → factor | term = factor |
| 5. | term → factor * $factor_2$ | term = mult(factor,$factor_2$) |
| 6. | term → factor **/** $factor_2$ | term = div(factor,$factor_2$) |
| 7. | factor → **variable** | factor = variable |
| 8. | factor → **(** expr **)** | factor = expr |

# Annotated Parse Tree Example

dog * ( cat + cow )



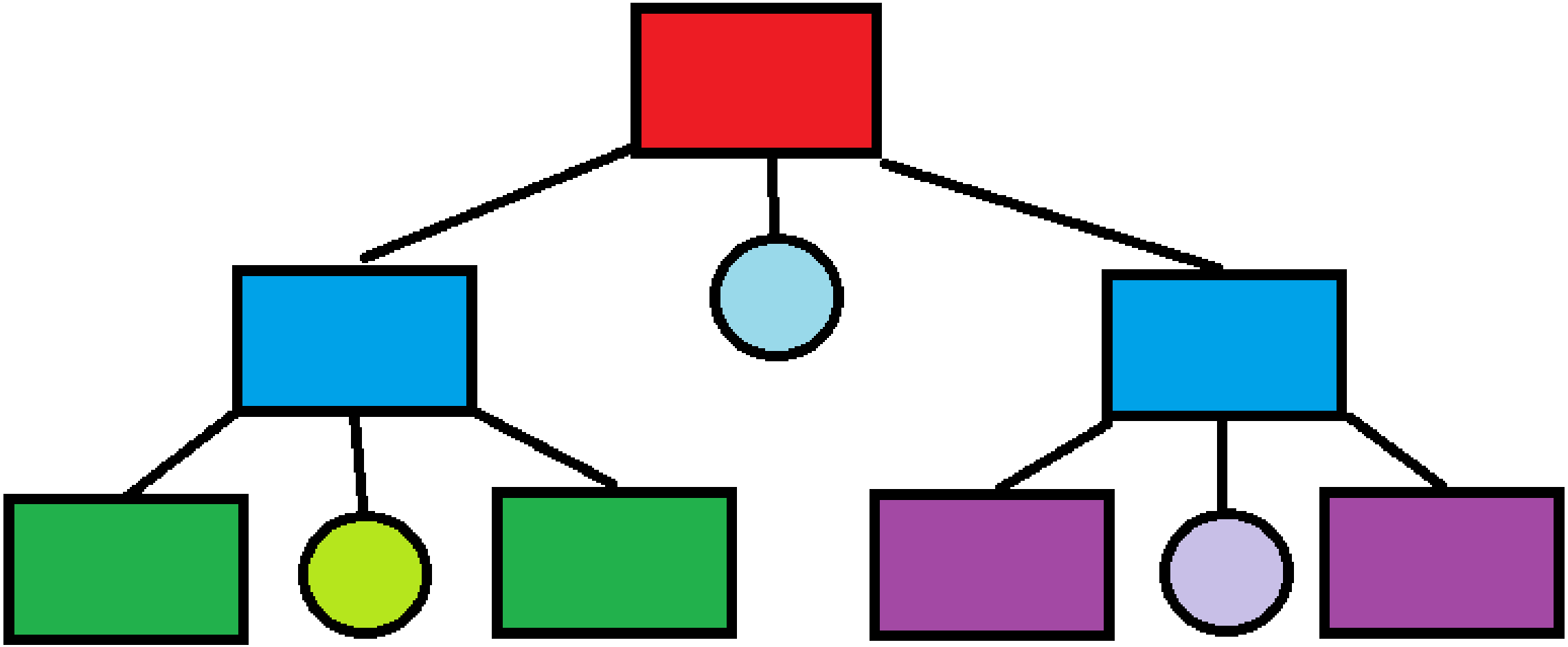
# Postfix Traversal

- Traversing the parse tree in a postfix order, taking the specified actions, will correctly execute the program
- After the parser has built a parse tree, the semantic analyzer can traverse the tree in postfix order
- Executable code can be generated in the order found by a postfix traversal

# Data Structures Review

- Assume we have nodes that have a left operand, operator and right operand
- Some nodes may only have one operand and no operator (copy action)
- An **infix** traversal visits the left node, the operation and then the right node
- A **prefix** traversal visits the operation, the left node and then the right node
- A **postfix** traversal visits the left node, the right node and then the operation

# Example Tree

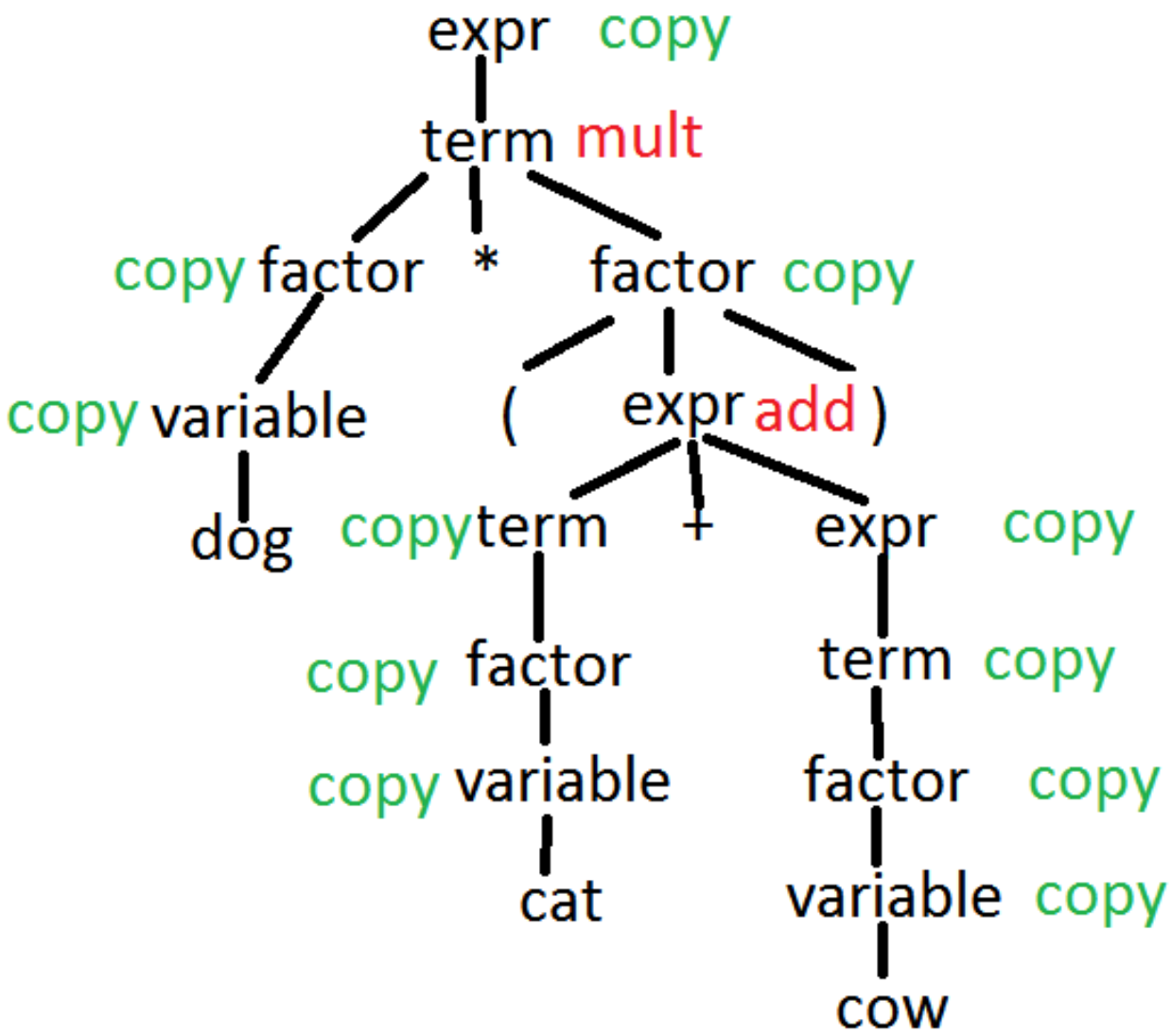


# Postfix Traversal Example

dog * ( cat + cow )

Get dog
Get cat
Get cow
temp1 = cat + cow
temp2 = dog * temp1

*copies omitted*



# Loop Invariants

- Calculations inside a loop that do not change in the loop, can be moved before the loop.
- This can make the program much more efficient if the calculation moved out of the loop require a lot of CPU time

```
for (int i=0; i<n; i++) {
   x = y+z;
   a[i] = 6*i + x*x;
}
```

```
x = y+z;
temp = x*x;
for (int i=0; i<n; i++) {
   a[i] = 6*i + temp;
}
```

from Wikipedia

# Peephole Optimization

- Peephole optimization is a kind of optimization performed over a very small set of instructions in a segment of generated code
- This contrasts with loop optimizations that have to loop at the entire loop

# Common peephole optimization

- Constant folding – Evaluate constant subexpressions in advance
- Strength reduction – Replace slow operations with faster equivalents
- Null sequences – Delete useless operations
- Combine operations – Replace several operations with one equivalent
- Algebraic laws – Use algebraic laws to simplify or reorder instructions

from Wikipedia

# Constant Simplification

- A statement may include arithmetic with constants that can be computed at compile time

```
volume = 4.0 / 3.0 * 3.14159 * r * r;
```

Can be simplified to

```
volume = 4.18879 * r * r;
```

# Constant Propagation

- Constants can be carried to the next statements

```
int x = 14;
int y = 7 - x / 2;
return y * (28 / x + 2);
```

- Propagating x yields:

```
int x = 14;
int y = 7 - 14 / 2;
return y * (28 / 14 + 2);
```

- Further simplifying to

```
return 0;
```

from Wikipedia

# Strength Reduction

- strength reduction is where expensive operations are replaced with equivalent but less expensive operations

- There are multiple ways to divide by 8

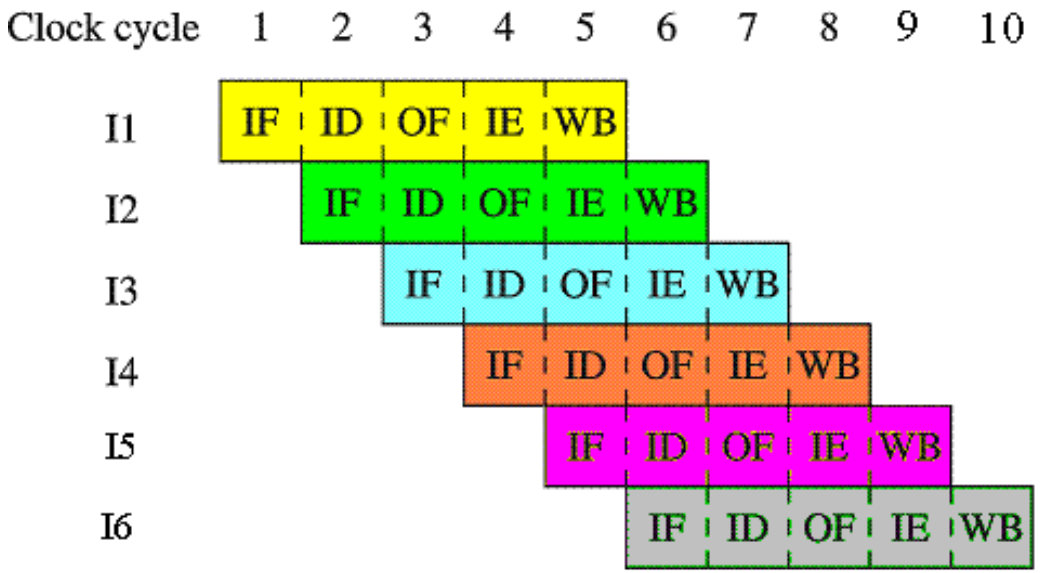
```
divide      R3, 8
shiftRight  R3, 3       // if integer
multiply    R3, 0.125   // if double
```

# Assembly Line

- Pipelining is like an assembly line.  Each stage of the execution cycle performs its function and passes the instruction to the next cycle
- Each stage can be working on a different instruction at the same time
- Several instructions can be in the process of execution simultaneously
- There is no reason to keep the ALU idle when you are fetching or storing an operand

# Pipelining


Serial execution




Pipelined execution

# Hazards

- A hazard is a situation that reduces the processors ability to pipeline instructions
- **Resource** – When different instructions want to use the same CPU resource
- **Data** – When the data used in an instruction is modified by the previous instruction
- **Control** – When a jump is taken or anything changes the sequential flow

# Avoiding Data Hazards

- The compiler can generate code that avoids data hazards
- Interleaving different parts of an expression or the program can space access to data values to avoid data hazards
- Using multiple registers instead of always the same register helps reduce data hazards

# Pipeline Efficient Code

Inefficient

```
Load   R1, W
Add    R1, 7
Store  R1, W
Load   R1, X
Add    R1, 8
Store  R1, X
Load   R1, Y
Add    R1, 9
Store  R1, Y
Load   R1, Z
Add    R1, 10
Store  R1, X
```

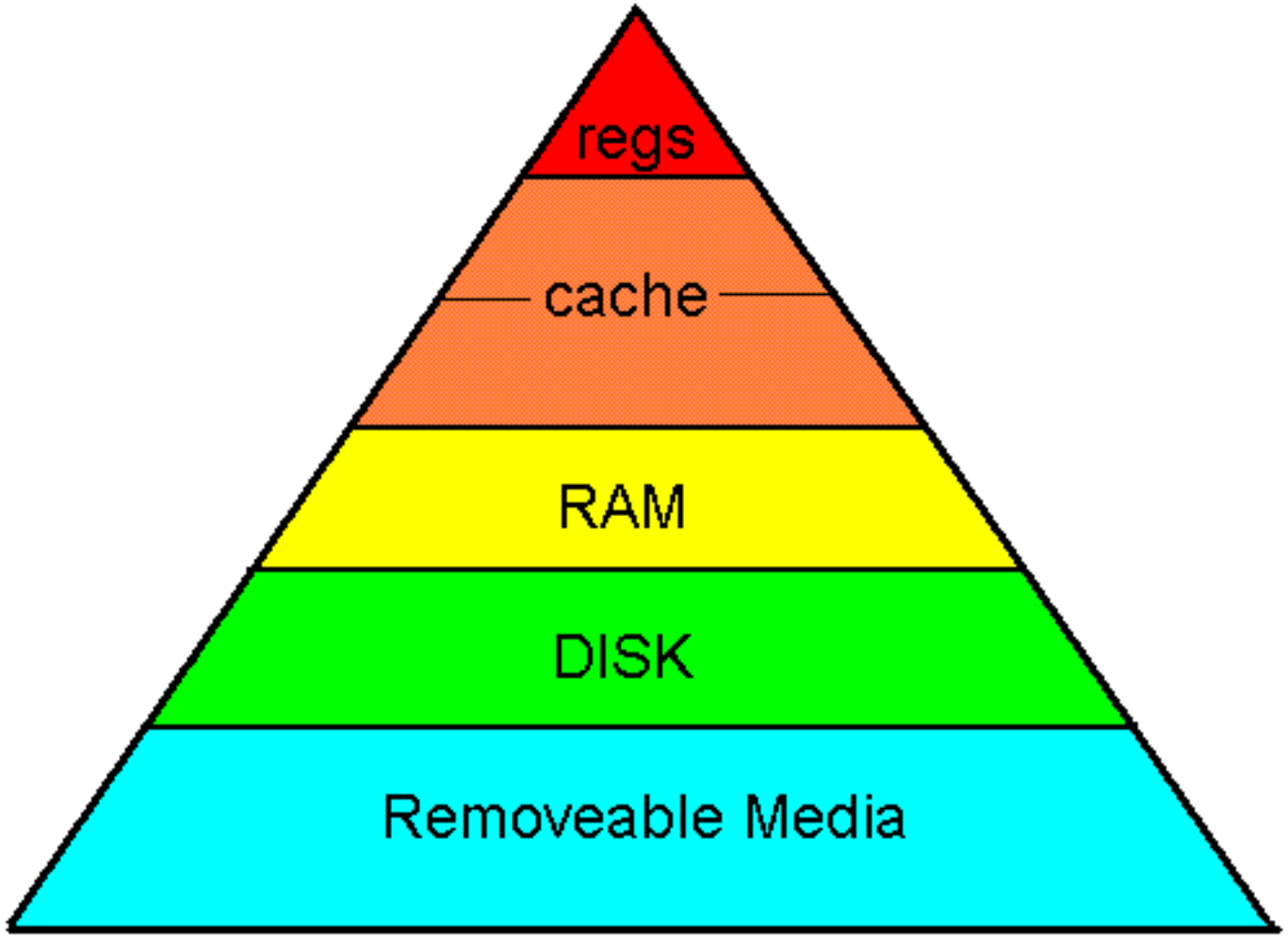
Pipeline Efficient (4 stage pipeline)

```
Load   R1, W
Load   R2, X
Load   R3, Y
Load   R4, Z
Add    R1, 7
Add    R2, 8
Add    R3, 9
Add    R4, 10
Store  R1, W
Store  R2, X
Store  R3, Y
Store  R4, Z
```

# Register Allocation

- Keeping values in the registers instead of memory makes the program run much faster
- Many computers have 16 or 32 general purpose registers allowing several values to be kept in a register
- Finding the optimal register allocation is equivalent to the NP-Complete graph coloring problem

# Memory Hierarchy



# Exam 1

- The first COMP360 exam will be on Friday, February 17, 2017
- You are allowed to have one 8½ by 11” page of notes