CS 335: Intermediate Representations

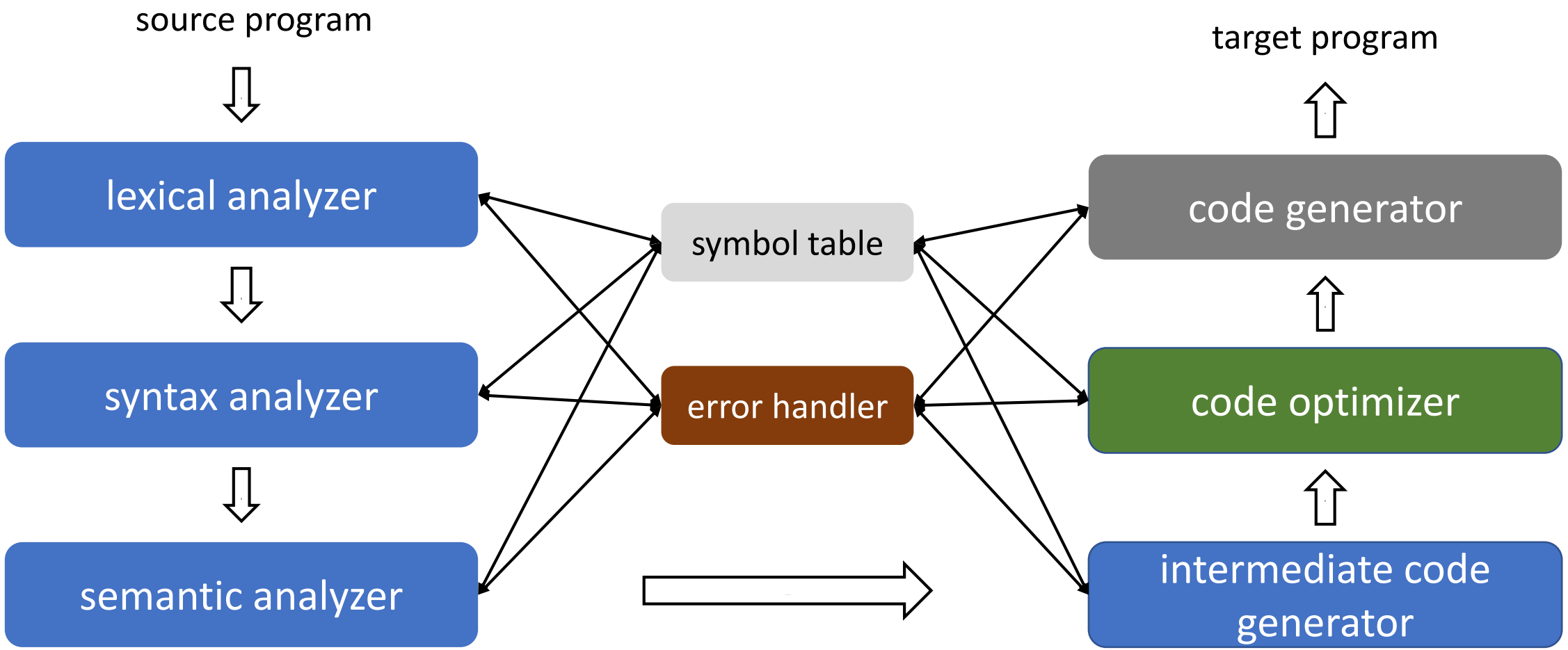
Swarnendu Biswas

Semester 2019-2020-II

CSE, IIT Kanpur

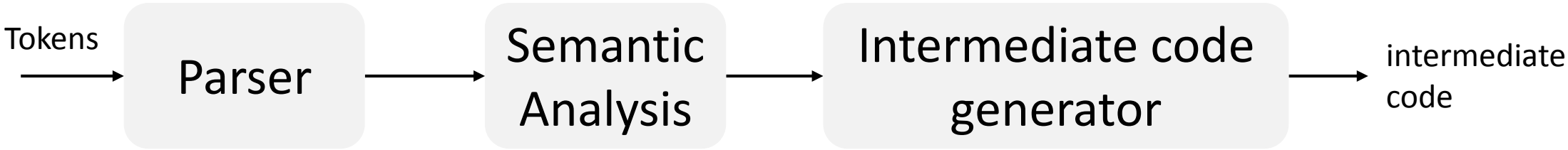
Content influenced by many excellent references, see References slide for acknowledgements.

An Overview of Compilation



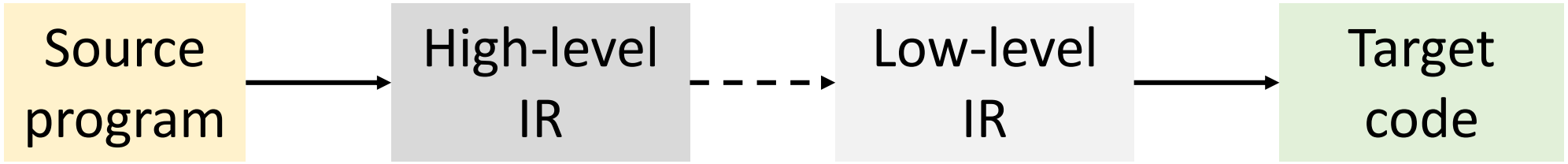
Intermediate Representation (IR)

- IR is a data structure used internally by a compiler or virtual machine (VM) while translating a source program
  - Front end analyses a source program and creates an IR
  - Back end analyses the IR and generates target code
- An well-designed IR helps ease compiler development
  - Plugin m front ends with n back ends to come up with $m \times n$ compilers



Intermediate Representation (IR)

- Compilers may create a number of IRs during the translation process



- High-level IRs are closer to the source (e.g., syntax trees)
  - Well-suited for tasks like static type checking
- Low-level IRs are closer to the target ISA
  - Well-suited for tasks like register allocation and instruction selection

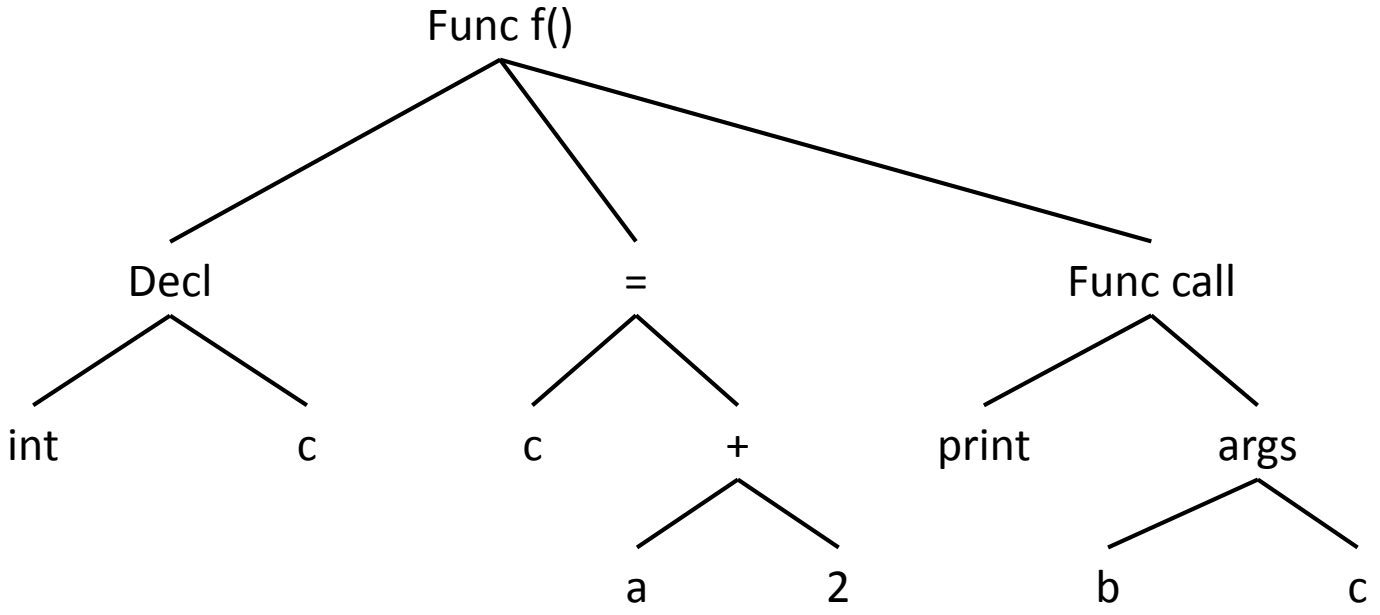
Types of IRs

- Abstraction-based classification
  - High-level IR – preserves many source structures
    - For example, loop structures and array bounds
  - Medium-level IR – often independent of the source language
    - Chosen to be suitable to represent language features and to generate code
  - Low-level IR – similar to the target ISA
- Structural classification
  - Graphical, or linear, or hybrid
  - Hybrid combines features of both graphical and linear IR

High-Level IR

- Maintains enough information to reconstruct source code (e.g., AST)
  - Structured control flow, variable names, method names
  - Allows high-level optimizations like inlining

```
int f(int a, int b) {
    int c;
    c = a + 2;
    print(b, c);
}
```



Medium-Level and Low-Level IR

- Medium-level IR (MIR)
  - Independent of source language
  - Amenable to code optimizations (for e.g., manipulating list of instructions)
  - Amenable for code generation for a variety of architectures
  - E.g., three-address code
- Low-level IR (LIR)
  - Close correspondence to the target ISA
    - Assembly code with extra pseudo-instructions plus infinite registers
  - Is often architecture-dependent
  - Allows low-level optimizations (e.g., instruction scheduling)

Points about IR Design

- IR needs to be amenable to analysis and modifications
- Issues to consider in IR design
  - Decide on the appropriate level of abstraction to cover many language features
  - Suitability for code optimization and code generation
  - Other factors like space overhead
- Difficult to come up with a general IR

Graphical IRs

Derivation of an input

CFG

$Start \rightarrow Expr$

$Expr \rightarrow Expr + Term$

$Expr \rightarrow Expr - Term$

$Expr \rightarrow Term$

$Term \rightarrow Term \times Factor$

$Term \rightarrow Term \div Factor$

$Term \rightarrow Factor$

$Factor \rightarrow (Expr)$

$Factor \rightarrow \mathbf{num} \mid \mathbf{name}$

$\boldsymbol{a \times 2 + a \times 2 \times b}$

$Start \rightarrow Expr \rightarrow Expr + Term$

$\rightarrow Term + Term$

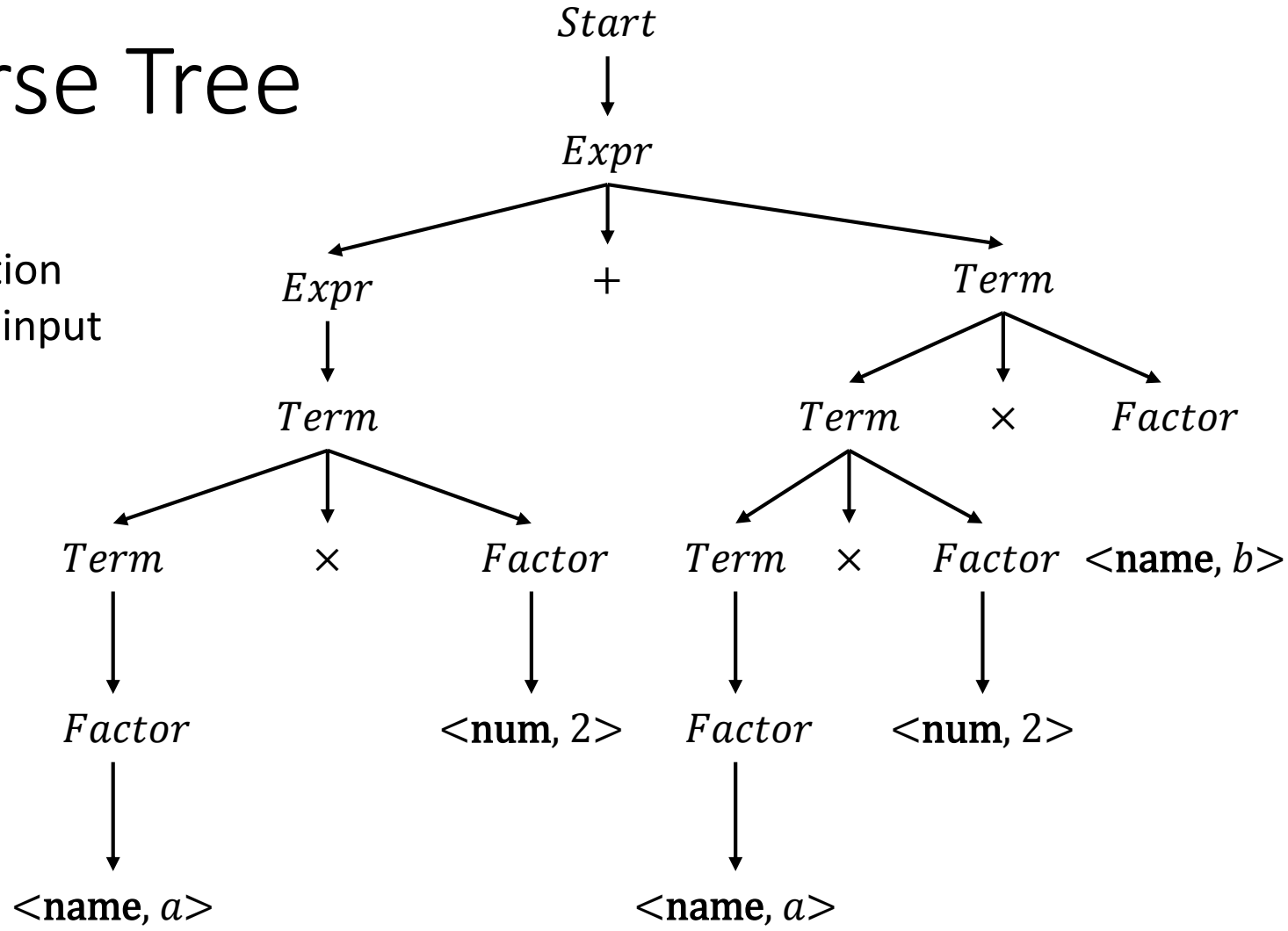
$\rightarrow Term \times Factor + Term$

…

Example of a Parse Tree

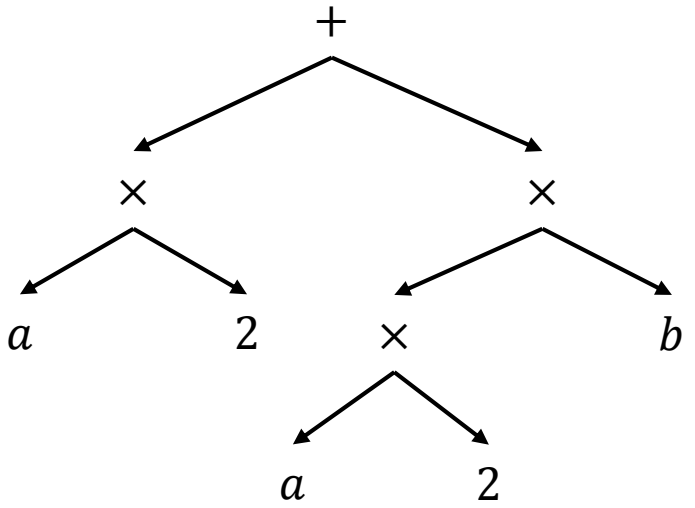
Parse tree is a graphical representation of a derivation corresponding to an input

Used in parsing and attribute grammar frameworks



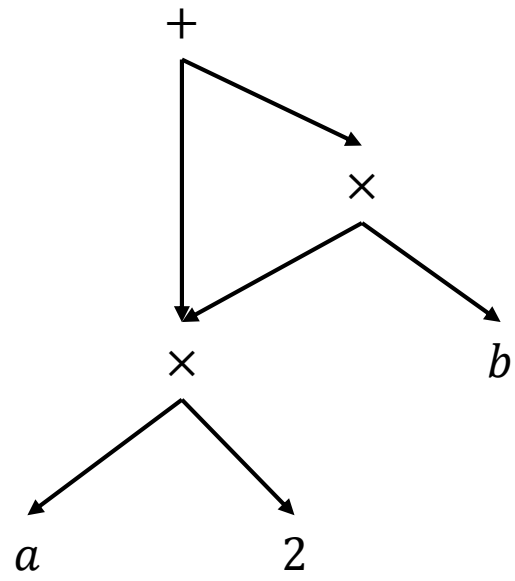
Abstract Syntax Tree (AST)

- An AST compresses the parse tree by omitting many internal nodes corresponding to nonterminal symbols
  - Leaf nodes represent operands
  - Retains precedence and meaning of the expression
  - AST is a near-source-level representation



Directed Acyclic Graph (DAG)

- DAGs compress ASTs and can avoid duplicate subtrees
  - Nodes in a DAG can have multiple parents
- DAGs encode hints for evaluating the expression
  - If a does not change between the two uses, then the compiler can generate code to evaluate $a \times 2$ only once
  - Reduces memory footprint



SDD to Construct Syntax Trees

Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node(" + ", E_1.node, T.node)$
$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node(" - ", E_1.node, T.node)$
$E \rightarrow T$	$E.node = T.node$
$T \rightarrow (E)$	$T.node = E.node$
$T \rightarrow$ **id**	$T.node = \textbf{new}\ Leaf(\textbf{id}, \textbf{id}.entry)$
$T \rightarrow$ **num**	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Constructing a DAG

$p_1 = Leaf(\text{id}, entry\text{–}a)$

$p_2 = Leaf(\text{id}, entry\text{–}a) = p_1$

$p_3 = Leaf(\text{id}, entry\text{–}b)$

$p_4 = Leaf(\text{id}, entry\text{–}c)$

$p_5 = Node(" - ", p_3, p_4)$

$p_6 = Node(" * ", p_1, p_5)$

$p_7 = Node(" + ", p_1, p_6)$

$p_8 = Leaf(\text{id}, entry\text{–}b) = p_3$

$p_9 = Leaf(\text{id}, entry\text{–}c) = p_4$

$p_{10} = Node(" - ", p_3, p_4) = p_5$

$p_{11} = Leaf(\text{id}, entry\text{–}d)$

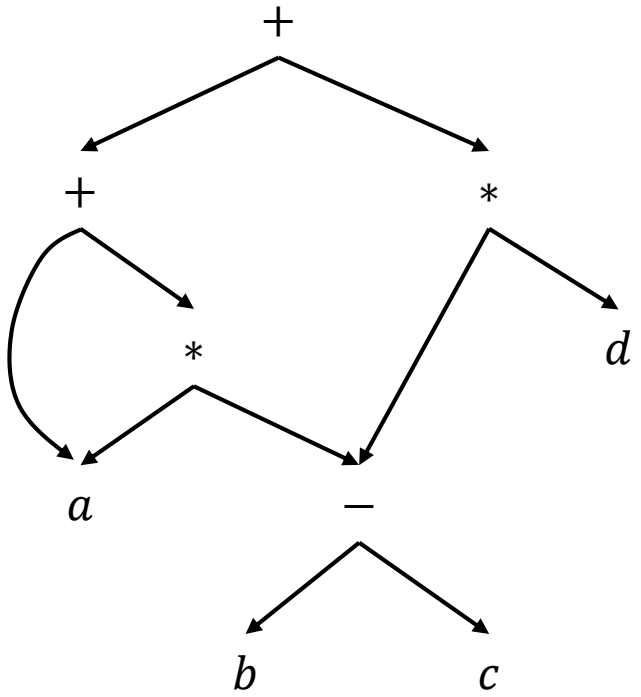
$p_{12} = Node(" * ", p_5, p_{11})$

$p_{13} = Node(" + ", p_7, p_{12})$

return existing node if it exits

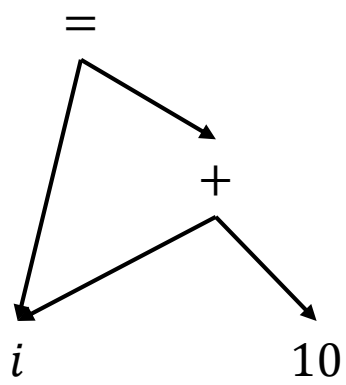
DAG for expression

$a + a * (b - c) + (b - c) * d$



Value-Number Method for DAGs

Often DAG nodes are stored in an array data structure



1	id	→ symbol table entry for i	
2	**num**	10	
3	+	1	2
4	=	1	3

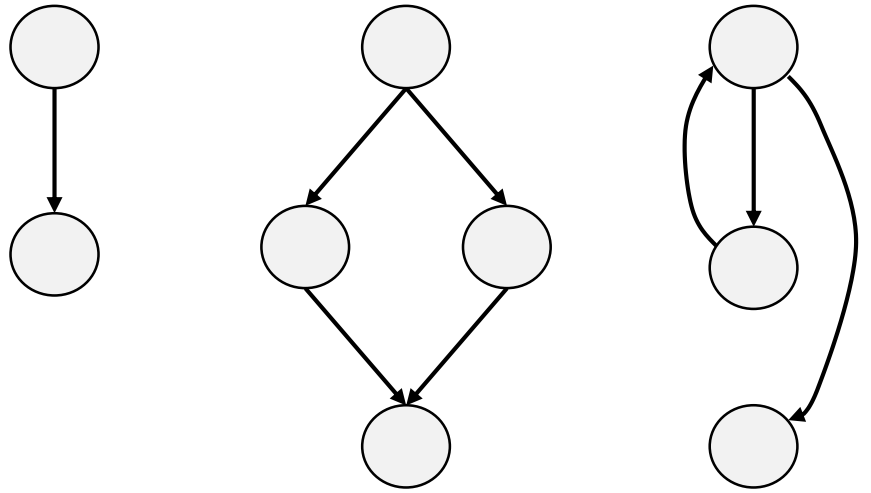
value number

signature

Sigature of an interior node is the triple $<op, l, r>$

Control Flow Graph (CFG)

- Graphical representation of control flow during execution
- Each node represents a statement or basic block (BB)
  - BB is a maximal sequence of instructions with only one entry and one exit point
  - An entry and an exit node are often added
  - An edge represents possible transfer of control between BBs
- Used for compiler optimizations and static analysis
  - Instruction scheduling, global register allocation



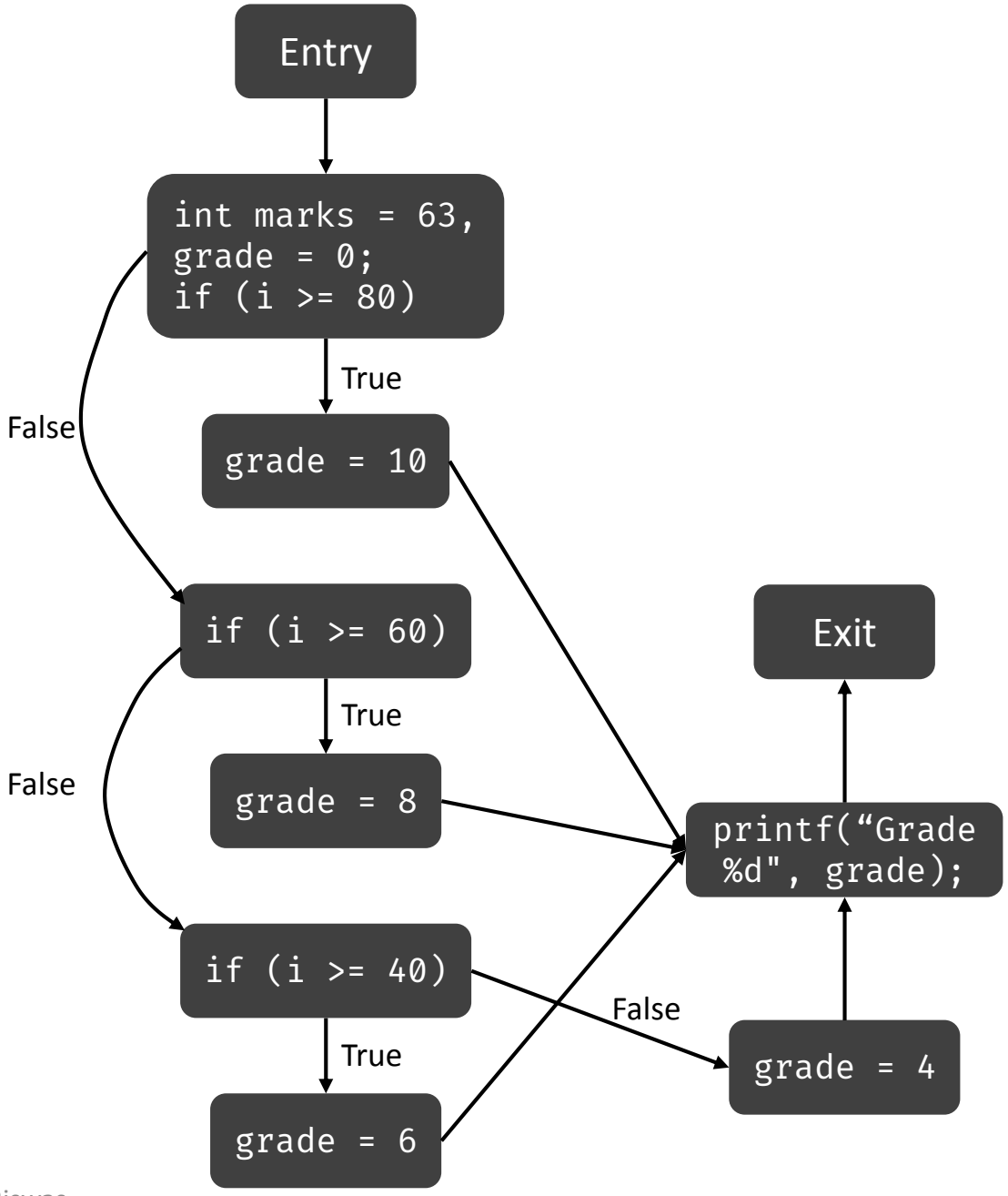
straight-line code

predicate

loop iteration

Example of a CFG

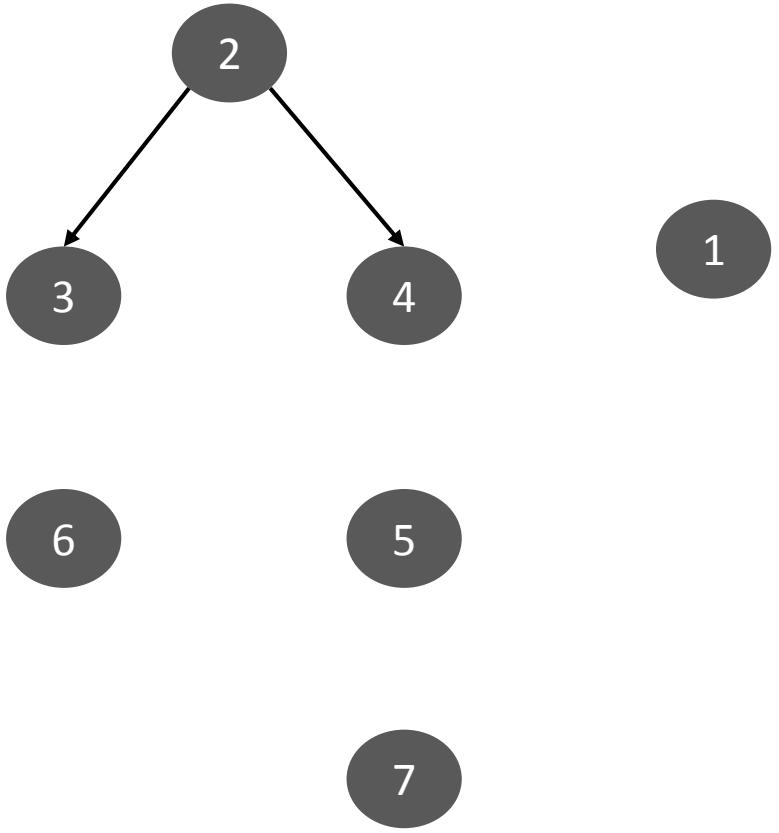
```
int main() {
    int marks = 63, grade = 0;
    if (i >= 80)
       grade = 10;
    else if (i >= 60)
       grade = 8;
    else if (i >= 40)
       grade = 6;
    else
       grade = 4;
    printf(“Grade %d", grade);
    return 0;
}
```



Data Dependence Graph

- A graph that models flow of values from definitions to uses in a code fragment
  - Nodes represent operations
  - Does not capture the control flow

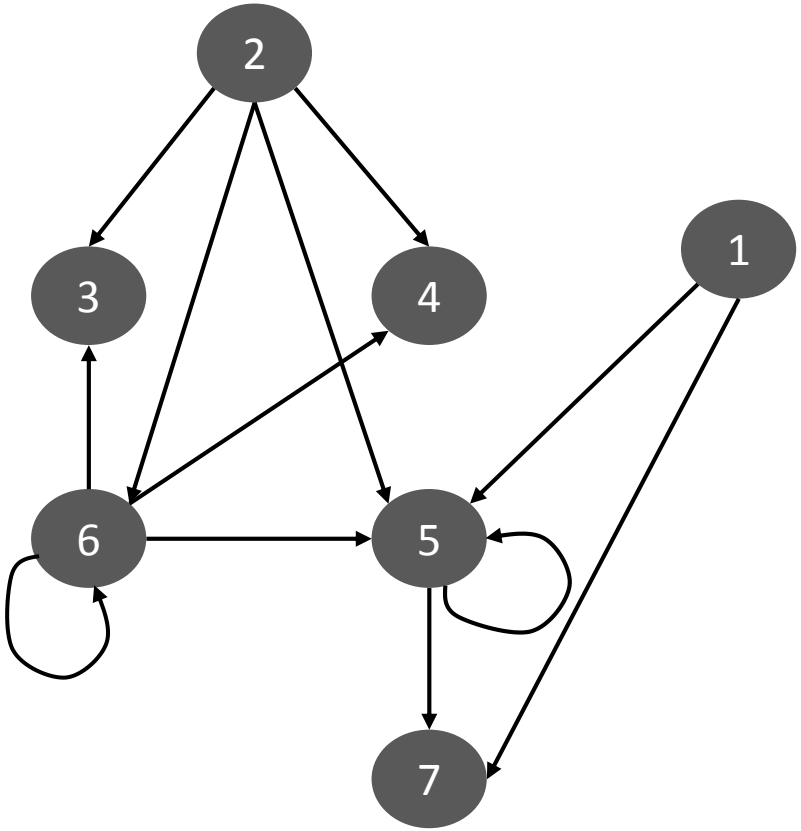
```
1.  x = 0
2.  i = 1
3.  while (i < 100)
4.     if (a[i] > 0)
5.         x = x + a[i]
6.     i = i + 1
7.  print(x)
```



Data Dependence Graph

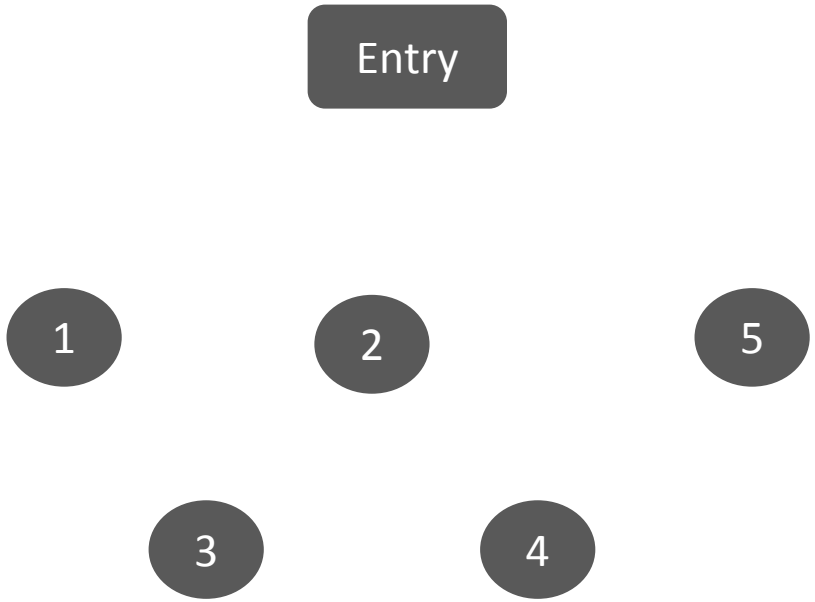
- A graph that models flow of values from definitions to uses in a code fragment
  - Nodes represent operations
  - Does not capture the control flow

```
1.  x = 0
2.  i = 1
3.  while (i < 100)
4.     if (a[i] > 0)
5.         x = x + a[i]
6.     i = i + 1
7.  print(x)
```



Control Dependence Graph

```
1.  read i
2.  if i == 1
3.     print “true”
     else
4.     print “false”
5.  print “done”
```

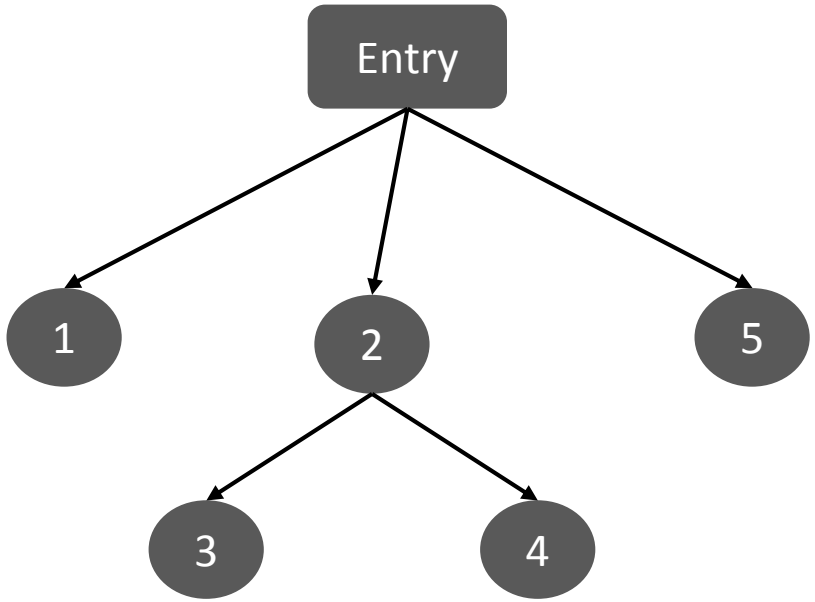


- Vertices represent executable statements
- A dummy entry node represents the start of execution

- If statement X determines whether statement Y is executed, then Y is control dependent on X
- Statements that are guaranteed to execute are control dependent on entry to the program

Control Dependence Graph

```
1.  read i
2.  if i == 1
3.     print “true”
     else
4.     print “false”
5.  print “done”
```



- Vertices represent executable statements
- A dummy entry node represents the start of execution

- If statement X determines whether statement Y is executed, then Y is control dependent on X
- Statements that are guaranteed to execute are control dependent on entry to the program

Call Graph

- A graph that represents the calling relationships among the procedures in a program
  - Represents runtime transfer of control among procedures
- The call graph has a node for each procedure and an edge for each call site
  - A function p calls another function q from three places
  - The call graph will have three (p, q) edges, one for each call site

Linear IRs

Types of Linear IRs

- One-address code
  - Models behavior of stack machines and accumulator machines
  - Makes use of implicit names
  - Useful where storage efficiency is important (for e.g., transmission over a network)
- Two-address code
  - The result of one address is often redefined with the result (called a destructive operation)
  - Not very popular currently
- Three-address code
  - Most operations take two operands and produce a result
  - Resembles a simple RISC machine

Stack Machine Code

- Assumes the presence of a stack with operands
- Operations take their operands from the stack and push the result onto the stack
  - Operands are addressed implicitly with the stack pointer

- JVM is a stack-based VM

$a - 2 \times b$

```
push 2
push b
multiply
push a
subtract
```

Three-Address Code (3AC)

- At most one operator in the RHS

$$x + y * z$$

$$t_1 = y * z$$
$$t_2 = x + t_1$$

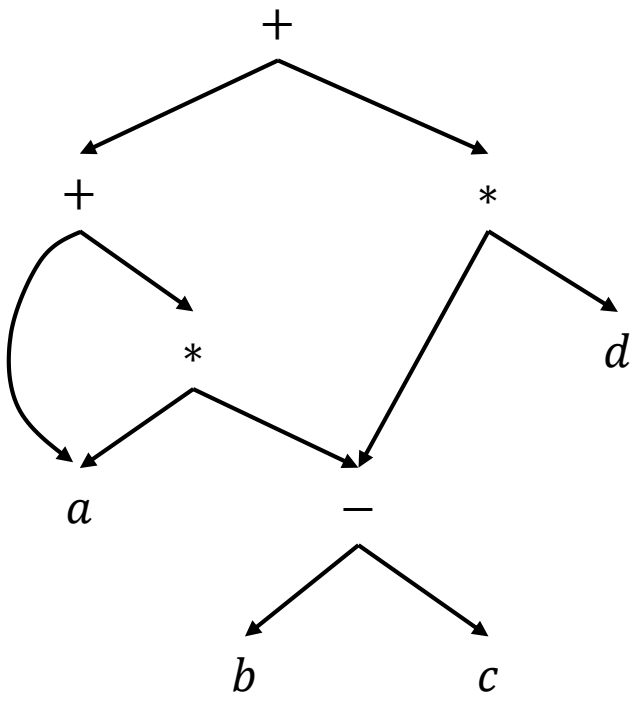
t_1 and t_2 are compiler-generated temporary variables

- 3AC is a linearized representation of a syntax tree where explicit names correspond to interior graph nodes
- 3AC is popularly used in code optimization and code generation
  - Use of names for intermediate values allows 3AC to be easily rearranged

DAG and Corresponding 3AC

DAG for expression

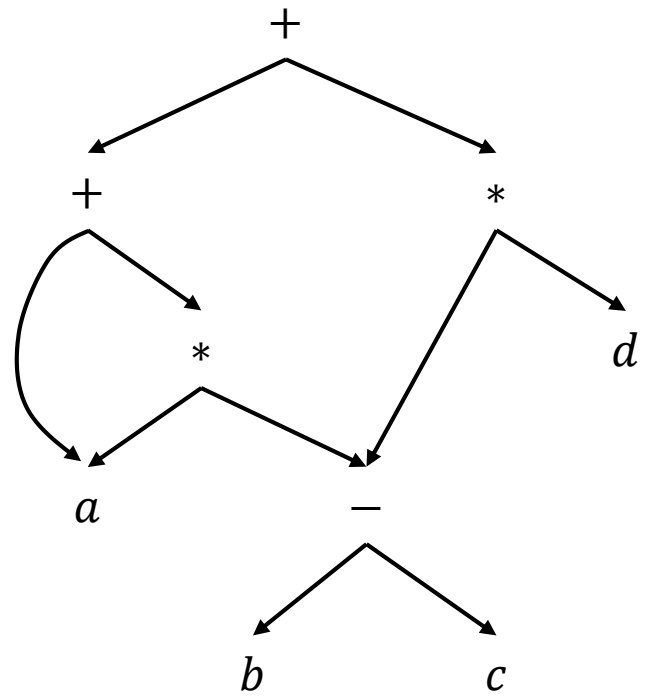
$a + a * (b - c) + (b - c) * d$



3AC?

DAG and Corresponding 3AC

DAG for expression
$a + a * (b - c) + (b - c) * d$



$$t_1 = b - c$$
$$t_2 = a * t_1$$
$$t_3 = a + t_2$$
$$t_4 = t_1 * d$$
$$t_5 = t_3 + t_4$$

Forms of 3AC

- Composed of two concepts: addresses and instructions
- Addresses can be program variables, constants, and temporaries
  - Variables can be pointers to the symbol table entries
  - Distinct names for each temporary helps in optimization analyses

Forms of 3AC

i. Assignments of the form $x = y$ **op** z, $x =$ **op** y, or $x = y$
ii. Unconditional jump goto L
iii. Conditional jumps of the form if x goto L, if x **relop** y goto L
iv. Procedure calls and returns of the form "param x", "call p, n", "$y =$ call p, n", and "return y"

param x_1	$p(x_1, x_2, \ldots, x_n)$
param x_2	
…	
param x_n	
call p, n	

Forms of 3AC

v. Indexed copy instructions of the form $x = y[i]$ and $x[i] = y$
vi. Address and pointer assignments of the form $x = \&y$, $x = *y$, and $*x = y$

Forms of 3AC

```
do
  i = i + 1;
while (a[i] < v);
```

Suppose each array element takes 8 units of space

Symbolic Labels	Position Numbers
L: $t_1 = i + 1$	100: $t_1 = i + 1$
$i = t_1$	101: $i = t_1$
$t_2 = i * 8$	102: $t_2 = i * 8$
$t_3 = a[t_2]$	103: $t_3 = a[t_2]$
if $t_3 < v$ goto L	104: if $t_3 < v$ goto 100

Implementing 3AC

- We can use data structures like quadruples, triples, and indirect triples
- Quadruple (or quad)
  - Has four fields op, arg_1, arg_2, and $result$
  - Instructions with unary operators will not use arg_2
  - Operators like $param$ do not use both arg_2 and $result$
  - Conditional and unconditional jumps put the target label in $result$

Implementing 3AC with Quadruples

Consider expression
$a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

	op	arg_1	arg_2	$result$
0	$-$	c		t_1
1	$*$	b	t_1	t_2
2	$-$	c		t_3
3	$*$	b	t_3	t_4
4	$+$	t_2	t_4	t_5
5	$=$	t_5		a

Implementing 3AC with Triples

Triples have three fields op, arg_1, and arg_2

Consider expression
$a = b * -c + b * -c$

$t_1 = -c$
$t_2 = b * t_1$
$t_3 = -c$
$t_4 = b * t_3$
$t_5 = t_2 + t_4$
$a = t_5$

	op	arg_1	arg_2
0	$-$	c	
1	$*$	b	(0)
2	$-$	c	
3	$*$	b	(2)
4	$+$	(1)	(3)
5	$=$	a	(4)

Representations in Triples

- How do you represent $x[i] = y$ and $x = y[i]$ with triples?

Representations in Triples

- How do you represent $x[i] = y$ and $x = y[i]$ with triples?

		op	arg_1	arg_2
$x[i] = y$	0	[ ]	x	i
	1	$=$	(0)	y

		op	arg_1	arg_2
$x = y[i]$	0	[ ]	y	i
	1	$=$	x	(0)

Quadruples vs Triples

Quadruples

- Requires many temporaries
- Easy to move around instructions

Triples

- Requires fewer temporaries, temporaries are implicit
- Implicit references need to be updated if instructions are moved around

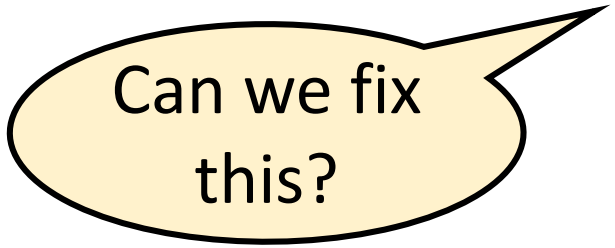
Quadruples vs Triples

Quadruples

- Requires many temporaries
- Easy to move around instructions

Triples

- Requires fewer temporaries, temporaries are implicit
- Implicit references need to be updated if instructions are moved around



Static Single Assignment (SSA) Form

- All assignments in SSA are to variables with different names
  - Every variable is defined before it is used
- SSA encodes information about both control and data flow

3AC	SSA
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q * d$	$p_2 = q_1 * d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

Static Single Assignment (SSA) Form

- How about variables defined in multiple control flow paths?

```
if (flag) {
   x = −1
} else {
   x = 1
}
y = x ∗ a
```

```
if (flag) {
   x_1 = −1
} else {
   x_2 = 1
}
y = ⋯ ∗ a
```

Static Single Assignment (SSA) Form

- How about variables defined in multiple control flow paths?

```
if (flag) {
   x = −1
} else {
   x = 1
}
y = x * a
```

```
if (flag) {
   x1 = −1
} else {
   x2 = 1
}
x3 = ϕ(x1, x2)
y = x3 * a
```

- A ϕ function takes several names and merges them defining a new name

Another Example in SSA Form

```
x = …
y = …
while (x < 100)
    x = x + 1
    y = y + x
```

```
        x0 = …
        y0 = …
        if (x0 >= 100) goto next
loop:   …
        …
        x2 = x1 + 1
        y2 = y1 + 1
        if (x2 < 100) goto loop
next:   x3 = …
        y3 = …
```

Another Example in SSA Form

```
x = …
y = …
while (x < 100)
    x = x + 1
    y = y + x
```

```
        x0 = …
        y0 = …
        if (x0 >= 100) goto next
loop:   x1 = φ(x0, x2)
        y1 = φ(y0, y2)
        x2 = x1 + 1
        y2 = y1 + x2
        if (x2 < 100) goto loop
next:   x3 = φ(x0, x2)
        y3 = φ(y0, y2)
```

Static Single Assignment (SSA) Form

- A program is in SSA form if
  i. Each definition has a new name
  ii. Each use refers to a single definition

- SSA helps code optimizations since no names are killed
  - Makes use-def chains explicit, otherwise Reaching definitions analysis would be required in absence of SSA

```
y = 1
y = 2
x = y
```

$y_1 = 1$
$y_2 = 2$
$x = y_2$

Static Single Assignment (SSA) Form

- SSA form has had a huge impact on compiler design
- Most modern production compilers use SSA form
  - For example, GCC, LLVM, Hotspot

Other Linear IRs

Java Bytecode

```
0:   iconst_2
1:   istore_1
2:   iload_1
3:   sipush  1000
6:   if_icmpge       44
9:   iconst_2
10:  istore_2
11:  iload_2
12:  iload_1
13:  if_icmpge       31
16:  iload_1
17:  iload_2
```

LLVM IR

```
define i32 @f(i32 %a, i32 %b) {
; <label>:0
    %1 = mul i32 2, %b
    %2 = add i32 %a, %1
    ret i32 %2
}
define i32 @main() {
; <label>:0
    %1 = call i32 @f(i32 10, i32 20)
    ret i32 %1
}
```

Symbol Table

Need for a Symbol Table

- Compilers generate meta-information during translation
  - For example, type of a variable, lexeme, line number for the declaration, and scope
- Information is saved in a data structure called symbol table
  - Alternate is to maintain meta-information in AST nodes and recompute the information when needed by AST traversal
    - Repeated AST traversals can be expensive
  - Saves all declarations, helps check if a variable is declared, helps with type checking and determining scope of a variable
- Symbol table needs to be updated whenever
  - A new name is discovered
  - New information about an existing name is discovered

Desired Properties of Symbol Table

- Symbol table is accessed across several compiler phases
- Interface
  - `lookup(name)` – Return the value stored against `name`
  - `insert(name, record)` – Add information about the variable `name`

- Efficiency is paramount
  - Unordered lists vs Ordered lists vs Hash tables
  - Should be efficient to grow the symbol table
    - Number of variables may vary across programs

Symbol Table Entries

- Each entry corresponds to a declaration of a name
- Entry format need not be uniform because information depends upon the type of the name
- Symbol table information is filled in at various times
  - Keywords can be entered initially
  - Identifier lexemes are added by the scanner
    - Attributes are filled in as information becomes available

Nested Scopes

```
static int w = 1;                       /* level 0 */
int x = 0;
void example(int a, int b) {
  int c = 1;                            /* level 1 */
  {
    int b = 2, z = 3;                   /* level 2a */
    …
  }
  {
    int a = 4, x = 5;                   /* level 2b */
    …
    {
      int c = 6, x = 7;                 /* level 3 */
      b = a + b + c + w;
    } } }

int main() { example(10, 20); }
```

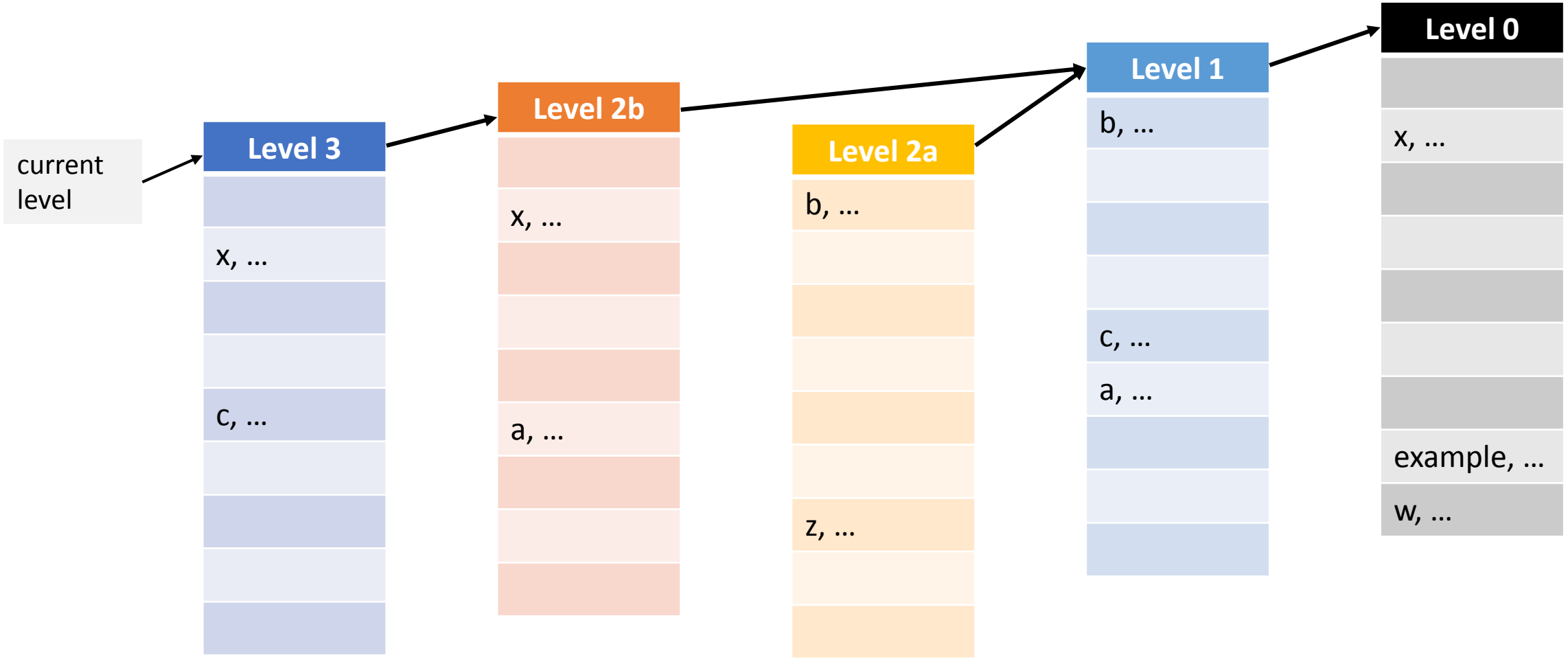
Level	Names
0	w, x, example
1	a, b, c
2a	b, z
2b	a, x
3	c, x

Dealing with Nested Scopes

- Name resolution – resolve a name reference to its specific declaration

- Possible idea
  - Create a new symbol table with each new lexical scope
  - `insert()` operates on the current symbol table
  - `lookup()` checks symbol tables in order
    - Start with current scope and go up the hierarchy
    - Report an error only if `lookup()` fails across all levels
  - Also called a “sheaf of tables”

Symbol Table Structure for Nested Scope



Maintaining Namespace of Structure Fields

- Separate tables
  - Maintain a separate table for each structure
- Selector table
  - Maintain a separate table for structure fields
  - Need to maintain fully-qualified field names
- Unified table
  - Club all information in a single global symbol table
  - Maintain fully-qualified field names

Name Resolution for Object-Oriented Languages

- Resolution rules are slightly more involved
  - Scoped symbol tables for a method, a class, and other classes in the package and package-level variables

- Consider resolving a name `foo` in a method `m` in class `Klass`
  - First check the lexically scoped symbol table corresponding to `m`
  - If not found, then search the symbol table according to the inheritance hierarchy, starting from `Klass`
  - If not found, then search the global symbol table for that name

Generating IR

Translate expressions, array references, declarations, Boolean expressions, and control flow statements

Intermediate Code Generation

- Code generation needs to map source language abstractions to target machine abstractions

- Example of language-level abstractions
  - Identifiers, operators, expressions, statements, conditionals, iterations, functions (user-defined or libraries)
- Example of target-level abstractions
  - Memory locations, registers, stack, opcodes, addressing modes, system libraries, interface with the operating systems

Translation to 3AC

$$a = b * -c + b * -c$$

$$t_1 = -c$$
$$t_2 = b * t_1$$
$$t_3 = -c$$
$$t_4 = b * t_3$$
$$t_5 = t_2 + t_4$$
$$a = t_5$$

Production
$S \rightarrow \textbf{id} = E$
$E \rightarrow E_1 + E_2$
$E \rightarrow E_1 * E_2$
$E \rightarrow -E_1$
$E \rightarrow (E_1)$
$E \rightarrow \textbf{id}$

- $E.addr$ – Holds the value of expression E
  - Can be a name, a constant, or a temporary
- $E.code$ – Sequence of 3AC that evaluates E
- $S.code$ – Stores the 3AC for statement S
- gen – Helper function to create a 3AC instruction

SDD for Translating Expressions to 3AC

Production	Semantic Rules
$S \rightarrow \textbf{id} = E$	
$E \rightarrow E_1 + E_2$	
$E \rightarrow E_1 * E_2$	
$E \rightarrow -E_1$	
$E \rightarrow (E_1)$	
$E \rightarrow \textbf{id}$	$E.addr = symtop.get(\textbf{id}.lexeme)$ $E.code = ""$

$symtop$ points to the current symbol table

SDD for Translating Expressions to 3AC

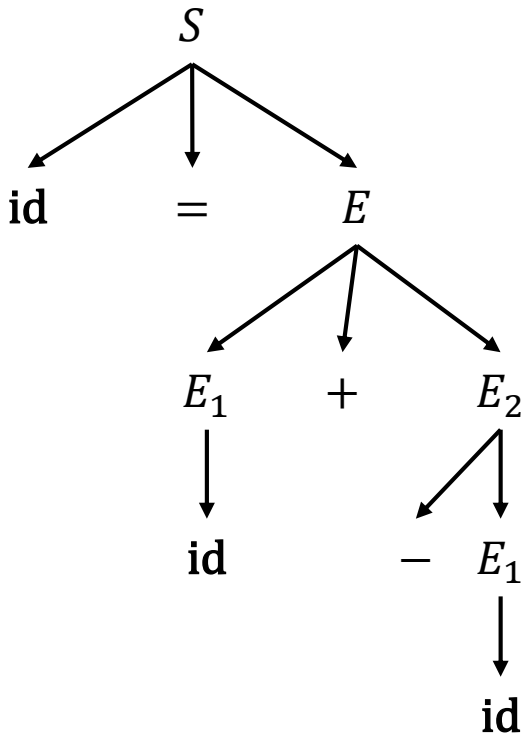
Production	Semantic Rules
$S \rightarrow \textbf{id} = E$	$S.code = E.code \,\|\|\, gen(symtop.get(\textbf{id}.lexeme)\ "="\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \,\|\|\, E_2.code \,\|\|\, gen(E.addr\ "="\ E_1.addr\ "+"\ E_2.addr)$
$E \rightarrow E_1 * E_2$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \,\|\|\, E_2.code \,\|\|\, gen(E.addr\ "="\ E_1.addr\ "*"\ E_2.addr)$
$E \rightarrow -E_1$	$E.addr = \textbf{new}\ Temp()$ $E.code = E_1.code \,\|\|\, gen(E.addr\ "="\ "-"E_1.addr)$
$E \rightarrow (\ E_1\ )$	$E.addr = E_1.addr$ $E.code = E_1.code$
$E \rightarrow \textbf{id}$	$E.addr = symtop.get(\textbf{id}.lexeme)$ $E.code = ""$

symtop points to the current symbol table

Example of 3AC Generation

$a = b + -c$

$t_1 = -c$
$t_2 = b + t_1$
$a = t_2$



Incremental Translation

Production	Semantic Rules
$S \rightarrow \textbf{id} = E$	$gen(symtop.get(\textbf{id}.lexeme)\ "="\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $gen(E.addr\ "="\ E_1.addr\ "+"\ E_2.addr)$
$E \rightarrow E_1 * E_2$	$E.addr = \textbf{new}\ Temp()$ $gen(E.addr\ "="\ E_1.addr\ "*"\ E_2.addr)$
$E \rightarrow -E_1$	$E.addr = \textbf{new}\ Temp()$ $gen(E.addr\ "="\ "-"E_1.addr)$
$E \rightarrow (\ E_1\ )$	$E.addr = E_1.addr$
$E \rightarrow \textbf{id}$	$E.addr = symtop.get(\textbf{id}.lexeme)$

gen creates an 3AC instruction and appends it to an instruction stream

Translating Array References

- Challenge is in computing addresses of array references like $A[i][j]$
- Suppose w_r and w_e are the widths of a row and an element of an array respectively
- Address of array reference $A[i][j]$ is $base + i \times w_r + j \times w_e$
- Grammar can generate expressions like $c + a[i][j]$

Production
$S \rightarrow \textbf{id} = E$
$S \rightarrow L = E$
$E \rightarrow E_1 + E_2$
$E \rightarrow \textbf{id}$
$E \rightarrow L$
$L \rightarrow \textbf{id} [E]$
$L \rightarrow L_1 [E]$

Translating Array References

- L has three synthesized attributes
  - $addr$ is used for computing the offset for array reference
  - $array$ points to the symbol table entry for the array name
    - $L.array.base$ gives the base address of the array
  - $type$ is the type of the array generated by L
    - For array of type t, $t.width$ is the width of type t and $t.elem$ gives the element type

Production
$S \rightarrow \textbf{id} = E$
$S \rightarrow L = E$
$E \rightarrow E_1 + E_2$
$E \rightarrow \textbf{id}$
$E \rightarrow L$
$L \rightarrow \textbf{id} [E]$
$L \rightarrow L_1 [E]$

Translating Array References

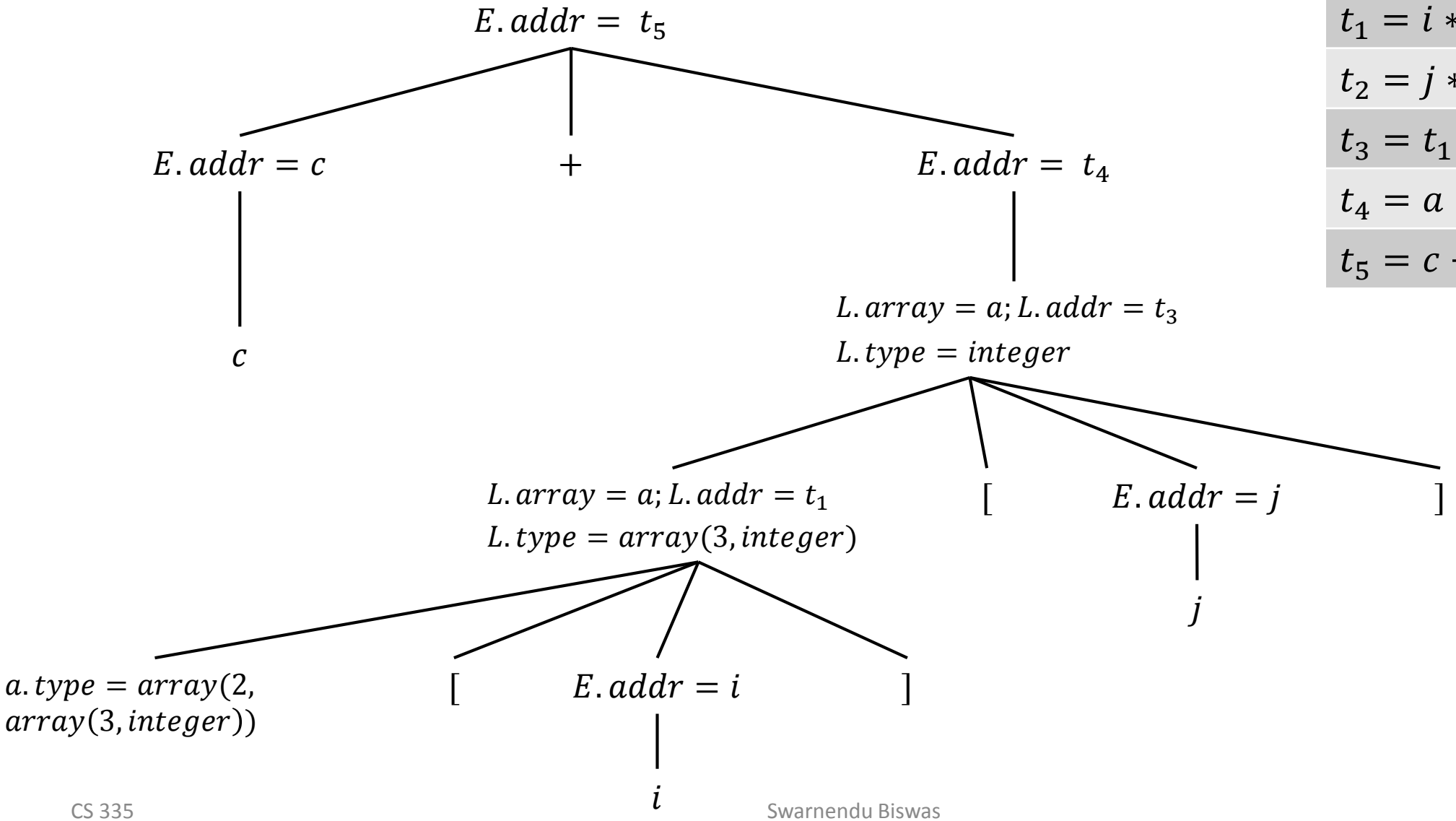
Production	Semantic Rules
$S \rightarrow \textbf{id} = E$	$gen(symtop.get(\textbf{id}.lexeme)\ "="\ E.addr)$
$S \rightarrow L = E$	$gen(L.array.base\ "["\ L.addr\ "]"\ "="\ E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \textbf{new}\ Temp()$ $gen(E.addr\ "="\ E_1.addr\ "+"\ E_2.addr)$
$E \rightarrow \textbf{id}$	$E.addr = symtop.get(\textbf{id}.lexeme)$
$E \rightarrow L$	$E.addr = \textbf{new}\ Temp()$ $gen(E.addr\ "="\ L.array.base\ "["\ L.addr\ "]")$
$L \rightarrow \textbf{id}\ [\ E\ ]$	$L.array = symtop.get(\textbf{id}.lexeme);\ L.type = L.array.type.elem$ $L.addr = \textbf{new}\ Temp();\ gen(L.addr\ "="\ E.addr\ "*"\ L.type.width)$
$L \rightarrow L_1\ [\ E\ ]$	$L.array = L_1.array;\ L.type = L_1.type.elem;\ t = \textbf{new}\ Temp()$ $gen(t\ "="\ E.addr\ "*"\ L.type.width)$ $gen(L.addr\ "="\ L_1.addr\ "+"\ t)$

Translating Expression $c + a[i][j]$

- Let a denote a 2×3 array of integers
  - Type of a is integer
  - Type of $a[i]$ is $array(3, integer)$, and $w_r = 12$ B
  - Type of $a[i][j]$ is $array(2, array(3, integer))$

3AC for $c + a[i][j]$
$t_1 = i * 12$
$t_2 = j * 4$
$t_3 = t_1 + t_2$
$t_4 = a\ [\ t_3\ ]$
$t_5 = c + t_4$

Annotated Parse Tree for $c + a[i][j]$



3AC for $c + a[i][j]$
$t_1 = i * 12$
$t_2 = j * 4$
$t_3 = t_1 + t_2$
$t_4 = a\ [\ t_3\ ]$
$t_5 = c + t_4$

Declarations

Goal: Layout storage for local variables as declarations are processed

$P \rightarrow D$
$D \rightarrow D; D$
$D \rightarrow$ **id**: T
$T \rightarrow$ **int**
$T \rightarrow$ **real**
$T \rightarrow$ **array** [**num**] **of** T_1
$T \rightarrow \uparrow T_1$

For each name create symbol table entry with information like type and relative address

The relative address is an offset from the base of the static data area or the field for local data in an activation record

Declarations

$P \rightarrow \{\ offset = 0; \}\ D$	
$D \rightarrow D; D$	
$D \rightarrow$ **id**: T	$\{\ enter(\textbf{id}.name, T.type, offset);\ \ offset = offset + T.width;\ \}$
$T \rightarrow$ **int**	$\{\ T.type\ =$ integer; $T.width = 4; \}$
$T \rightarrow$ **real**	$\{\ T.type\ =$ real; $T.width = 8;\ \}$
$T \rightarrow$ **array** [**num**] **of** T_1	$\{\ T.type = array(num.val, T_1.type);\ T.width = num.val \times T_1.width;\ \}$
$T \rightarrow \uparrow T_1$	$\{\ T.type = pointer(T_1.type);\ T.width = 4;\ \}$

- Global variable $offset$ keeps track of the next available relative address
- Function $enter$ creates a symbol table entry for $name$

Nested Procedures

- Invisible outside of its immediately enclosing procedure
- Can access local data of its enclosing procedures
- Used in languages like Pascal and functional languages like Haskell

```
function E(x: real): real;
    function F(y: real): real;
    begin
        F := x + y
    end;
begin
    E := F(3) + F(4)
end;
```

Nested Functions in GNU C

```
double foo(double a, double b) {
  double square(double z) { return z*z; }
  return square(a) + square(b);
}
```

```
void bar(int *array, int offset, int size) {
  int access(int *array, int index) { return array[index+offset]; }
  /* … */
  for (int i=0; i < size; i++)
    access(array, i);
  /* … */
}
```

https://gcc.gnu.org/onlinedocs/gcc/Nested-Functions.html

Keeping Track of Scope Information

$$P \rightarrow D$$
$$D \rightarrow D ; D \mid \mathbf{id} : T \mid \mathbf{proc\ id} ; D ; S$$

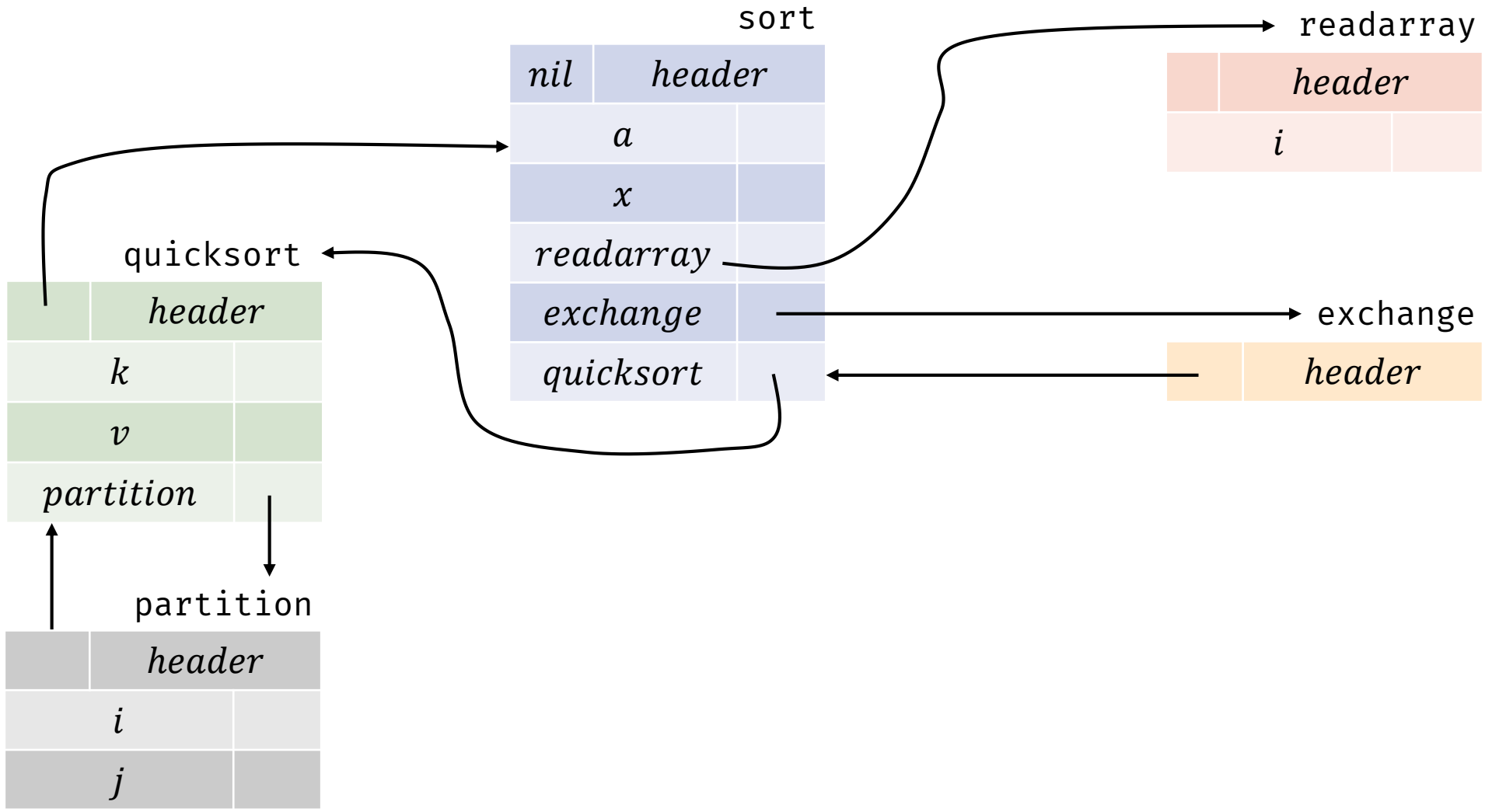
- A simple idea
  - When a nested procedure is seen, processing of declarations in enclosing procedure is temporarily suspended
  - A new symbol table is created for every procedure declaration $D \rightarrow \mathbf{proc\ id;}\ D_1 S$
    - Entries for D_1 are made in the new symbol table
  - Name represented by $\mathbf{id}$ is local to the enclosing procedure

Example Program with Nested Procedures

```
program sort;
  var a : array[1..n] of integer;
      x : integer;

  procedure readarray;
    var i : integer;
    ……

  procedure exchange(i,j:integers);
  ……
```

```
  procedure quicksort(m,n : integer);
    var k,v : integer;
    function partition(x,y:integer):
integer;
      var i,j: integer;
      ……
    ……

begin{main}
  ……
end
```

Symbol Tables for Nested Procedures



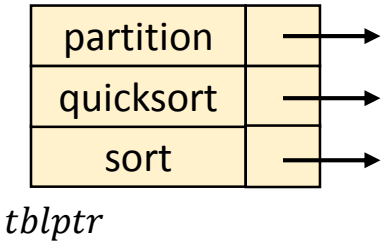
Helper Functions to Manipulate Symbol Table

- $mktable(previous)$
  - Create a new symbol table and return a pointer to the new table
- $enter(table, name, type, offset)$
  - Creates a new entry for name $name$ in the symbol table pointed by $table$
- $addwidth(table, width)$
  - Records the cumulative width of all the entries in $table$ in the header
- $enterproc(table, name, newtable)$
  - Creates a new entry for procedure $name$ in $table$
  - Argument $newtable$ points to the symbol table for procedure $name$

Constructing Nested Symbol Tables

$P \rightarrow \{\ t = mktable(nil);\ push(t, tblptr);\ push(0, offset);\ \}$ $D\ \{\ addwidth\big(top(tblptr), top(offset)\big);\ pop(tblptr);\ pop(offset);\ \}$
$D \rightarrow D_1 D_2$
$D \rightarrow \mathbf{proc\ id};\{\ t = mktable\big(top(tblptr)\big);\ push(t, tblptr);\ push(0, offset);\ \}$ $D_1; S \quad \{\ t = top(tblptr);\ addwidth\big(t, top(offset)\big);\ pop(tblptr);\ pop(offset);$ $enterproc(top(tblptr), \mathbf{id}.name, t);\ \}$
$D \rightarrow \mathbf{id} : T \quad \{\ enter\big(top(tblptr), \mathbf{id}.name, T.type, top(offset)\big);$ $top(offset) = top(offset) + T.width\ \}$

- $tblptr$ is a stack to hold pointers to symbol tables of enclosing procedures
- $offset$ is a stack to maintain relative offsets



Boolean Expressions

- Used to compute logical values (e.g., `x=true`) and influence flow of control (e.g., `if E then S`)

$B \rightarrow B \mid\mid B$
$\quad \rightarrow B \;\&\&\; B$
$\quad \rightarrow \; !\, B$
$\quad \rightarrow (\; B \;)$
$\quad \rightarrow E \; relop \; E$
$\quad \rightarrow$ **true**
$\quad \rightarrow$ **false**
$relop \rightarrow < \mid \leq \mid = \mid \neq \mid > \mid \geq$

Methods to represent value of Boolean expressions

- Evaluate similar to arithmetic expressions
  - True can be 1 (or any nonzero) and False can be zero
- Implement using flow of control, value is given by the position reached
  - Given expression $E_1 \; or \; E_2$, if E_1 is true, then the entire expression evaluates to true without evaluating E_2

Translating Boolean Expressions Using Numerical Representation

a **or** b **and not** c ➡

$t_1 = \textbf{not}\ c$
$t_2 = b\ \textbf{and}\ t_1$
$t_3 = a\ \textbf{or}\ t_2$

$a < b$ ➡ **if** $a < b$ **then** 1 **else** 0 ➡

100: **if** $a < b$ **then** goto 103
101: $t = 0$
102: goto 104
103: $t = 1$

Short Circuit Code

- Short circuit code translates to conditional and unconditional jumps
  - $B.true$ if B is true and $B.false$ if B is false

ifFalse x goto L	if x is false, execute the instruction labeled L next
ifTrue x goto L	if x is true, execute the instruction labeled L next

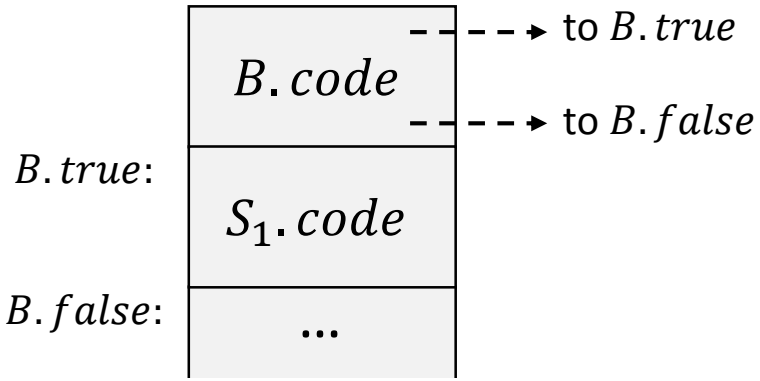
if ($x < 100$ || $x > 200$ && $x \neq y$)
  $x = 0$;

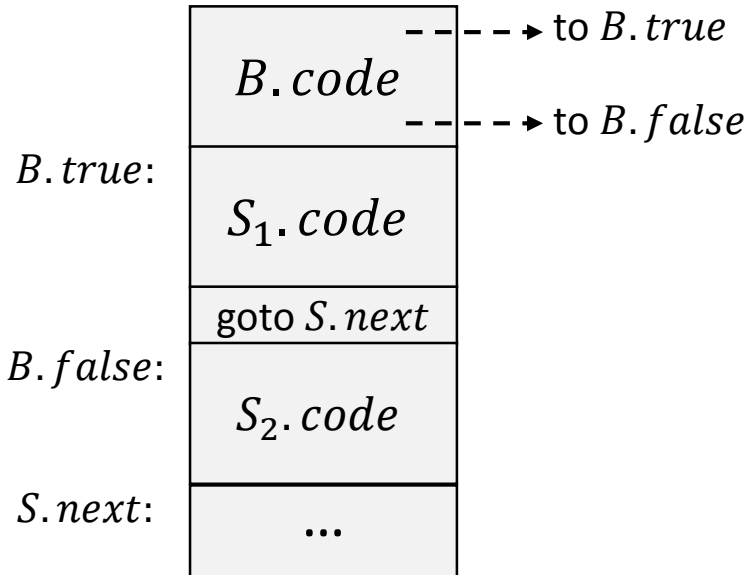
if $x < 100$ goto L_2
ifFalse $x > 200$ goto L_1
ifFalse x $\neq y$ goto L_1
L_2: $x = 0$
L_1:

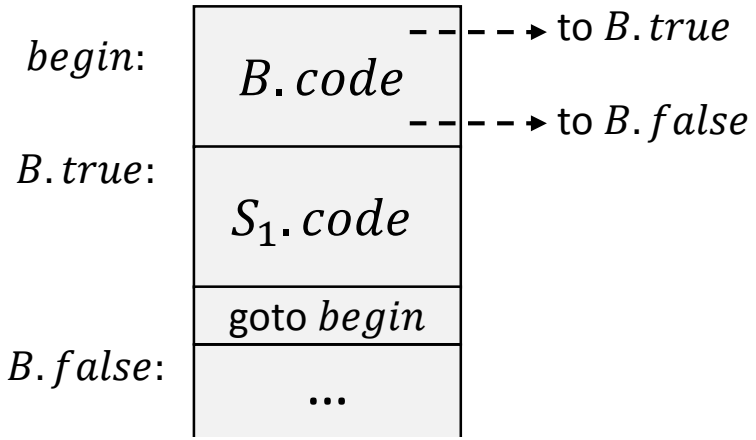
Control Flow

$S \rightarrow \textbf{if } (B)\ S_1 \mid \textbf{if } (B)\ S_1 \textbf{ else } S_2 \mid \textbf{while } (B)\ S_1$

Synthesized attributes $S.code$ and $B.code$ store 3AC
Inherited attributes $S.next$, $B.true$, and $B.false$ are for jumps







Generating 3AC for Boolean Expressions

Production	Semantic Rules
$B \rightarrow B_1 \mid\mid B_2$	$B_1.true = B.true;\ B_1.false = newlabel();$ $B_2.true = B.true;\ B_2.false = B.false;$ $B.code = B_1.code \mid\mid label(B_1.false) \mid\mid B_2.code$
$B \rightarrow B_1 \,\&\&\, B_2$	$B_1.true = newlabel();\ B_1.false = B.false;$ $B_2.true = B.true;\ B_2.false = B.false;$ $B.code = B_1.code \mid\mid label(B_1.true) \mid\mid B_2.code$
$B \rightarrow !\, B_1$	$B_1.true = B.false;\ B_1.false = B.true; B.code = B_1.code$
$B \rightarrow E_1\ relop\ E_2$	$B.code = E_1.code \mid\mid E_2.code \mid\mid$ $gen(\textbf{if}\ E_1.addr\ relop.op\ E_2.addr\ \text{goto}\ B.true)$ $gen(\text{"goto"}\ B.false)$
$B \rightarrow \textbf{true}$	$B.code = gen(\text{"goto"}\ B.true\ )$
$B \rightarrow \textbf{false}$	$B.code = gen(\text{"goto"}\ B.false)$

SDD for Control Flow Statements

Production	Semantic Rules
$P \rightarrow S$	$S.next = newlabel(); P.code = S.code \,\|\|\, label(S.next)$
$S \rightarrow$ **assign**	$S.code =$ **assign**$.code$
$S \rightarrow$ **if** $(B)\ S_1$	$B.true = newlabel(); B.false = S_1.next = S.next;$ $S.code = B.code \,\|\|\, label(B.true) \,\|\|\, S_1.code;$
$S \rightarrow$ **if** $(B)\ S_1$ **else** S_2	$B.true = newlabel(); B.false = newlabel();$ $S_1.next = S_2.next = S.next;$ $S.code = B.code \,\|\|\, label(B.true) \,\|\|\, S_1.code \,\|\|\, gen(\text{"goto"}\ S.next) \,\|\|$ $label(B.false) \,\|\|\, S_2.code$
$S \rightarrow$ **while** $(B)\ S_1$	$begin = newlabel(); B.true = newlabel(); B.false = S.next;$ $S_1.next = begin;$ $S.code = label(begin) \,\|\|\, B.code \,\|\|\, label(B.true) \,\|\|\, S_1.code \,\|\|$ $gen(\text{"goto"}\ begin)$
$S \rightarrow S_1\ S_2$	$S_1.next = newlabel();\ S_2.next = S.next;$ $S.code = S_1.code \,\|\|\, label(S_1.next) \,\|\|\, S_2.code$

Example of Control Flow Translation

```
if ( x < 100 || x > 200  && x ≠ y)
   x = 0;
```

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x ≠ y goto L2
        goto L1
L2:     x = 0
L1:
```

Example of Control Flow Translation

```
if ( x < 100 || x > 200  && x ≠ y)
   x = 0;
```

```
        if x < 100 goto L2
        goto L3
L3:     if x > 200 goto L4
        goto L1
L4:     if x ≠ y goto L2
        goto L1
L2:     x = 0
L1:
```

```
if ( x < 100 || x > 200  && x ≠ y)
   x = 0;
```

```
        if x < 100 goto L2
        ifFalse x  >  200 goto L1
        ifFalse x ≠ y goto L1
L2:     x = 0
L1:
```

Avoiding Redundant Gotos

- 3AC generation strategy can lead to redundant `gotos`

ifFalse x goto L	if x is false, execute the instruction labeled L next
ifTrue x goto L	if x is true, execute the instruction labeled L next

L_3: **if** $x > 200$ goto L_4
goto L_1
L_4: **…**

L_3: **ifFalse** $x > 200$ goto L_1
L_4: **…**

Avoiding Redundant Gotos

- Natural way is to let the control fall through, avoiding a jump
  - $fall$ is a special label indicating no jump

$S \rightarrow \text{if } (B)\ S_1$	$B.true = fall;$ $B.false = S_1.next = S.next;$ $S.code = B.code \ \|\| \ S_1.code;$
$B \rightarrow E_1\ relop\ E_2$	$test = E_1.addr\ relop.op\ E_2.addr$ $s =$ **if** $B.true \neq fall\ and\ B.false \neq fall\ then$ $gen(\text{"if"}\ test\ \text{"goto"}\ B.true) \ \|\| \ gen(\text{"goto"}\ B.false)$ **else if** $B.true \neq fall\ then\ gen(\text{"if"}\ test\ \text{"goto"}\ B.true)$ **else if** $B.false \neq fall\ then\ gen(\text{"ifFalse"}\ test\ \text{"goto"}\ B.false)$ **else** "" $B.code = E_1.code \ \|\| \ E_2.code \ \|\| \ s$

Challenge in Generating Code

- How do you associate labels to instruction addresses?
  - Consider the statement **if** (B) S
  - B is translated before S, so then how do we set the target of $B.false$?
  - A separate pass is needed to bind labels to addresses of instructions

$S \rightarrow$ **if** $(B)\ S_1$	$B.true = fall;$ $B.false = S_1.next = S.next;$ $S.code = B.code \ \|\| \ S_1.code;$

Backpatching

- Backpatching is an approach to generate code in one pass
  - Jump labels are synthesized attributes
  - Target of a jump is temporarily left unspecified when a jump is generated
  - Labels are filled when the proper target address can be determined

Backpatching

- Nonterminal B has two synthesized attributes
  - $truelist$ and $falselist$ – List of jump instructions to which control goes if B is true or false
  - Instructions will require a label
- Assume instructions are stored in an array
- Labels represent array indices

$$
\begin{aligned}
B &\rightarrow B_1 \,||\, MB_2 \\
&\rightarrow B_1 \,\&\&\, MB_2 \\
&\rightarrow !\, B_1 \\
&\rightarrow (B_1) \\
&\rightarrow E_1\ rel\ E_2 \\
&\rightarrow \mathbf{true} \\
&\rightarrow \mathbf{false} \\
M &\rightarrow \epsilon
\end{aligned}
$$

Backpatching

- We insert a marker nonterminal M in the grammar to pick up index of next quadruple

$M \rightarrow \epsilon$	$\{ M.instr = nextinstr; \}$

$B \rightarrow B_1 \mid\mid MB_2$

$\rightarrow B_1 \;\&\&\; MB_2$

$\rightarrow !\, B_1$

$\rightarrow (B_1)$

$\rightarrow E_1 \; rel \; E_2$

$\rightarrow$ **true**

$\rightarrow$ **false**

$M \rightarrow \epsilon$

Backpatching

- $makelist(i)$
  - Create a new list containing only i, return a pointer to the list
- $merge(p_1, p_2)$
  - Merge lists pointed to by p_1 and p_2 and return a pointer to the concatenated list
- $backpatch(p, i)$
  - Insert i as the target label for the statements in the list pointed to by p

Translation Scheme with Backpatching

$B \rightarrow B_1 \,\&\&\, MB_2$	$\{\ backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist);\ \}$

- Example translation scheme that can be used with bottom-up parsing
- If B_1 is false, then jump instructions in $B_1.falselist$ is part of $B.falselist$
- If B_1 is true, then target of $B_1.truelist$ is marker M

Translation Scheme with Backpatching

$B \rightarrow B_1 \,\|\|\, MB_2$	$\{ backpatch(B_1.falselist, M.instr);$ $B.truelist = merge(B_1.truelist, B_2.truelist);$ $B.falselist = B_2.falselist; \}$
$B \rightarrow B_1 \,\&\&\, MB_2$	$\{ backpatch(B_1.truelist, M.instr);$ $B.truelist = B_2.truelist;$ $B.falselist = merge(B_1.falselist, B_2.falselist); \}$
$B \rightarrow !\, B_1$	$\{ B.truelist = B_1.falselist;$ $B.falselist = B_1.truelist; \}$
$B \rightarrow (B_1)$	$\{ B.truelist = B_1.truelist;$ $B.falselist = B_1.falselist; \}$

Translation Scheme with Backpatching

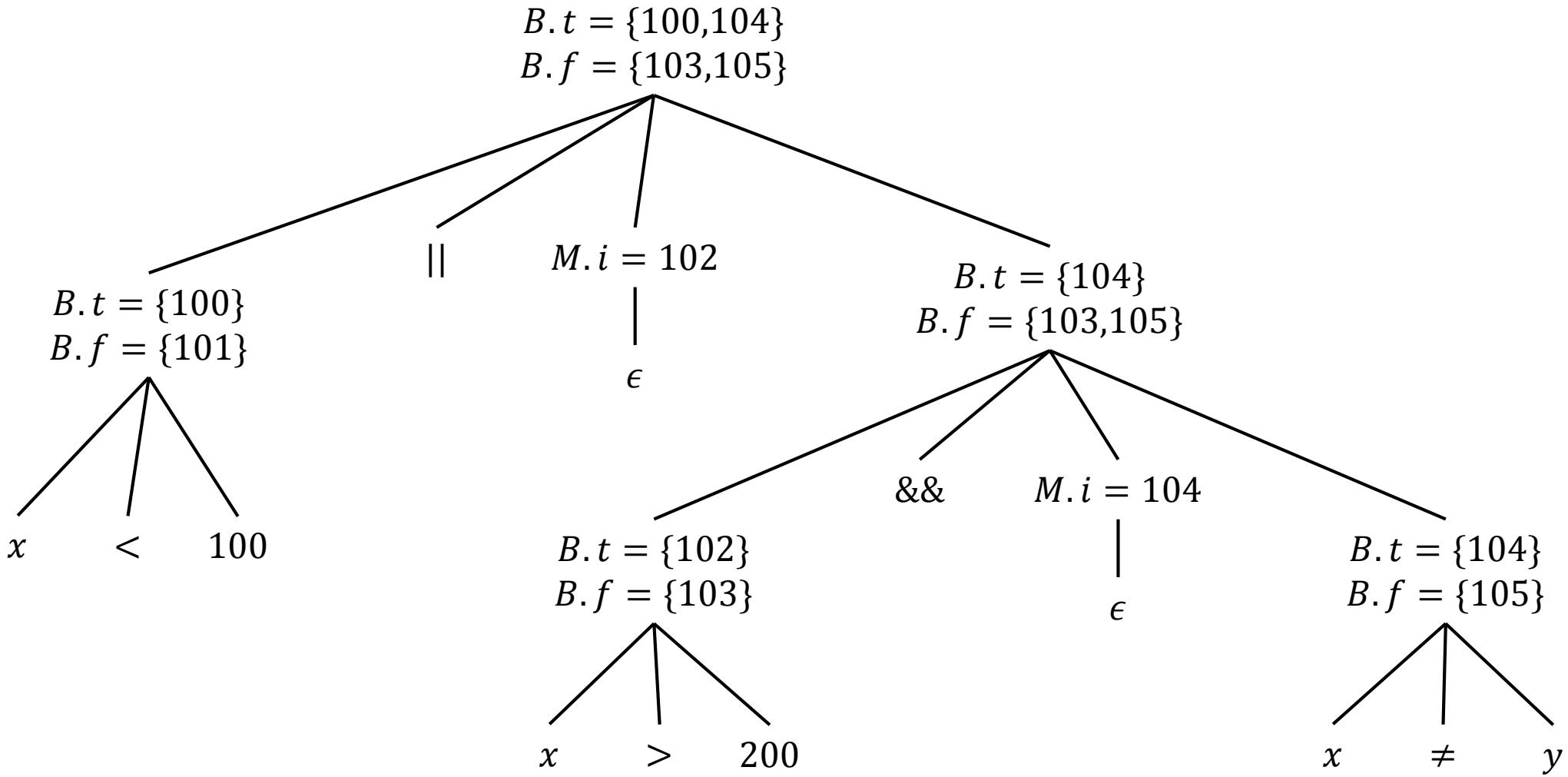
$B \rightarrow E_1\ rel\ E_2$	$\{\ B.truelist\ = makelist(nextinstr);$ $B.falselist\ = makelist(nextinstr + 1);$ $emit(\text{"if"}\ E_1.addr\ rel.op\ E_2.addr\ \text{"goto }-\text{"});$ $emit(\text{"goto }-\text{"});\ \}$
$B \rightarrow$ true	$\{\ B.truelist\ = makelist(nextinstr);$ $emit(\text{"goto }-\text{"});\ \}$
$B \rightarrow$ false	$\{\ B.falselist\ = makelist(nextinstr);$ $emit(\text{"goto }-\text{"});\ \}$
$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$

Example of Backpatching

$x < 100 \,||\, x > 200 \,\&\&\, x \neq y$

100: if $x < 100$ goto ...
101: goto ...
102: if $x > 200$ goto ...
103: goto ...
104: if $x \neq y$ goto ...
105: goto ...

Annotated Parse Tree



Example of Backpatching

$x < 100 \,||\, x > 200 \,\&\&\, x \neq y$

100: if $x < 100$ goto ...
101: goto ...
102: if $x > 200$ goto 104
103: goto ...
104: if $x \neq y$ goto ...
105: goto ...

Example of Backpatching

$x < 100 \,||\, x > 200 \,\&\&\, x \neq y$

100: if $x < 100$ goto ...
101: goto 102
102: if $x > 200$ goto 104
103: goto ...
104: if $x \neq y$ goto ...
105: goto ...

Backpatching Control Flow Statements

- S denotes a statement
- L denotes a statement list
- A is an assignment statement
- B is a Boolean expression

$$
\begin{aligned}
S &\rightarrow \textbf{if}\ (B)\ S \\
&\rightarrow \textbf{if}\ (B)\ S\ \textbf{else}\ S \\
&\rightarrow \textbf{while}\ (B)\ S \\
&\rightarrow \{\ L\ \} \mid A; \\
L &\rightarrow L\ S \mid S
\end{aligned}
$$

Backpatching Control Flow Statements

$S \rightarrow \textbf{if}\ (B)\ M\ S_1$	$\{\ backpatch(B.truelist, M.instr);$ $S.nextlist = merge(B.falselist, S_1.nextlist);\ \}$
$S \rightarrow \textbf{if}\ (B)\ M_1\ S_1\ N\ \textbf{else}\ M_2\ S_2$	$\{\ backpatch(B.truelist, M_1.instr);$ $backpatch(B.falselist, M_2.instr);$ $temp = merge(S_1.nextlist, N.nextlist);$ $S.nextlist = merge(temp, S_2.nextlist);\ \}$
$S \rightarrow \textbf{while}\ M_1\ (B)\ M_2\ S_1$	$\{\ backpatch(S_1.nextlist, M_1.instr);$ $backpatch(B.truelist, M_2.instr);$ $S.nextlist = B.falselist;$ $emit(\text{"goto"}\ M_1.instr);\ \}$

Backpatching Control Flow Statements

$S \rightarrow \{\ L\ \}$	$\{\ S.nextlist = L.nextlist;\ \}$
$S \rightarrow A;$	$\{\ S.nextlist = \mathbf{null};\ \}$
$M \rightarrow \epsilon$	$\{\ M.instr = nextinstr;\ \}$
$N \rightarrow \epsilon$	$\{ N.nextlist = makelist(nextinstr);$ $emit(\text{"goto } -\text{"});\ \}$
$L \rightarrow L_1\ M\ S$	$\{\ backpatch(L_1.nextlist, M.instr);$ $L.nextlist = S.nextlist;\ \}$
$L \rightarrow S$	$\{\ L.nextlist = S.nextlist;\ \}$

Intermediate 3AC for Procedures

$n = f(a[i]);$

$t_1 = i * 4$
$t_2 = a[t_1]$
$\text{param } t_2$
$t_3 = \text{call } f, 1$
$n = t_3$

$D \rightarrow \textbf{define}\ T\ \textbf{id}\ (\ F\ )\ \{\ S\ \}$
$F \rightarrow \epsilon \mid T\ \textbf{id}\ ,\ F$
$S \rightarrow \textbf{return}\ E\ ;$
$E \rightarrow \textbf{id}\ (\ A\ )$
$A \rightarrow \epsilon \mid E\ ,\ A$

- Generate function types:
  - $func\ pop()\colon\ void\ \rightarrow\ integer$
- Check for correct usage of the function type
- Start a new symbol table after seeing **define** and **id**

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, Chapters 2.7, 6.1-6.2, 6.4, 6.6-6.8.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, Chapter 5.