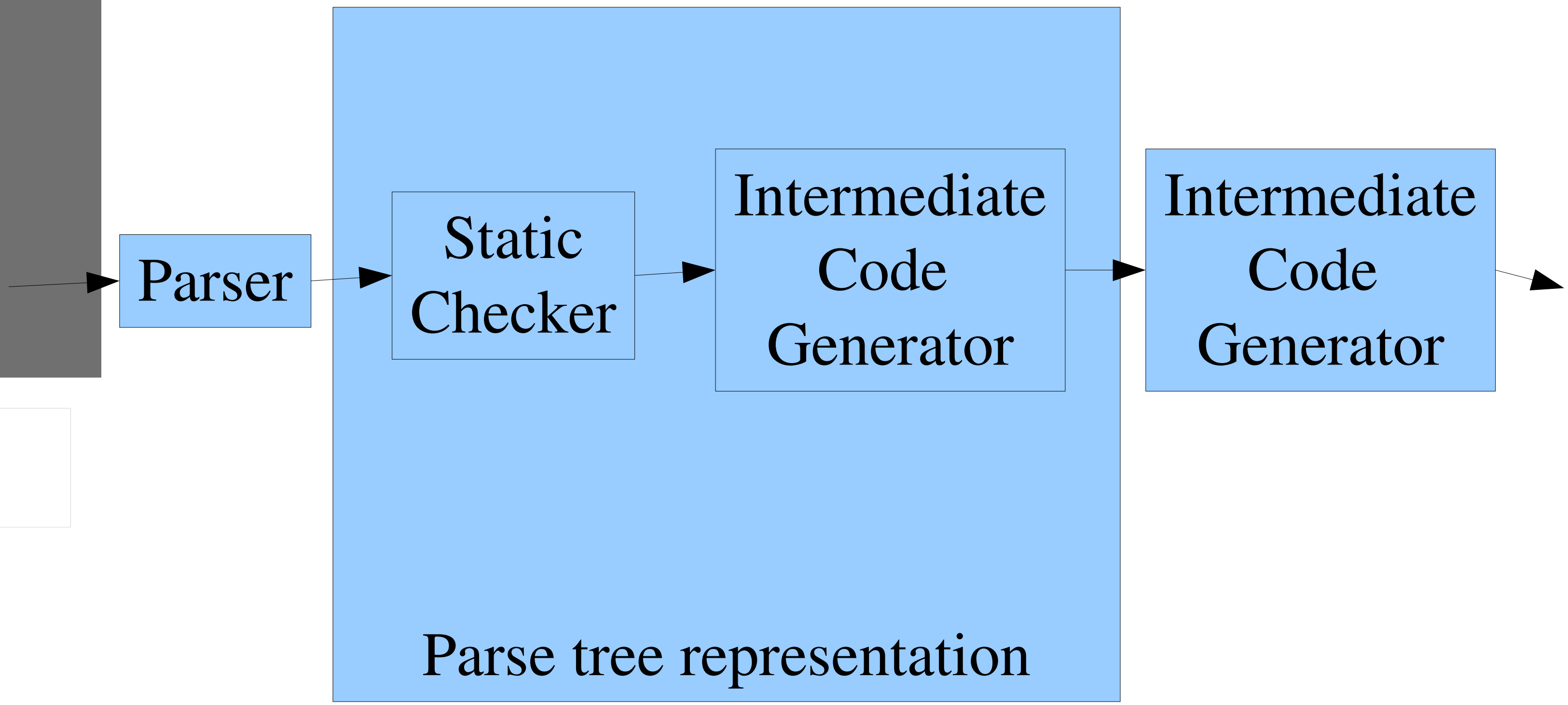
Static Checking and Intermediate Code Generation

Pat Morin

COMP 3002

Static Checking and Intermediate Code Generation



Why Static Checking?

- Parsing finds *syntactic* errors
  - An input that can't be derived from the grammar
- Static checking finds *semantic* errors
  - Calling a function with the wrong number/kind of arguments
  - Applying operators to the wrong kinds of arguments
  - Using undeclared variables
  - Warnings about common errors
    - `if (a = b) { ... }`
  - Invalid conditions (not boolean) in conditionals
  - Instantiation of virtual classes
  - inappropriate instruction
    - return, break, continue used in wrong place
  - ...

Why Static Checking?

- Parsing finds *syntactic* errors
  - An input that can't be derived from the grammar
- Static checking finds *semantic* errors
  - Calling a function with the wrong number/kind of arguments
  - Applying operators to the wrong kinds of arguments
  - Using undeclared variables
  - Warnings about common errors
    - `if (a = b) { ... }`
  - Invalid conditions (not boolean) in conditionals
  - Instantiation of virtual classes
  - inappropriate instruction
    - return, break, continue used in wrong place
- Typechecking errors

The Need for Type Inference

- We want to generate machine code
- Memory layout
  - Different data types have different sizes
    - In C, char, short, int, long, float, double usually have different sizes
    - Need to allocate different amounts of memory for different types
- Choice of instructions
  - Machine instructions are different for different types
    - add (for i386 ints)
    - fadd (for i386 floats)

Type Checking

- One important kind of static checking is type checking
  - Do operators match their operands?
  - Do types of variables match the values assigned to them
  - Do function parameters match the function declarations
  - Have called function and variable names been declared?
- Not all languages can be completely type checked
- All compiled languages must be at least partially type checked

Type Checking (Cont'd)

- Type checking can be done bottom up using the parse tree
- For convenience, we may create one or more pseudo-types for error handling purposes
  - Error type can be generated when a type checking error occurs
    - e.g., adding a number and a string
  - Unknown type can be generated when the type of an expression is unknown
    - e.g., an undeclared variable

Type Checking Operators

- For each operator, create a table
  - TypeA op TypeB = TypeC
- This allows us to assign a type to an operation if we know the types of its operands

+	**String**	**Number**	**Boolean**	**Error**
String	String	String	String	String
Number	String	Number	Error	Number
Boolean	String	Error	Error	Boolean
Error	String	Number	Boolean	Error

Type Checking Function Calls

- To type-check function calls we need to
  - Check that the arguments to a function match the function's declaration
- The return type of a function call is specified by its declaration

Determining Types of Constants

- Determining the types of constants is usually done by the tokenizer
- The type of a constant determines the type of the node in the parse tree

Determining the Types of Variables

- To determine the type of a variable, we need to keep track of the current environment.
- Usually, an environment is a stack of *frames*, where each frame maps variable names onto types
  - Starting a new code block or new function definition creates a new frame
  - Closing a code block pops a frame
  - Declaring a variable or function adds a new mapping to the current frame

Environment Example

- Show the environment at lines 0, 2, 4, 6, and 8

```
0
1 int x, y;
2
3 if (x > y) {
4   int p = x * y;
5 } else {
6   int q = x + y;
7 }
8
9
```

Object-Oriented Languages

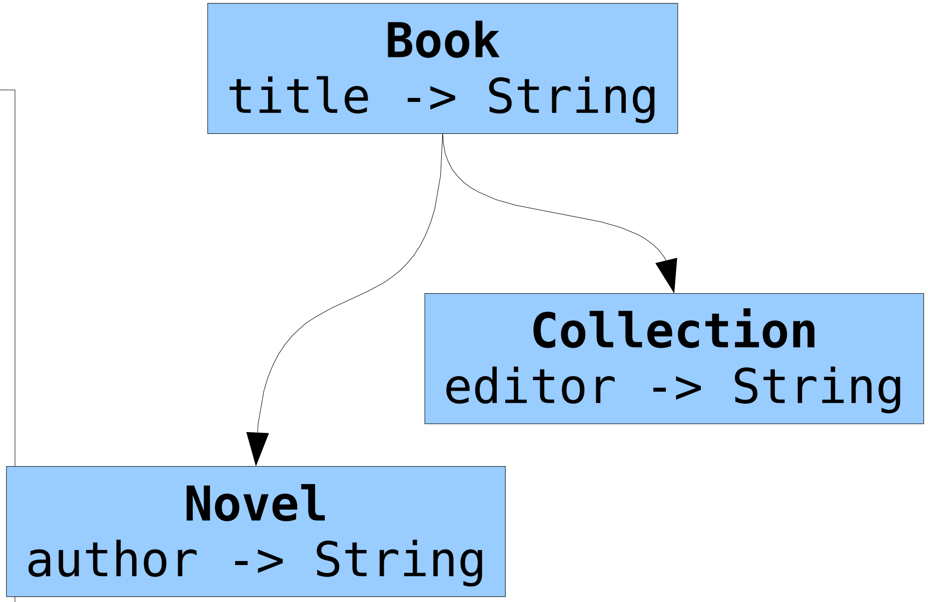
- Object-oriented languages are a little more complicated
- In addition to the usual environment, there is an environment containing all the object's variables and methods
- And objects inherit environments from their superclasses.
- Typically use two environments, one for the object and one usual environment
  - The object environments are organized according to the inheritance tree

OO Environment Examples

```
class Book {
  String title;
};

class Novel
  extends Book {
  String author;
}

class Collection
  extends Book {
  String editor;
}
```



OO Type Inference

- To identify the type of a variable, we usually
  - Look first in the usual environment
  - Next look in the object environment
- Many OO languages provide a method of scope resolution

```
class Book {
  String title;

  public Book(String title) {
    this.title = title;
  }
}
```

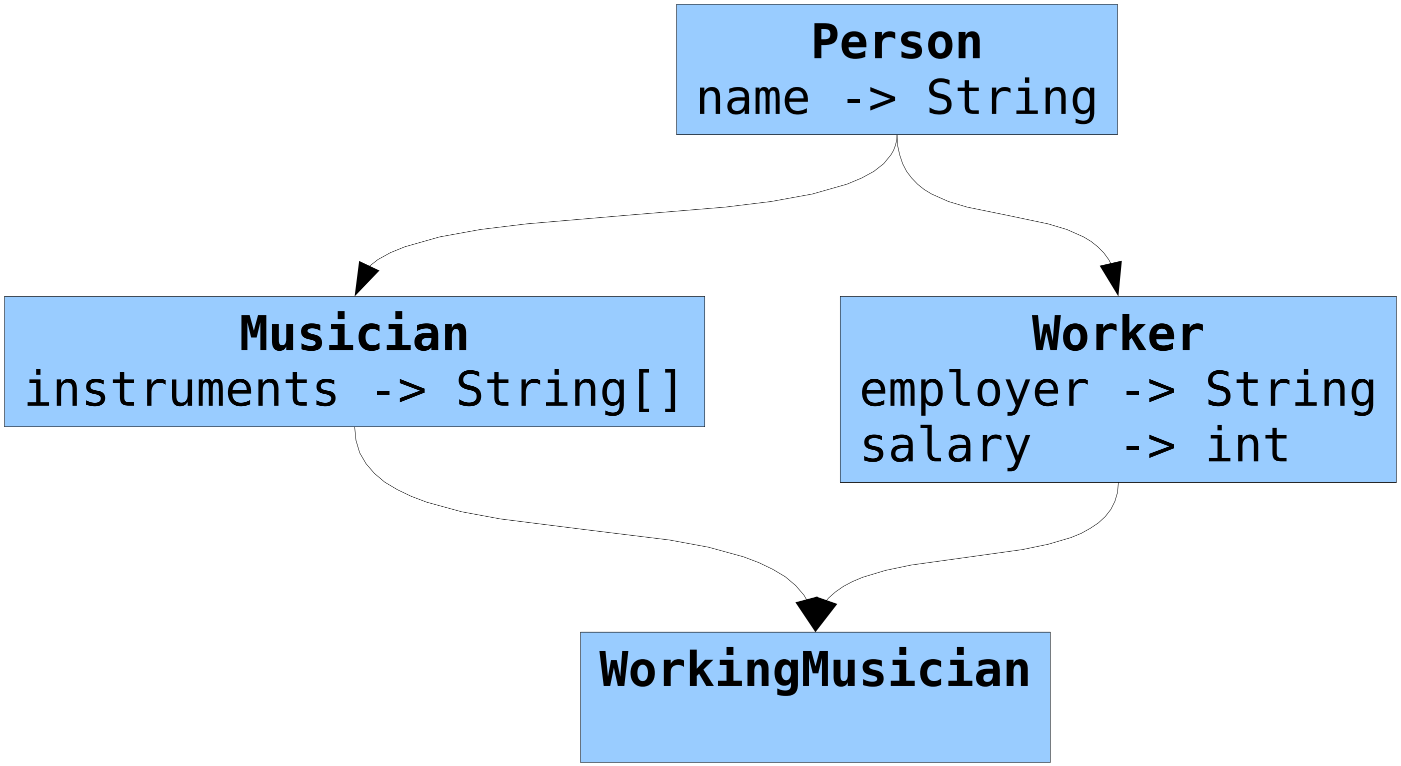
Scope Resolution (C++ style)

```
class Book {
  String title;
}

class Collection extends Book {
  String title;

  Collection (String title) {
    this.title = title;
    Book::title = title + " (collected works)";
  }
}
```

Multiple Inheritance



- Object environment becomes more complex

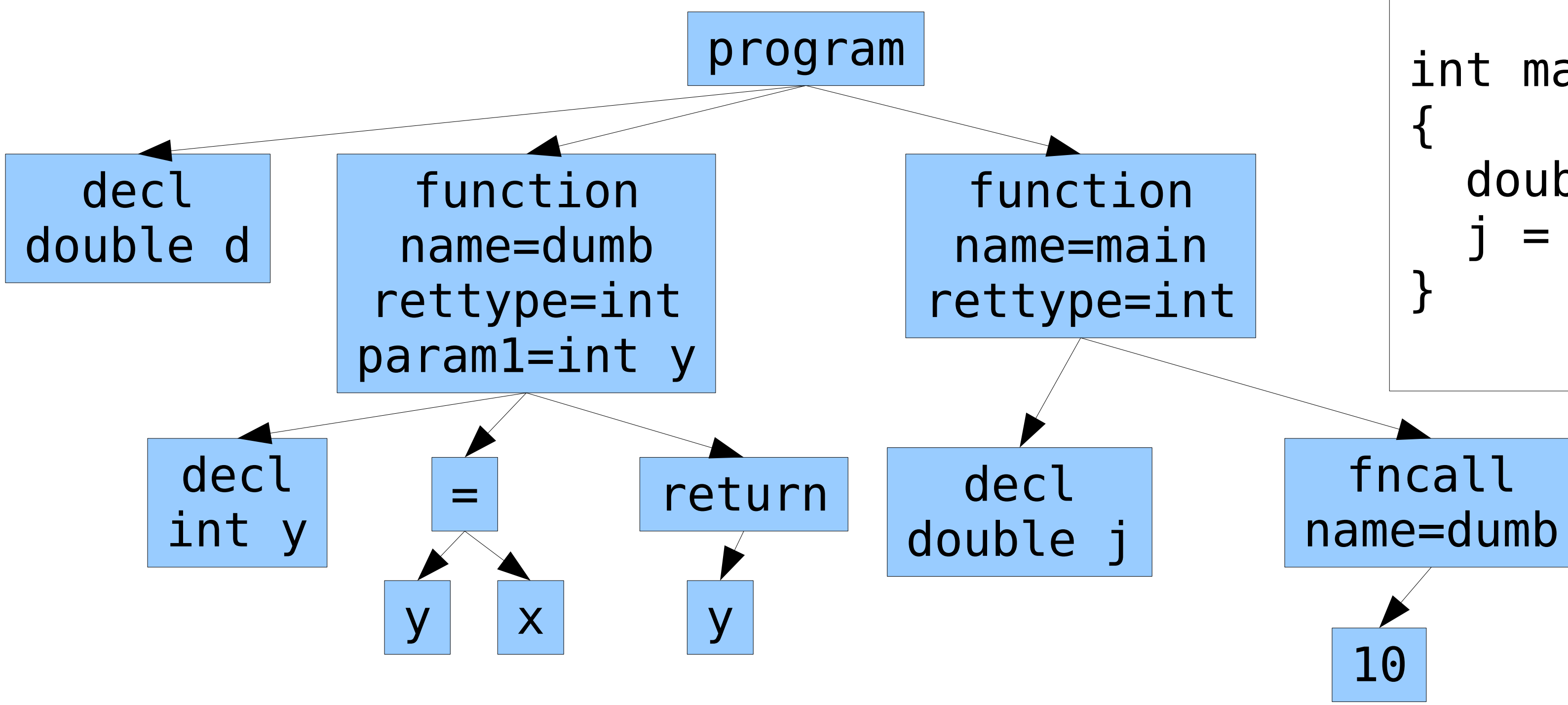
Typechecking Return Values

- Functions should only return values of the correct type
- This is easily checked by introducing a pseudovariable __retval to the function's environment whose type is the function's return type
- Return statements should check that the returned value matches the type of __retval

```
double d;

int dumb(int x)
{
  int y;
  y = x;
  return y
}

int main()
{
  double j;
  j = dumb(10);
}
```



Type Checking Summary

- A type checker includes
  - Rules for deriving the types of operators given the types of their operands
  - Mapping from constant tokens onto types
  - A mechanism (environments) for matching variables and function names with their declarations to determine their type
- The type inference mechanism gets reused during code generation

Other Static Checks

- A variety of other miscellaneous static checks can be performed
  - Check for return statements outside of a function
  - Check for case statements outside of a switch statement
  - Check for duplicate cases in a case statement
  - Check for break or continue statements outside of any loop
  - Check for goto statements that jump to undefined labels
  - Check for goto statements that jump to labels not in scope
- Most such checks can be done using 1 or 2 traversals of (part of) the parse tree

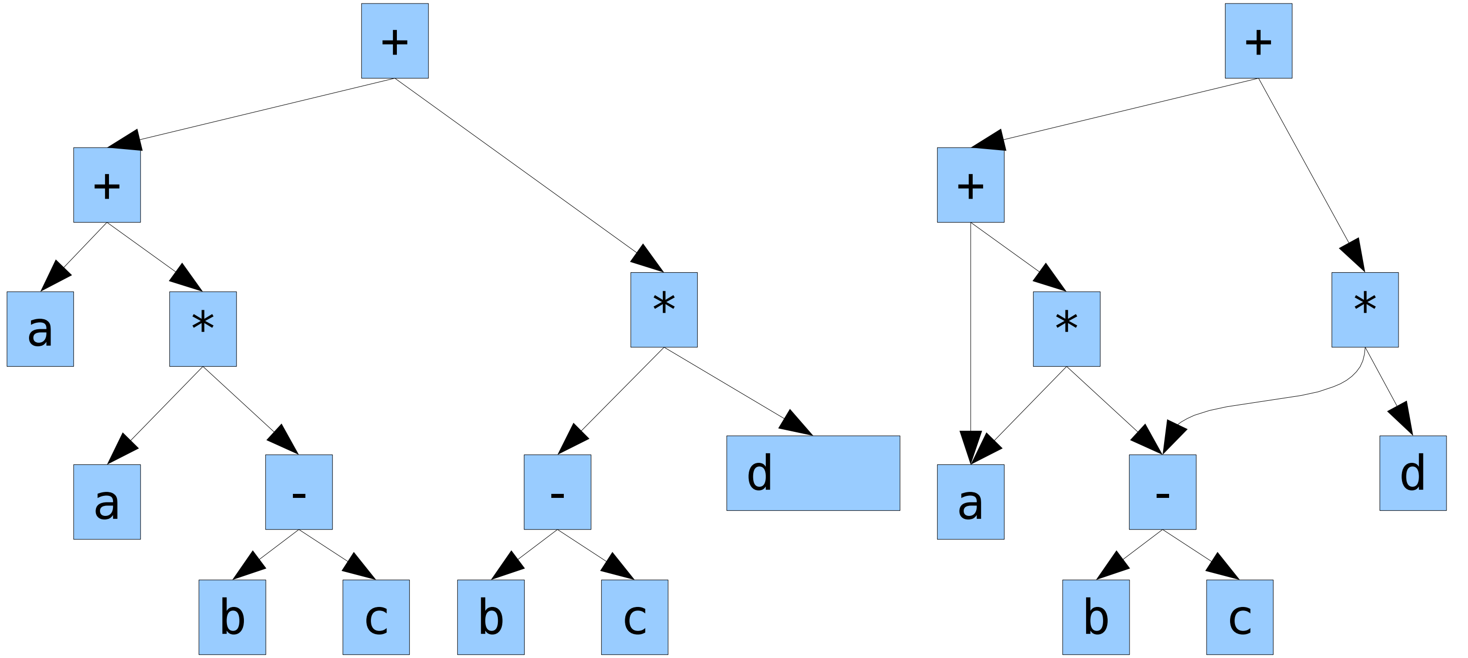
Intermediate Code Generation

- A compiler may have several levels of intermediate code
  - High level intermediate code is simpler
  - Low level intermediate code is closer to machine code
- The choice of intermediate representations varies between compilers
  - Parse tree
  - Assembly-like language (e.g., 3-address codes, and virtual stack machines)
  - High level programming language (e.g., C)

Parse DAGs

- The output of a parser is usually a parse tree
- Often, this can be improved into a more concise and meaningful *directed acyclic graph* (*DAG*)

Parse DAGs



Constructing a Parse Dag

- From a parse tree we can construct a parse DAG using a hash table
- Do a post-order traversal of the parse tree:
  - When encountering a new identifier (leaf node) add it to the hash table, keyed by its name
  - When encountering a new subexpression (internal node) add a new key to the hash table that contains the key of the left child, the operator name, and the key of the right child.
  - Never add the same key to the hash table twice (just point to the existing nodes instead)
- This is most commonly done for simple expressions

Parse DAG Exercises

- Construct the parse DAG for
  - (x+y)-((x+y)*(x-y))
  - ((x1-x2)*(x1-x2))+((y1-y2)*(y1-y2))
- Construct a parse DAG of size *n* that represents a parse tree of size 2^n
- How do parse DAGs interact with operators like ++ and - -?

Directed Acyclic Graphs

- DAG - directed graph with no cycles
- DAGs can represent dependencies between items
- Reversing all the edges of a DAG gives another DAG

Topological Sort

- Processes the nodes of a DAG in order
  - Node i is not processed until all nodes j with edges from j to i have been processed

```
 For each i indeg(i) <- in-degree(i)

Q <- all nodes with no outgoing edges

while Q is not empty
    i = Q.dequeue()
    process(i)
    for each edge i->j
       indeg(j) <- indeg(j) - 1
       if (indeg(j) = 0)
         Q.enqueue(j)
```

Topological Sort Example

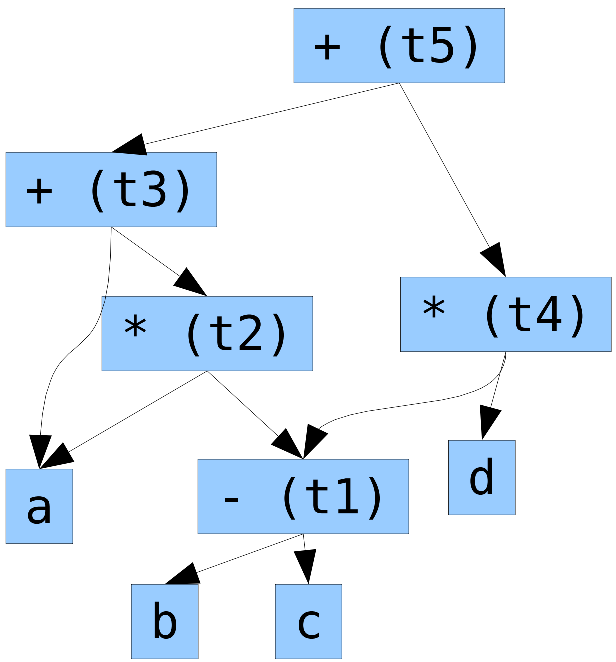
CarletonWide_Tag_K_186

Two Types of Intermediate Representations

- 3-address codes:
  - Each instruction operates on up to 3 addresses
  - An address can be a name, a constant, a label, or a compiler generated temporary variable
- Virtual stack machine
  - We can push and pop items from a stack
  - Various operators operate on the top few items of the stack and leave the result of the operation on the top of the stack
- These may be local to individual function definitions

3-Address Codes for Simple Expressions

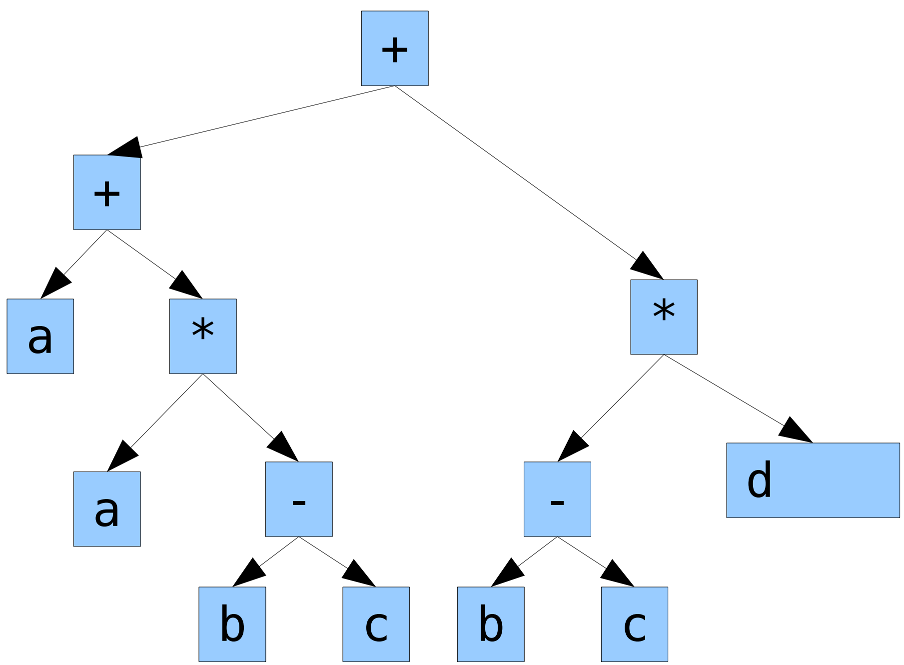
- Traverse the parse tree (or DAG) and assign temporary names to the internal nodes
- Traverse the tree in post-order generating the instructions




```
t1 = b – c
t2 = a * t1
t3 = a * t2
t4 = t1 * d
t5 = t3 * t4
```

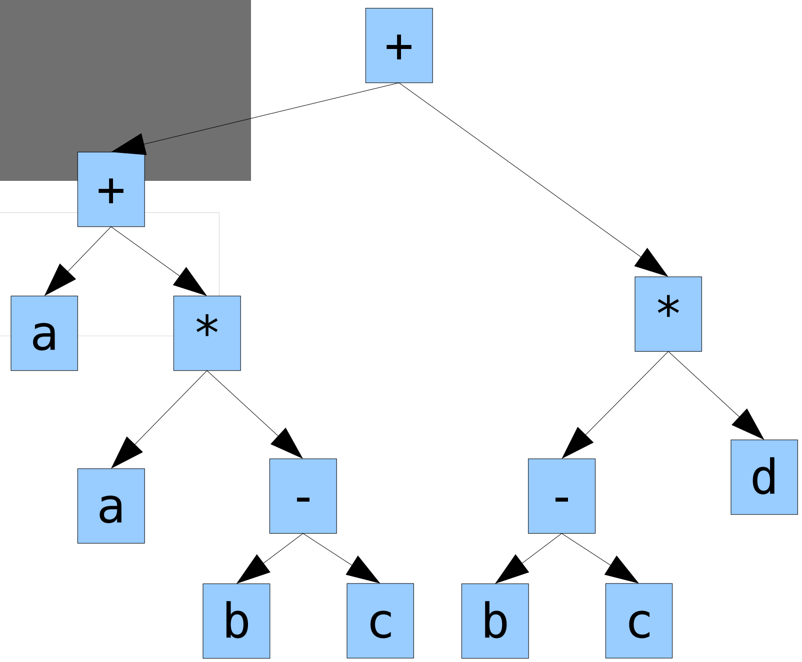
3-Address Code Examples

- Generate the 3-address codes for this parse tree:



Virtual Stack Machine for Simple Expressions

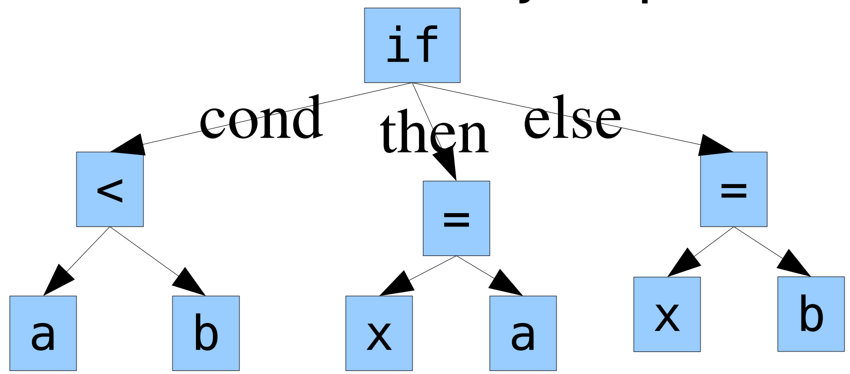
- Traverse the parse tree in post-order, making sure that each node leaves its return value on the stack




```
push a      [a]
push a      [a,a]
push b      [a,a,b]
push c      [a,a,b,c]
subtract    [a,a,b-c]
multiply    [a,a*(b-c)]
add         [a+a*(b-c)]
push b      [a+a*(b-c),b]
push c      [a+a*(b-c),b,c]
subtract    [a+a*(b-c),b-c]
push d      [a+a*(b-c),b-c,d]
multiply    [a+a*(b-c),(b-c)*d]
add         [a+a*(b-c)+(b-c)*d]
```

Conditional Statements

- Conditional statements use conditional and unconditional jump instructions



3AI

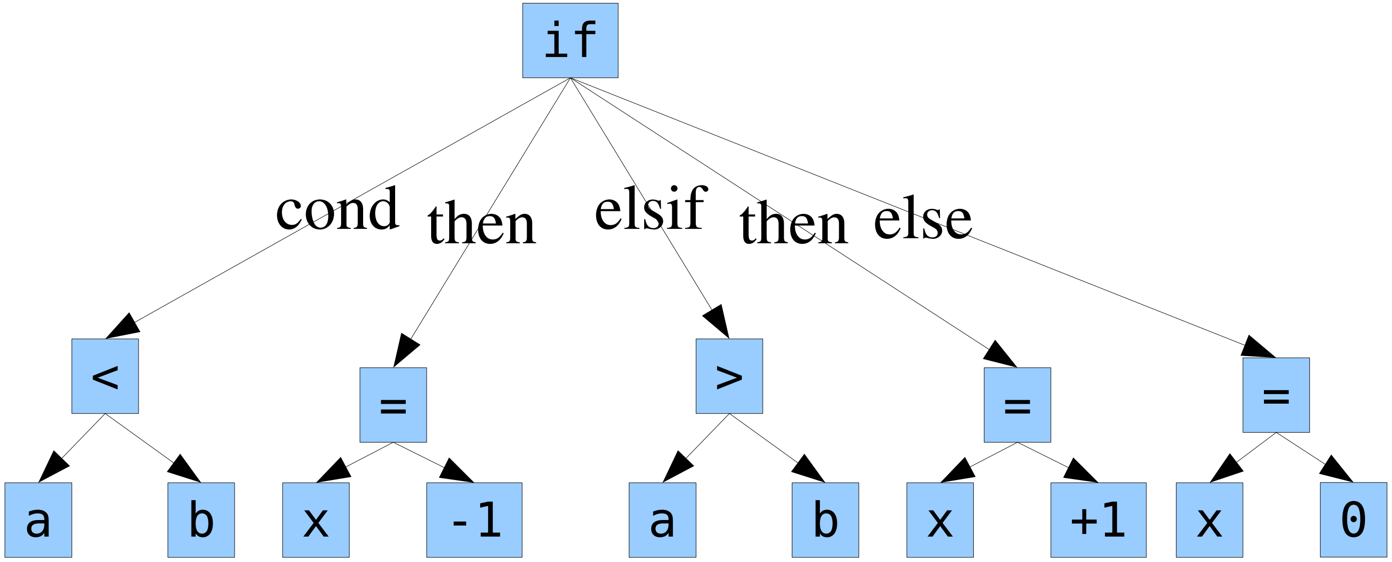
```
    t1 = a < b
    if t1 then L1 else L2
L1: x = a
    jump L3
L2: x = b
L3:
```

VSM

```
    push a
    push b
    lessthan
    push L2
    jumpif
L1: push a
    pop x
    push L3
    jump
L2: push b
    pop x
L3:
```

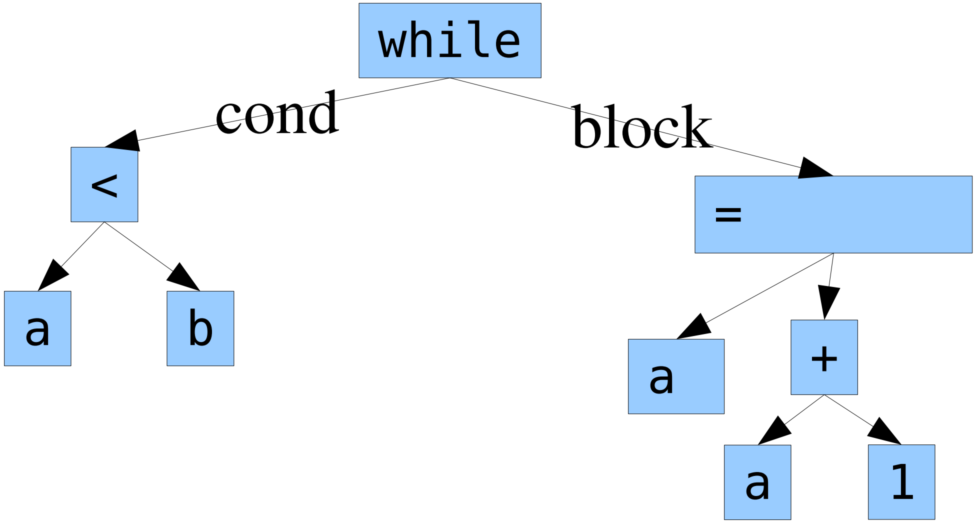
If-then-elsif-else statements

- Generate 3AI and VSM code for the following parse tree



Looping

- Looping can be done using conditional and unconditional jumps
- Exercise: Write the 3AI and VSM code for the following parse tree:



Switch Statements

- Switch statements, like those in C, C++, and Java
- For this, we introduce new 3-address instruction
  - 3AI: case A B : “if A is true then goto label b”
  - VSM: case (A and B are the top two stack items)
- This instruction is treated as a candidate for special treatment during the code generation phase

Function Calls

- In 3-address codes
  - Function arguments are passed using the param instruction
  - Functions are called using the call instruction
  - Return values are returned using the return instruction
- In a virtual stack machine
  - Function arguments are just pushed onto a stack
  - Functions are called using the call instruction
  - Return values are left on the stack
  - A function should leave only its parameters and return value on the stack when it returns

Function Calls Example

```
int ack(n,m) {
  int x;
  ...
  return x;
}

{
  ...
  r = ack(d, d+4)
  ...
}
```

```
ack:
   ...
   return x


...
  param d
  t1 = d + 4
  param t1
  t2 = call ack
  r = t2
```

```
ack:
   ...
   push x
   return


...
  push d
  push d
  push 4
  add
  call ack
  pop r
```

Where Do We Go From Here?

- After generating intermediate code there are a few options
  - We can optimize the intermediate code
  - We can generate machine code
- Challenges
  - To optimize intermediate representation code we need to reason about it
    - But this leads to undecidable problems
  - To generate code we need to manage storage
    - VSM hides this by giving us an infinite stack
    - 3AI hides this by giving us an infinite number of temporary variables