

CS 335: Intermediate Representations

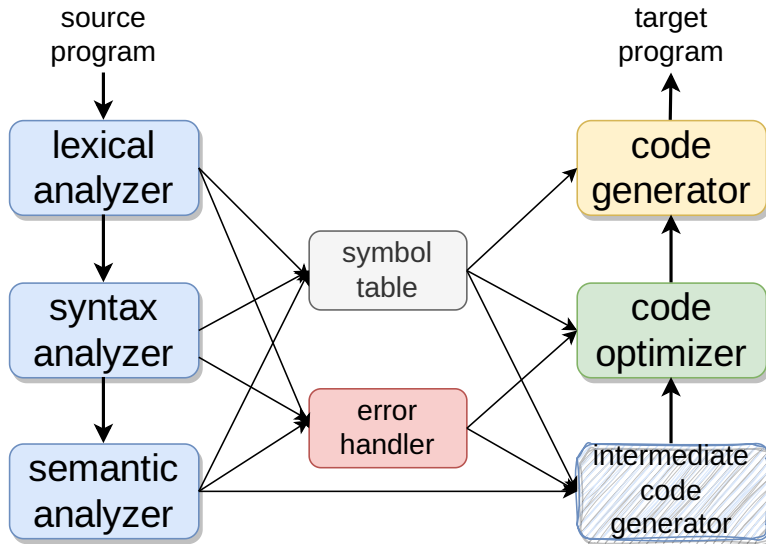
Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II



An Overview of Compilation



Intermediate Representation (IR)

Definition

IR is a **data structure** used internally by a compiler or virtual machine (VM) while translating a source program

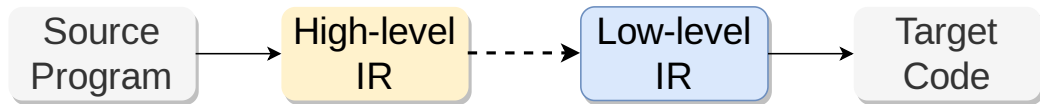
- Front end analyses a source program and creates an IR
- Back end analyses the IR and generates target code

An well-designed IR helps ease compiler development

- Plug in m front ends with n back ends to come up with $m \times n$ compilers

More on IRs

- Compilers may create several IRs during the translation process



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

Types of IR

Abstraction-based classification

High-level IR Preserves many source structures like loops and array bounds

Mid-level IR Chosen to be suitable to represent language features, often independent of the source language, and to generate code

Low-level IR Similar to the target ISA

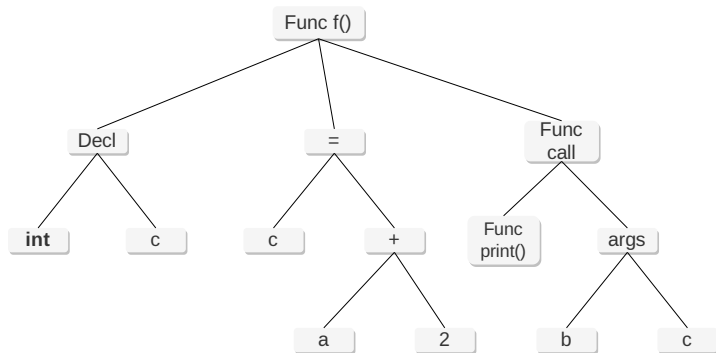
Structural classification

- Graphical, or linear, or hybrid
- Hybrid combines features of both graphical and linear IR (e.g., AST+3AC)

High-Level IR

- Maintains enough information (e.g., structured control flow, variable names, and method names) to reconstruct source code (e.g., AST)
 - ▶ 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) (e.g., three-address code)

- Independent of the source language
- Amenable to code optimizations (e.g., manipulating a list of instructions)
- Amenable for code generation for a variety of architectures

Low-level IR

- Similar to assembly code with extra pseudo-instructions plus infinite registers
- Close correspondence to the target ISA and is often architecture-dependent
- Allows low-level optimizations (e.g., instruction scheduling)

Considerations During 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 languages and architecture features
 - ▶ For example, LLVM IR and Java bytecode are IRs that have been used successfully for a number of source languages
 - ▶ Suitability for code optimization and code generation
 - ▶ Other factors like space overhead

Difficult to come up with a general IR that meets all objectives

Graphical IRs

Parse Tree (Concrete Syntax Tree)

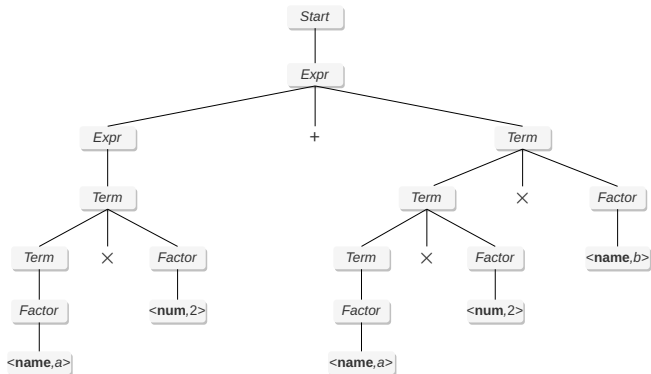
Definition

A parse tree is a graphical representation of a derivation corresponding to an input

$Start \rightarrow Expr$
 $Expr \rightarrow Expr + Term$
 $Expr \rightarrow Expr - Term$
 $Expr \rightarrow Term$
 $Term \rightarrow Term \times Factor$
 $Term \rightarrow Term \div Factor$
 $Term \rightarrow Factor$
 $Factor \rightarrow (Expr)$
 $Factor \rightarrow \text{num} \mid \text{name}$

$a \times 2 + a \times 2 \times b$

$Start \rightarrow Expr \rightarrow Expr + Term$
 $\rightarrow Term + Term$
 $\rightarrow Term \times Factor + Term$
 $\rightarrow \dots$



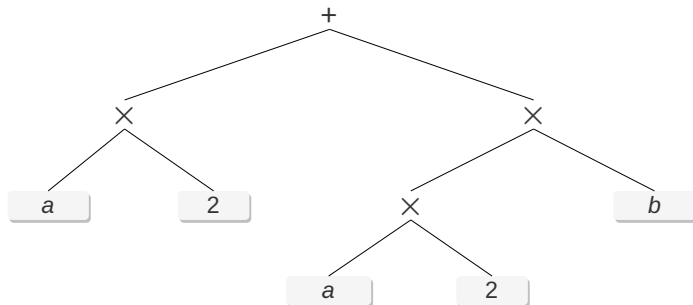
Abstract Syntax Tree (AST)

Definition

An AST is a near-source-level representation that compresses the parse tree by omitting many internal nodes corresponding to nonterminal symbols but still retains the meaning of the high-level expression

- Leaf nodes represent operands

$a \times 2 + a \times 2 \times b$

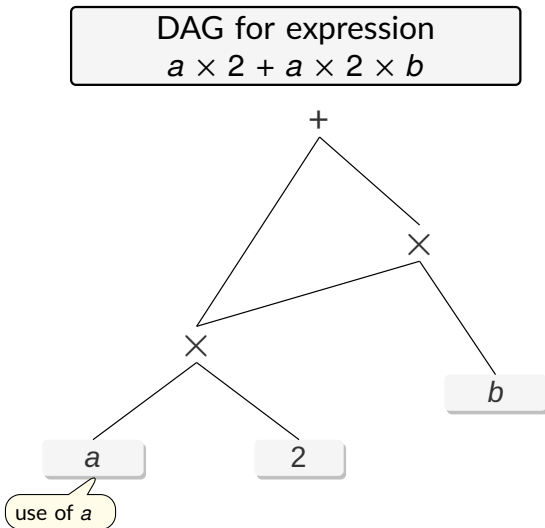


SDT for Constructing ASTs

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

Directed Acyclic Graph (DAG)

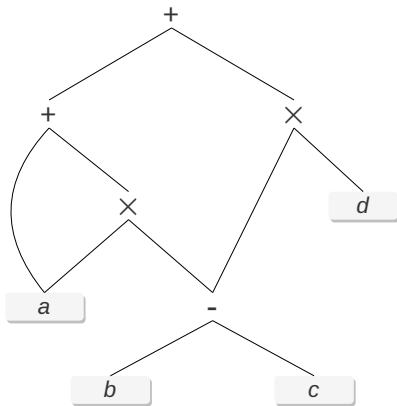
- DAGs compress ASTs and **can avoid duplicate** subtrees
 - ▶ Reduces memory footprint
 - ▶ Nodes in a DAG can have multiple parents
- DAGs **encode hints** for evaluating the expression
 - ▶ If a does not change between the two uses, then the compiler can generate code to evaluate $a \times 2$ only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



Constructing a DAG using the SDT for ASTs

DAG for expression

$a + a \times (b - c) + (b - c) \times d$



$p_1 = \text{new Leaf}(\text{id}, \text{entry}_a)$

$p_2 = \text{new Leaf}(\text{id}, \text{entry}_a) = p_1$

$p_3 = \text{new Leaf}(\text{id}, \text{entry}_b)$

$p_4 = \text{new Leaf}(\text{id}, \text{entry}_c)$

$p_5 = \text{new Node}("-", p_3, p_4)$

$p_6 = \text{new Node}("x", p_1, p_5)$

$p_7 = \text{new Node}("+", p_1, p_6)$

$p_8 = \text{new Leaf}(\text{id}, \text{entry}_b) = p_3$

$p_9 = \text{new Leaf}(\text{id}, \text{entry}_c) = p_4$

$p_{10} = \text{new Node}("-", p_3, p_4)$

$p_{11} = \text{new Leaf}(\text{id}, \text{entry}_d)$

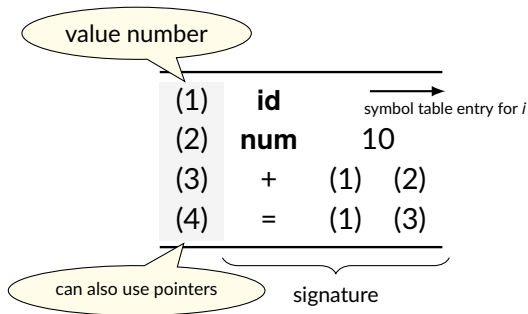
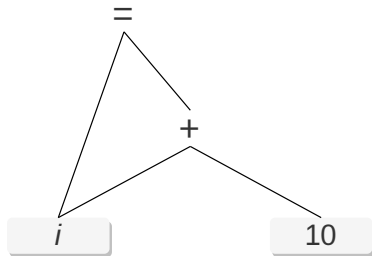
$p_{12} = \text{new Node}("x", p_5, p_{11})$

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

return existing
node if it exists

Value-Number Method for Representing DAGs

DAG nodes are often stored in an array data structure



Signature of an interior node is the triple $\langle \text{op}, l, r \rangle$

Basic Block (BB)

Definition

A BB is a **maximal** sequence of instructions with only one entry and one exit point

- Entry is at the start of the BB, and exit is from the end of the BB
- Only the start/leader instruction can be the target of a JUMP instruction
- There are no jumps in or out of the middle of a BB

- Identifying BBs

- (i) The first instruction of the input code is a **leader**
- (ii) Instructions that are targets of conditional/unconditional jumps are leaders
- (iii) Instructions that immediately follow conditional/unconditional jumps are leaders

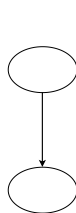
Identifying BBs

(1)	$i = 1$	
(2)	$j = 1$	
(3)	$t_1 = 10 \times i$	
(4)	$t_2 = t_1 + j$	
(5)	$t_3 = 8 \times t_2$	target
(6)	$t_4 = t_3 - 88$	
(7)	$a[t_4] = 0.0$	
(8)	$j = j + 1$	
(9)	if $j \leq 10$ goto (3)	
(10)	$i = i + 1$	
(11)	if $i \leq 10$ goto (2)	
(12)	$i = 1$	
(13)	$t_5 = i - 1$	
(14)	$t_6 = 88 \times t_5$	follows a conditional
(15)	$a[t_6] = 1.0$	
(16)	$i = i + 1$	
(17)	if $i \leq 10$ goto (13)	

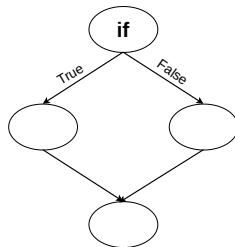
- Statements (1), (2), (3), (10), (12), and (13) are leaders
- There are six BBs: (1), (2), (3)–(9), (10)–(11), (12), (13)–(17)

Control Flow Graph (CFG)

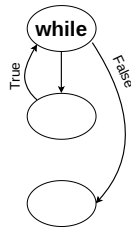
- Graphical representation of control flow during execution of a program
 - ▶ Each node represents a statement or a BB
 - ▶ An entry and an exit node are often added to a CFG for a function
 - ▶ An edge represents the **possible** transfer of control between nodes
- Used for static program analysis (e.g., compiler optimizations like instruction scheduling and global register allocation)



straight-line code



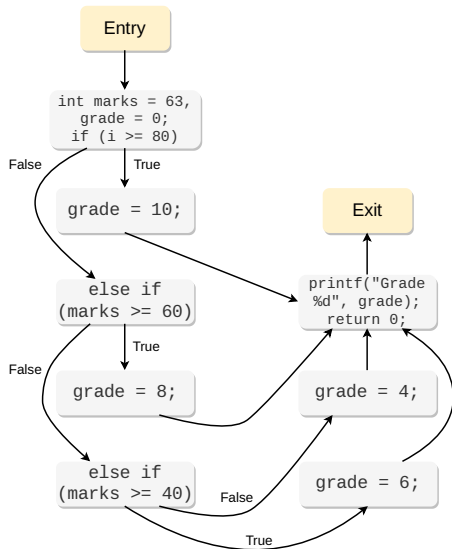
predicated code



loop-based code

Example of BBs and a CFG

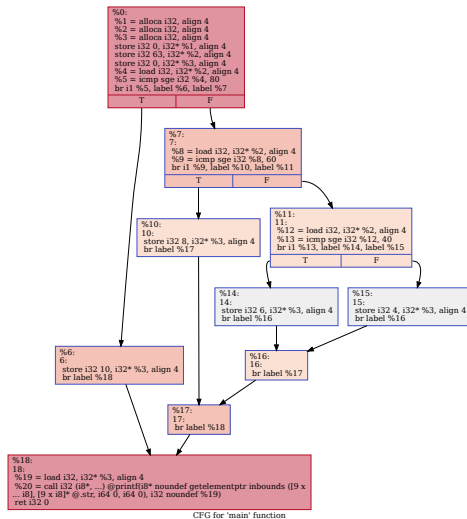
```
int main() {  
    int marks = 63, grade = 0;  
    if (marks >= 80)  
        grade = 10;  
    else if (marks >= 60)  
        grade = 8;  
    else if (marks >= 40)  
        grade = 6;  
    else  
        grade = 4;  
    printf("Grade %d", grade);  
    return 0;  
}
```



Example CFG Generated with LLVM

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

```
$ clang++ -S -emit-llvm ctrl-flow.cpp -o ctrl-flow.ll  
$ opt -analyze -enable-new-pm=0 -dot-cfg ctrl-flow.ll  
$ dot -Tpdf -o ctrl-flow.pdf .main.dot
```

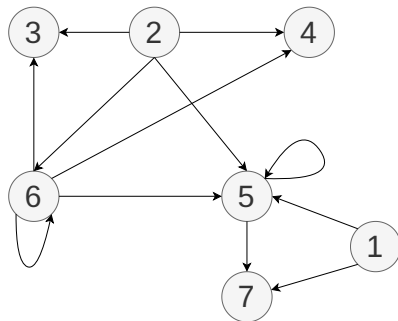


CFG for 'main' function

Data Dependence Graph (DDG)

- DDG is a graph that models the **flow of values** from definitions to uses in a code fragment
 - ▶ DDG **does not** capture the control flow
 - ▶ Out-of-order execution is akin to executing any valid partial order that respects dependences
- DDG is used in instruction scheduling and loop transformations

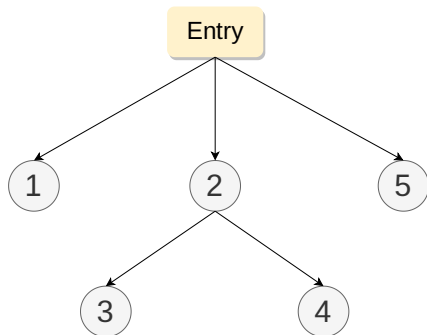
```
1.    x = 0
2.    i = 1
3.    while (i < 100)
4.        if (a[i] > 0)
5.            x = x + a[i]
6.            i = i + 1
7.    print(x)
```



Control Dependence Graph (CDG)

- 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
 - ▶ An entry node represents the start of execution

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



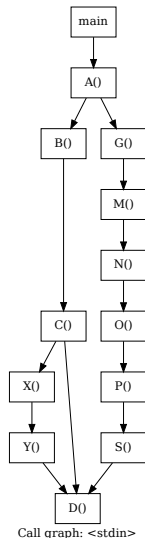
Call Graph

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

Example of a Call Graph

```
static void D() { }  
static void Y() { D(); }  
static void X() { Y(); }  
static void C() { D(); X(); }  
static void B() { C(); }  
static void S() { D(); }  
static void P() { S(); }  
static void O() { P(); }  
static void N() { O(); }  
static void M() { N(); }  
static void G() { M(); }  
static void A() { B(); G(); }  
int main() { A(); }
```

```
$ clang++ -S -emit-llvm cg.cpp -o - | opt -analyze  
-enable-new-pm=0 -dot-callgraph  
$ cat '<stdin>.callgraph.dot' | c++filt &> cg.dot  
$ dot * -Tpdf -o cg.pdf cg.dot
```



Linear IRs

Types of Linear IR

One-address code

- Models behavior of stack machines and accumulator machines
- Makes use of implicit names
- Useful where storage efficiency is important (e.g., transmission over a network)

Two-address code

- The result of one address is often overwritten with the result
- Not very popular currently

Three-address code

- Most operations take two operands and produce a result
- Similar to RISC instructions

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 and the CPython bytecode interpreter are stack-based
- x87 is a stack-based floating-point subset of the x86 ISA

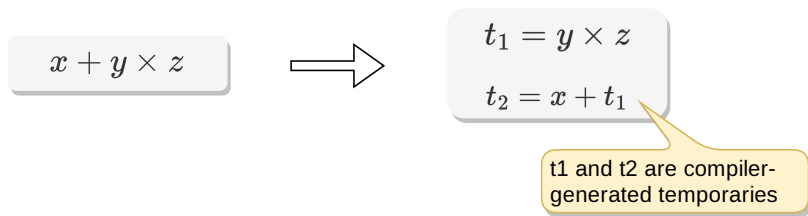
$$a - 2 \times b$$

\Rightarrow

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

Three-Address Code (3AC)

- 3AC has **at most one operator** in the RHS

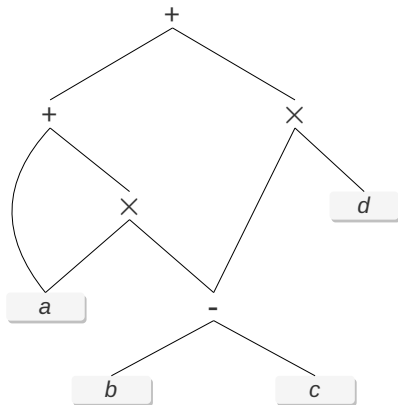


- 3AC is a linearized representation of a syntax tree where temporaries correspond to interior graph nodes
- Popularly used in code optimization and code generation
 - ▶ Distinct names for each temporary helps in optimization analyses
 - ▶ Use of names for intermediate values allows 3AC to be easily rearranged

DAG and the Corresponding 3AC

DAG for expression

$a + a \times (b - c) + (b - c) \times d$



$$t_1 = b - c$$

$$t_2 = a \times t_1$$

$$t_3 = a + t_2$$

$$t_4 = t_1 \times d$$

$$t_5 = t_3 + t_4$$

Forms of 3AC

- Operands can be program variables, constants, and temporaries
 - ▶ Variables can be pointers to the symbol table entries

do

$i = i + 1;$

while ($a[i] < v$);

Assume each array element
takes 8 units of space

Using symbolic labels

Using position numbers

L: $t_1 = i + 1$

$i = t_1$

$t_2 = i \times 8$

$t_3 = a[t_2]$

if $t_3 < v$ goto L

arbitrary start-
ing point

100: $t_1 = i + 1$

101: $i = t_1$

102: $t_2 = i \times 8$

103: $t_3 = a[t_2]$

104: if $t_3 < v$ goto 100

Forms of 3AC

- (i) Assignments of the form $x = y \text{ op } z$, $x = \text{op } y$, or $x = y$
- (ii) Unconditional jump “goto L”
- (iii) Conditional jumps of the form “if x goto L” and “if $x \text{ relop } y$ goto L”
- (iv) Procedure calls and returns of the form “param x ”, “call p, n ”, “ $y = \text{call } p, n$ ”, and “return y ”
- (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 $*x = y$

	param x_1
	param x_2
$p(x_1, x_2, \dots, x_n)$...
	param x_n
	call p, n

Example Java Code and its 3AC

```
public class Example {  
    int x;  
    double y;  
    Example(int x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    public static void main(String[]  
                                args) {  
        /* Using argument args so that  
           we can compile with javac */  
        Example a = new Example(2,3.14);  
        System.out.println(a.x);  
        System.out.println(a.y);  
    }  
}
```

```
Example.ctor:  
    beginfunc  
        // get object reference,  
        // implicit this pointer  
        t1 = popparam  
        // get offset for x  
        t2 = symtable(Example, x)  
        // get x's value  
        t3 = popparam  
        *(t1+t2) = t3  
        // get offset for y  
        t4 = symtable(Example, y)  
        // get y's value  
        t5 = popparam  
        *(t1+t4) = t5  
        return  
    endfunc
```


Example Java Code and its 3AC

Example.main:

```
beginfunc
t1 = 12 // size of object in bytes
param t1
// manipulate stack pointer
stackpointer +xxx
call allocmem 1 // 1 param
stackpointer -yyy
// Save object reference
t2 = popparam
t3 = 2
t4 = 3.14
param t2 // object reference
param t3
param t4
stackpointer +xxx
call Example.ctor
... // other lines
```

```
// get offset for x
t5 = symtable(Example, x)
t6 = *(t2+t5) // read a.x
param t6
stackpointer +xxx
call print 1
stackpointer -yyy
// get offset for y
t7 = symtable(Example, y)
t8 = *(t2+t7) // read a.y
param t8
stackpointer +xxx
call print 1
stackpointer -yyy
... // other lines
return
endfunc
```

Implementing 3AC with Quadruples

- A quadruple (or quad) is a data structure that has four fields *op*, *arg₁*, *arg₂*, and *result*
 - ▶ Instructions with unary operators do not use *arg₂*
 - ▶ Operators like param do not use both *arg₂* and *result*
 - ▶ Conditional and unconditional jumps put the target label in *result*

Consider the expression

$$a = b \times -c + b \times -c$$

$$t_1 = -c$$

$$t_2 = b \times t_1$$

$$t_3 = -c$$

$$t_4 = b \times t_3$$

$$t_5 = t_2 + t_4$$

$$a = t_5$$

	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

pointer to symbol table entry

Implementing 3AC with Triples

- Triples have three fields *op*, *arg₁*, and *arg₂*, and refer to the result of an operation by its position

Consider the expression

$$a = b \times -c + b \times -c$$

$$t_1 = -c$$

$$t_2 = b \times t_1$$

$$t_3 = -c$$

$$t_4 = b \times t_3$$

$$t_5 = t_2 + t_4$$

$$a = t_5$$

	op	arg ₁	arg ₂
0	-	c	
1	×	b	(0)
2	-	c	
3	×	b	(2)
4	+	(1)	(3)
5	=	a	(4)

refers to the position

Representation 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
	1	=	i

	op	arg ₁	arg ₂
$x = y[i]$	0	[]	y
	1	=	x

Quadruples vs Triples

Quadruples

- Requires many temporaries
- Easy to move around instructions, does not impact instructions that use results

Triples

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

Reordering Instructions with Quadruples and Triples

Quadruples

op	arg ₁	arg ₂	result
+	t_2	t_3	t_1
+	t_5	t_6	t_4
+	t_2	t_8	t_7
+	t_8	t_5	t_9
...
×	...	t_1	...
+	...	t_1	...
-	...	t_1	...

Triples

	op	arg ₁	arg ₂
(0)	+	t_2	t_3
(1)	+	t_5	t_6
(2)	+	t_2	t_8
(3)	+	t_8	t_5
...
...	×	...	(0)
...	+	...	(0)
...	-	...	(0)

Updating references is expensive

Indirect Triples Representation of 3AC

Indirect triples consist of a list of pointers to triples, rather than a list of triples

Consider the expression

$$a = b \times -c + b \times -c$$

$t_1 = -c$	(0)
$t_2 = b \times t_1$	(1)
$t_3 = -c$	(2)
$t_4 = b \times t_3$	(3)
$t_5 = t_2 + t_4$	(4)
$a = t_5$	(5)
	...

list of
instructions

simplifies reordering
of instructions

	op	arg ₁	arg ₂
0	-	c	
1	×	b	(0)
2	-	c	
3	×	b	(2)
4	+	(1)	(3)
5	=	a	(4)

decouples from the position

Importance of Naming

```
a = b + c
b = a - d
c = b + c
d = a - d
```

- Value of a and c **may** be different
- Value of b and d **are** the same

```
t1 = b
t2 = c
t3 = t1 + t2
a = t3
t4 = d
t1 = t3 - t4
b = t1
t2 = t1 + t2
c = t2
t4 = t3 - t4
d = t4
```

Assigns names to source variables, uses fewer names, but difficult to identify that b and d have the same value

```
t1 = b
t2 = c
t3 = t1 + t2
a = t3
t4 = d
t5 = t3 - t4
b = t5
t6 = t5 + t2
c = t6
t5 = t3 - t4
d = t5
```

Assigns names to destination values, uses more names, makes it explicit that b and d have the same value

Static Single Assignment (SSA) Form

- All **assignments** in SSA form 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
$p = a + b$	$p_1 = a + b$
$q = p - c$	$q_1 = p_1 - c$
$p = q \times d$	$p_2 = q \times d$
$p = e - p$	$p_3 = e - p_2$
$q = p + q$	$q_2 = p_3 + q_1$

Managing Control Flow with SSA

What about variables defined in multiple control flow paths?

```
if (flag) {  
    x1 = -1  
} else {  
    x2 = 1  
}  
  
y = ... × a
```

```
if (flag) {  
    x1 = -1  
} else {  
    x2 = 1  
}  
x3 =  $\phi(x_1, x_2)$   
y = x3 × a
```

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

Example of a Loop in SSA

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

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

What to fill in?

Example of a Loop in SSA with ϕ Functions

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

```
x0 = ...  
y0 = ...  
if (x0 ≥ 100) goto next  
loop: x1 =  $\phi$ (x0, x2)  
      y1 =  $\phi$ (y0, y2)  
      x2 = x1 + 1  
      y2 = y1 + x2  
      if (x2 < 100) goto loop  
next: x3 =  $\phi$ (x0, x2)  
      y3 =  $\phi$ (y0, y2)
```

Importance of SSA Form

- A program is in SSA form if (i) each definition has a new name, and (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 the absence of SSA

$y = 1$

$y = 2$

$x = y$

$y_1 = 1$

$y_2 = 2$

$x = y_2$

- SSA form has had a huge impact on compiler design
 - ▶ Simplifies and improves the accuracy of many optimizations (e.g., constant propagation, dead-code elimination, and register allocation)
- Most modern production compilers use SSA form (e.g., GCC, LLVM, and 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

```
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @f(i32 noundef %0, i32 noundef %1) #0 {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    store i32 %0, i32* %3, align 4
    store i32 %1, i32* %4, align 4
    %5 = load i32, i32* %3, align 4
    %6 = load i32, i32* %4, align 4
    %7 = mul nsw i32 2, %6
    %8 = add nsw i32 %5, %7
    ret i32 %8
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
    %1 = alloca i32, align 4
    store i32 0, i32* %1, align 4
    %2 = call i32 @f(i32 noundef 10, i32 noundef 20)
    ret i32 %2
}
```

- Fully-typed 3AC IR
- Scalars are in SSA form
- Supports SIMD/vector operations

What are the differences between LLVM bitcode and Java bytecode?

Symbol Table

Need for a Symbol Table

- Compilers generate meta-information about the input program during translation
 - ▶ For example, the lexeme, type, and scope of a variable, and line number for a declaration
- Information is saved in a **data structure** called symbol table
 - ▶ Alternate option 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 before use, helps determine the scope of a variable, and also helps with type checking

Desired Properties of Symbol Table

- Symbol table is accessed across several compiler phases
- Interface

`lookup(name)` Return the data 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 resize the symbol table because the number of names will vary across programs

Symbol Table Entries

- Each entry corresponds to a declaration of a name
- Format of table entries 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 when a new name is discovered
 - ▶ Attributes are filled in as new information about an existing name is discovered

Nested Scopes

```
static int w = 1; /*level 0*/
int x = 0;
void example(int a, int b) { /*level 1*/
    int c = 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, x
2b	a, x
3	c, x

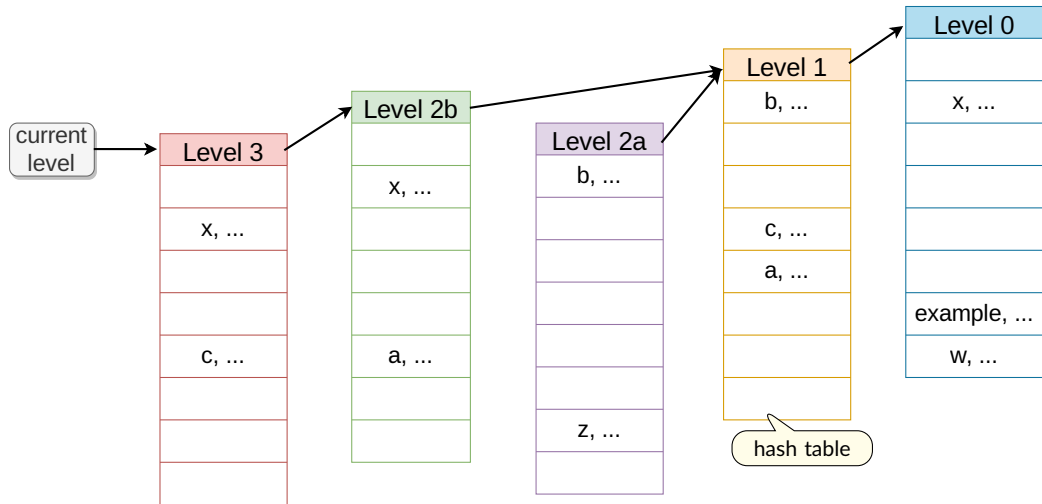
names in
each scope

$$b_1 = a_{2b} + b_1 + c_3 + w_0$$

Dealing with Nested Scopes

- **Name resolution** means resolving a reference to a name 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 the current scope and go up the hierarchy
 - ▶ Report an error only if `lookup()` fails across all levels
 - ▶ Structure is also called a “sheaf of tables”

Symbol Table Structure for Nested Scopes



Maintaining Namespace of Structure Fields

Separate table

- 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
- Need to maintain fully-qualified field names

Name Resolution for Object-Oriented Languages

- Resolution rules are slightly more involved
 - ▶ Need to search scoped symbol tables for a method, a class, and other classes in the package (e.g., for package-level variables)
- Consider resolving a name `foo` in a method `m()` in class `Klass`
 - (i) First check the lexically-scoped symbol table corresponding to `m()`
 - (ii) If `foo` not found, then search the symbol table according to the inheritance hierarchy, starting from `Klass`
 - (iii) If not found, then search the global symbol table for `foo`

Name Mangling in C++

- C++ linker supports a global namespace
 - ▶ Compiler has to pass more information about names to the linker for name resolution
- **Name mangling** facilitates function overloading and visibility within different scopes by constructing a unique string for every source-language name
 - ▶ Mangled names in C++ start with `_Z`, followed by attributes that encode information about the name
 - ▶ Name mangling applies to all C++ symbols (except for those in an `extern "C"` block)

<code>int f()</code>	<code>_Z1fv</code>
----------------------	--------------------

<code>int f(int)</code>	<code>_Z1fi</code>
-------------------------	--------------------

<code>void g()</code>	<code>_Z1gv</code>
-----------------------	--------------------

<code>namespace a {</code>	
<code> int bar;</code>	<code>_ZN1a3barE</code>
<code>}</code>	

```
$ c++filt _ZN5cs3355Outer5InnerC1ERKi  
cs3355::Outer::Inner::Inner(int const&)
```

Itanium C++ ABI: External Names (a.k.a. Mangling)

The Secret Life of C++: Symbol Mangling

GCC CPP Mangling Documentation

Practical Concern: Linking C and C++ Code

add.c

```
int add(int a, int b) {  
    return a + b;  
}
```

sub.c

```
int sub(int a, int b) {  
    return a + b;  
}
```

demo.h

```
#ifndef __LIBRARY_H__  
#define __LIBRARY_H__  
/*#ifdef __cplusplus  
extern "C" {  
    #endif */  
int add(int a, int b);  
int sub(int a, int b);  
/*#ifdef __cplusplus  
}  
#endif */  
#endif // __LIBRARY_H__
```

main.cpp

```
#include "demo.h"  
#include <cstdlib>  
#include <iostream>  
using std::cout;  
int main() {  
    int a = 0, b = 1, c = 2;  
    cout << add(a, b) << "\t"  
        << sub(c, b) << "\n";  
    return EXIT_SUCCESS;  
}
```

Practical Concern: Linking C and C++ Code

```
$ gcc -fPIC -c -o add.o add.c
$ gcc -fPIC -c -o sub.o sub.c
# Create a static library
$ ar rcs libdemo.a add.o sub.o

$ g++ -c -I. -o main.o main.cpp
$ g++ -o main main.o -L. -l:libdemo.a
```

```
$ g++ main.o -L. -l:libdemo.a -o main
/usr/bin/ld: main.o: in function 'main':
main.cpp:(.text+0x2d): undefined reference to 'add(int, int)'
/usr/bin/ld: main.cpp:(.text+0x65): undefined reference to 'sub(int, int)'
collect2: error: ld returned 1 exit status
```

```
$ readelf -s main.o
Num: Value Size Type Bind Vis Ndx Name
8: 0000000000000000 196 FUNC GLOBAL DEFAULT 1 main
9: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND _Z3addii
13: 0000000000000000 0 NOTYPE GLOBAL DEFAULT UND _Z3subii
```

Intermediate Code Generation

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

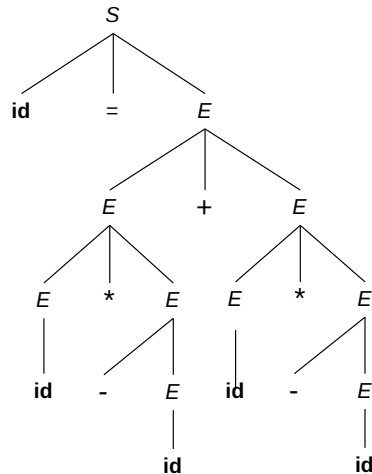
- Example of language-level abstractions
 - ▶ Identifiers, operators, expressions, statements, conditionals, iterations, functions (user-defined or libraries)
- Example of target-level abstractions
 - ▶ Memory locations, registers, stack, opcodes, addressing modes, system libraries, interface with the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

Examples of 3AC Generation

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

\Rightarrow

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



SDD for Translating Expressions to 3AC

Production

$S \rightarrow \mathbf{id} = E$

$E \rightarrow E_1 + E_2$

$E \rightarrow E_1 * E_2$

$E \rightarrow -E_1$

$E \rightarrow (E_1)$

$E \rightarrow \mathbf{id}$

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
$S \rightarrow \text{id} = E$	$S.code = E.code gen(symtop.get(\text{id.lexeme}) = "E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \text{new Temp}();$ $E.code = E_1.code E_2.code gen(E.addr = "E_1.addr" + "E_2.addr)$
$E \rightarrow E_1 * E_2$	$E.addr = \text{new Temp}();$ $E.code = E_1.code E_2.code gen(E.addr = "E_1.addr" * "E_2.addr)$
$E \rightarrow -E_1$	$E.addr = \text{new Temp}();$ $E.code = E_1.code gen(E.addr = "" - "E_1.addr)$
$E \rightarrow (E_1)$	$E.addr = E_1.addr$ $E.code = E_1.code$
$E \rightarrow \text{id}$	$E.addr = symtop.get(\text{id.lexeme})$ $E.code = ""$

symtop points to the current symbol table

Incremental Translation

Production	Semantic Rules
$S \rightarrow \mathbf{id} = E$	$gen(symtop.get(\mathbf{id.lexeme}) \text{“} = \text{”} E.addr)$
$E \rightarrow E_1 + E_2$	$E.addr = \mathbf{new Temp}();$ $gen(E.addr \text{“} = \text{”} E_1.addr \text{“} + \text{”} E_2.addr)$
$E \rightarrow E_1 * E_2$	$E.addr = \mathbf{new Temp}();$ $gen(E.addr \text{“} = \text{”} E_1.addr \text{“} * \text{”} E_2.addr)$
$E \rightarrow -E_1$	$E.addr = \mathbf{new Temp}();$ $gen(E.addr \text{“} = \text{”} - \text{”} E_1.addr)$
$E \rightarrow (E_1)$	$E.addr = E_1.addr$
$E \rightarrow \mathbf{id}$	$E.addr = symtop.get(\mathbf{id.lexeme})$

- *gen* creates a 3AC instruction and appends it to an instruction stream
- *code* attribute is not required

Translating Array References

- Grammar can generate expressions like $c + A[i][j]$
- 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 $base + i \times w_r + j \times w_e$
- 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 an array of type t , $t.width$ is the width of type t and $t.elem$ gives the element type

Production

$S \rightarrow id = E$

$S \rightarrow L = E$

$E \rightarrow E_1 + E_2$

$E \rightarrow id$

$E \rightarrow L$

$L \rightarrow id[E]$

$L \rightarrow L_1[E]$

Translating Array References

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

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 $\text{array}(3, \text{integer})$ and $w_r = 12$ Bytes
 - ▶ Type of $a[i][j]$ is $\text{array}(2, \text{array}(3, \text{integer}))$

3AC for $c + a[i][j]$

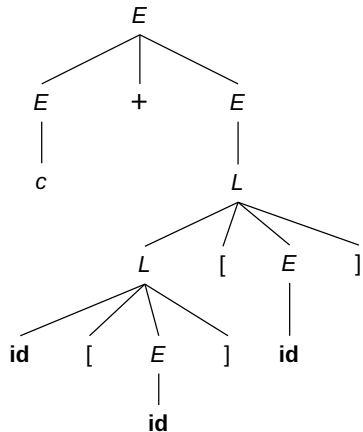
$$t_1 = i * 12$$

$$t_2 = j * 4$$

$$t_3 = t_1 + t_2$$

$$t_4 = a[t_3]$$

$$t_5 = c + t_4$$



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

3AC for $c + a[i][j]$

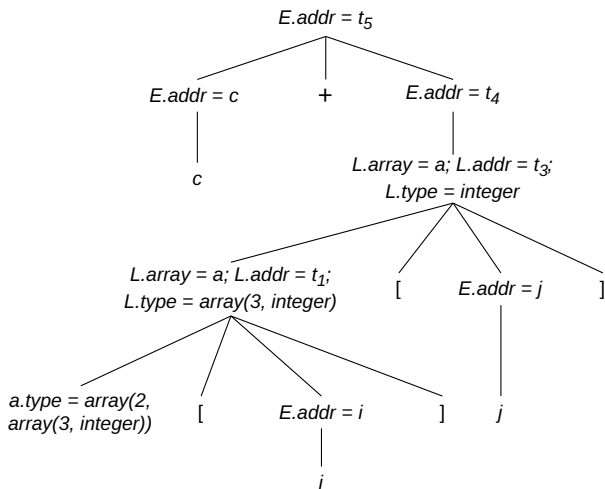
$$t_1 = i * 12$$

$$t_2 = j * 4$$

$$t_3 = t_1 + t_2$$

$$t_4 = a[t_3]$$


$$t_5 = c + t_4$$



Tracking Declarations

Goal: Lay out storage for local variables as declarations are processed

$P \rightarrow D$
$D \rightarrow D_1; D_2$
$D \rightarrow \mathbf{id} : T$
$T \rightarrow \mathbf{int}$
$T \rightarrow \mathbf{real}$
$T \rightarrow \mathbf{array}[num] \text{ of } T_1$
$T \rightarrow \uparrow T_1$



pointer type

For each name, create a 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

Tracking Declarations

$$P \rightarrow \{offset = 0; \} D$$
$$D \rightarrow D_1; D_2$$
$$D \rightarrow \mathbf{id} : T \quad \{enter(\mathbf{id.name}, T.type, offset); offset = offset + T.width; \}$$
$$T \rightarrow \mathbf{int} \quad \{T.type = integer; T.width = 4; \}$$
$$T \rightarrow \mathbf{real} \quad \{T.type = real; T.width = 8; \}$$
$$T \rightarrow \mathbf{array}[num] \text{ of } T_1 \quad \{T.type = array(num.val, T_1.type); T.width = num.val \times T_1.width; \}$$
$$T \rightarrow \uparrow T_1 \quad \{T.type = pointer(T_1.type); T.width = 4; \}$$

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

Nested Functions

Definition

A nested function is a named function that is defined within an enclosing block and is lexically scoped within it

- Enclosing block is usually a function
- Invisible outside its immediately enclosing function
- Can access local data of its enclosing functions
- Supported 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

Nested functions are supported as an extension in GNU C, but are not supported by 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);  
    /* ... */  
}
```


Keeping Track of Scope Information

$$\frac{P \rightarrow D}{D \rightarrow D_1; D_2 \mid \mathbf{id} : T \mid \mathbf{proc\ id}; D_1; S}$$

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

Example Program with Nested Procedures

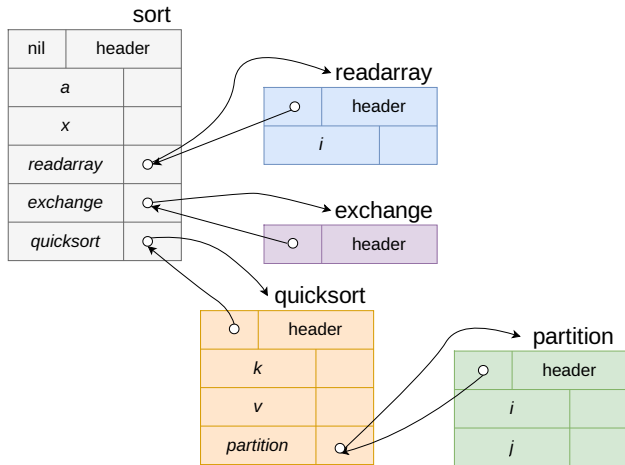
```
program sort;  
  var a : array[1..n] of integer;  
  x : integer;  
  procedure readarray;  
    var i : integer;  
    ...  
  procedure exchange(i,j : integers);  
    ...
```

```
  procedure quicksort(m,n : integer);  
    var k,v : integer;  
    function partition(x,y : integer)  
      :integer;  
      var i,j: integer;  
      ...  
    ...  
  begin  
    a[1-10]:= 0;  
    readarray;  
    quicksort(1,n);  
  end
```

Symbol Tables for 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
    a[1-10] := 0;
    readarray;
    quicksort(1,n);
  end
end
    
```



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

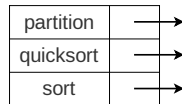
$$P \rightarrow \{t = \text{mktable}(\text{nil}); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$$
$$D\{\text{addwidth}(\text{top}(\text{tblptr}), \text{top}(\text{offset})); \text{pop}(\text{tblptr}); \text{pop}(\text{offset}); \}$$

$$D \rightarrow D_1; D_2$$

$$D \rightarrow \mathbf{id} : T \{ \text{enter}(\text{top}(\text{tblptr}), \mathbf{id.name}, T.type, \text{top}(\text{offset}));$$
$$\text{top}(\text{offset}) = \text{top}(\text{offset}) + T.width; \}$$

$$D \rightarrow \mathbf{proc id}; \{t = \text{mktable}(\text{top}(\text{tblptr})); \text{push}(t, \text{tblptr}); \text{push}(0, \text{offset}); \}$$
$$D_1; S \quad \{t = \text{top}(\text{tblptr}); \text{addwidth}(t, \text{top}(\text{offset})); \text{pop}(\text{tblptr}); \text{pop}(\text{offset});$$
$$\text{enterproc}(\text{top}(\text{tblptr}), \mathbf{id.name}, t); \}$$

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



tblptr

Boolean Expressions

Used to compute logical values (e.g., $x = \text{true}$) and influence control flow (e.g., `if (E) then S`)

$B \rightarrow B \mid \mid B$

$\rightarrow B \&\& B$

$\rightarrow !B$

$\rightarrow (B)$

$\rightarrow E \text{ relop } E$

$\rightarrow \text{true}$

$\rightarrow \text{false}$

$\text{relop} \rightarrow < \mid \leq \mid = \mid \neq \mid > \mid \geq$

How do we represent the value of Boolean expressions?

- (i) Evaluate similar to arithmetic expressions
 - ▶ True can be 1 (or any nonzero value) and False can be 0
- (ii) Implement using flow of control, value is given by the position reached

Short Circuit Code

Short circuit code translates to conditional and unconditional jumps

- Target is *B.true* if *B* is true or *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 ≠ y  
  x = 0;
```

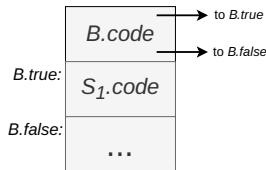


```
if x < 100 goto L2  
ifFalse x > 200 goto L1  
ifFalse x ≠ y goto L1  
L2 : x = 0  
L1 : ...
```

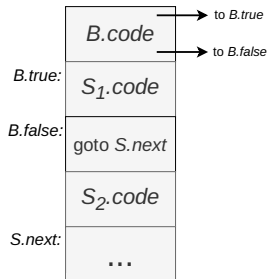
Control Flow

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

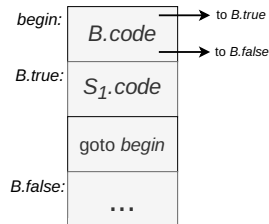
- Synthesized attributes $S.code$ and $B.code$ store 3AC
- Inherited attributes $S.next$, $B.true$, and $B.false$ are for jumps



`if (B) S1`



`if (B) S1 else S2`



`while (B) S1`

Generating 3AC for Boolean Expressions

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

SDD for Control Flow Statements

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

Example of Control Flow Translation

```
while (a < b)
  if (c < d)
    x = y + z;
  else
    x = y - z;
```

$P \rightarrow S$
 $\rightarrow \text{while } (B_1) S_1$
 $\rightarrow \text{while } (E_1 \text{ relop } E_2) S_1$
 $\rightarrow \text{while } (E_1 \text{ relop } E_2) \text{ if } (B_2) S_2 \text{ else } S_3$
 $\rightarrow \dots$

```
L2: // begin
    if (a < b) goto L3
    goto L1 // S.next
L3: // label(B1.true)
    // S1.code starts
    if (c < d) goto L4 // B2.true
    goto L5 // B2.false
L4: x = y + z // S2.code
    goto L2 // S1.next
L5: x = y - z // S3.code
    goto L2 // begin
L1: // label(S.next)
```

Redundant Gotos During Control Flow Translation

$P \rightarrow S$

$\rightarrow \text{if } (B) S$

$\rightarrow \text{if } (B_1 || B_2) S$

$\rightarrow \text{if } (B_1 || B_2 \&\& B_3) S$

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

\Downarrow

```
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 :
```

\Rightarrow

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

Avoiding Redundant Gotos

Discussed 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
---------------------	----------------------------------------------------------

```
L3 : if  $x > 200$  goto  $L_4$   
      goto  $L_1$   
L4 : ...
```



```
L3 : ifFalse  $x > 200$  goto  $L_1$   
L4 : ...
```

avoids a jump

Avoiding Redundant Gotos

A more efficient way is to let the control fall through, avoiding a jump

- *fall* is a special label indicating no jump

$S \rightarrow \text{if } (B) S_1$	$B.true = fall; B.false = S_1.next = S.next;$ $S.code = B.code S_1.code;$
------------------------------------	---------------------------------------------------------------------------------

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

► Example

Challenge in Generating Code

- How do you associate labels to instruction addresses?
 - ▶ Consider the statement **if** (B) S_1
 - ▶ Given B is translated before S_1 in a pass, how do we set the target of $B.false$?

	$B.true = fall;$
$S \rightarrow \text{if } (B) S_1$	$B.false = S_1.next = S.next;$
	$S.code = B.code S_1.code$

	<code>ifFalse (E_1 relop E_2) goto L_1</code>
L_2 :	<code>code for TRUE block</code>
L_1 :	<code>next statement</code>

A **separate pass** is needed to bind labels (passed via inherited attributes) to addresses of instructions (forward jumps)

Backpatching

One-Pass Code Generation using Backpatching

- Backpatching generates 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
- Nonterminal B has two synthesized attributes
 - ▶ *truelist* and *falselist* — Lists of jump instructions to which control goes if B is true or false
- S has a synthesized attribute *nextlist* denoting a list of jumps to the instruction immediately following S

$$\begin{aligned} B &\rightarrow B_1 \mid \mid MB_2 \\ &\rightarrow B_1 \ \&\& \ MB_2 \\ &\rightarrow !B_1 \\ &\rightarrow (B_1) \\ &\rightarrow E_1 \text{ relop } E_2 \\ &\rightarrow \text{true} \\ &\rightarrow \text{false} \\ M &\rightarrow \epsilon \end{aligned}$$

Instructions will require a label

Translation Scheme with Backpatching

- Assume instructions are stored in an array of quadruples
- Labels represent array indices

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 instructions in the list pointed to by p

Need for Marker Nonterminals

We insert a marker nonterminal M in the grammar to pick up the index of the next instruction (i.e., quadruple)

- Allows executing the action during a reduction in bottom-up parsing

$$M \rightarrow \epsilon \quad \{M.instr = nextinstr; \}$$
$$\begin{aligned} B &\rightarrow B_1 \mid \mid MB_2 \\ &\rightarrow B_1 \&\& MB_2 \\ &\rightarrow !B_1 \\ &\rightarrow (B_1) \\ &\rightarrow E_1 \text{ relop } E_2 \\ &\rightarrow \text{true} \\ &\rightarrow \text{false} \\ M &\rightarrow \epsilon \end{aligned}$$

Instructions will require a label

Translation Scheme with Backpatching

Example translation scheme that can be used with bottom-up parsing

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

- 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 the marker M
 - ▶ Each instruction in $B_1.truelist$ will receive $M.instr$ as its target label

Translation Scheme with Backpatching

$B \rightarrow B_1 \ \&\& \ MB_2$	$\{ \text{backpatch}(B_1.\text{truelist}, M.\text{instr});$ $B.\text{falselist} = \text{merge}(B_1.\text{falselist}, B_2.\text{falselist}); \}$
$B \rightarrow B_1 \ \ MB_2$	$\{ \text{backpatch}(B_1.\text{falselist}, M.\text{instr});$ $B.\text{truelist} = \text{merge}(B_1.\text{truelist}, B_2.\text{truelist});$ $B.\text{falselist} = B_2.\text{falselist}; \}$
$B \rightarrow !B_1$	$\{ B.\text{truelist} = B_1.\text{falselist}; B.\text{falselist} = B_1.\text{truelist}; \}$
$B \rightarrow (B_1)$	$\{ B.\text{truelist} = B_1.\text{truelist}; B.\text{falselist} = B_1.\text{falselist}; \}$
$B \rightarrow E_1 \ \text{relop} \ E_2$	$\{ B.\text{truelist} = \text{makelist}(\text{nextinstr}); B.\text{falselist} = \text{makelist}(\text{nextinstr} + 1);$ $\text{emit}(\text{"if"}(E_1.\text{addr} \ \text{relop.op} \ E_2.\text{addr}) \ \text{"goto"} \ \underline{\hspace{1cm}}); \text{emit}(\text{"goto"} \ \underline{\hspace{1cm}}); \}$
$B \rightarrow \text{true}$	$\{ B.\text{truelist} = \text{makelist}(\text{nextinstr}); \text{emit}(\text{"goto"} \ \underline{\hspace{1cm}}); \}$
$B \rightarrow \text{false}$	$\{ B.\text{falselist} = \text{makelist}(\text{nextinstr}); \text{emit}(\text{"goto"} \ \underline{\hspace{1cm}}); \}$
$M \rightarrow \epsilon$	$\{ M.\text{instr} = \text{nextinstr}; \}$

Example of Backpatching

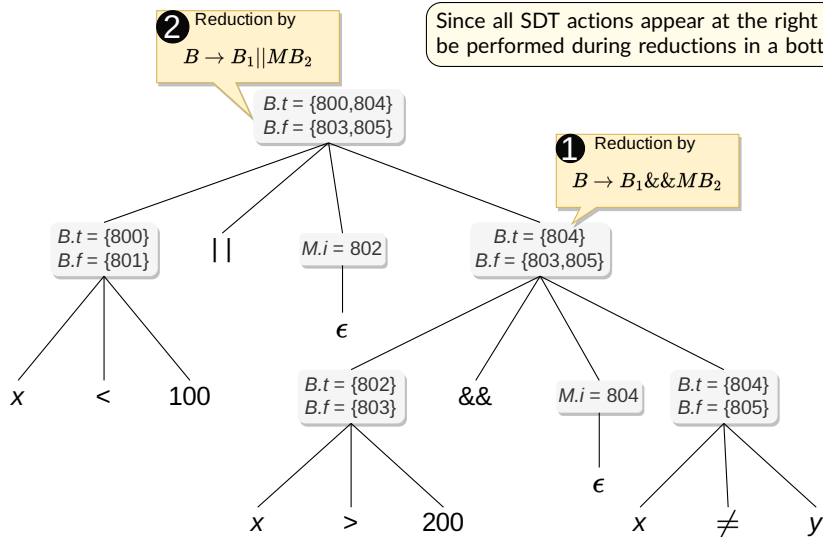
$x < 100 \parallel x > 200 \ \&\& \ x \neq y$

\Rightarrow

arbitrary start

```
800:  if x < 100 goto ...  
801:  goto ...  
802:  if x > 200 goto ...  
803:  goto ..  
804:  if x ≠ y goto ...  
805:  goto ...
```

Annotated Parse Tree



Example of Backpatching

$x < 100 \parallel x > 200 \ \&\& \ x \neq y$

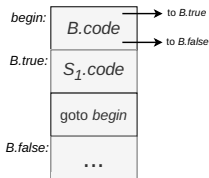
\Rightarrow

```
800:   if x < 100 goto ...  
801:   goto 802  
802:   if x > 200 goto 804  
803:   goto ..  
804:   if x ≠ y goto ...  
805:   goto ...
```

- Entire expression is true if the goto target at 800 or 804 is reached
- Entire expression is false if the goto target at 803 or 805 is reached
- Pending jump targets will be filled in **later** as more targets become known

Backpatching Control Flow Statements

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

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

$$S \rightarrow \text{while } M_1 (B) M_2 S_1$$
$$\begin{aligned} &\{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr}); \\ &\quad \text{backpatch}(B.\text{truelist}, M_2.\text{instr}); \\ &\quad S.\text{nextlist} = B.\text{falselist}; \text{emit}(\text{"goto"} M_1.\text{instr}); \} \end{aligned}$$

Backpatching Control Flow Statements

$S \rightarrow \text{if } (B) MS_1$	$\{ \text{backpatch}(B.\text{truelist}, M.\text{instr});$ $S.\text{nextlist} = \text{merge}(B.\text{falselist}, S_1.\text{nextlist}); \}$
$S \rightarrow \text{if } (B) M_1 S_1 N \text{ else } M_2 S_2$	$\{ \text{backpatch}(B.\text{truelist}, M_1.\text{instr}); \text{backpatch}(B.\text{falselist}, M_2.\text{instr});$ $\text{temp} = \text{merge}(S_1.\text{nextlist}, N.\text{nextlist});$ $S.\text{nextlist} = \text{merge}(\text{temp}, S_2.\text{nextlist}); \}$
$S \rightarrow \text{while } M_1 (B) M_2 S_1$	$\{ \text{backpatch}(S_1.\text{nextlist}, M_1.\text{instr}); \text{backpatch}(B.\text{truelist}, M_2.\text{instr});$ $S.\text{nextlist} = B.\text{falselist}; \text{emit}(\text{"goto"} M_1.\text{instr}); \}$
$S \rightarrow \{L\}$	$\{ S.\text{nextlist} = L.\text{nextlist}; \}$
$S \rightarrow A;$	$\{ S.\text{nextlist} = \text{null}; \}$
$M \rightarrow \epsilon$	$\{ M.\text{instr} = \text{nextinstr}; \}$
$N \rightarrow \epsilon$	$\{ N.\text{nextlist} = \text{makelist}(\text{nextinstr}); \text{emit}(\text{"goto"} ______); \}$
$L \rightarrow L_1 MS$	$\{ \text{backpatch}(L_1.\text{nextlist}, M.\text{instr}); L.\text{nextlist} = S.\text{nextlist}; \}$
$L \rightarrow S$	$\{ L.\text{nextlist} = S.\text{nextlist}; \}$

assumed to be a termination production,
hence backpatching is not required

Example of Backpatching Control Flow Statements

```
while (a < b)
  if (c < d)
    x = y + z;
  else
    x = y - z;
```

```
800: if (a < b) goto ...
801: goto ...
802: if (c < d) goto ...
803: goto ...
804: x = y + z
805: goto ...
806: x = y - z
807: goto ...
808:
```

$P \rightarrow S$

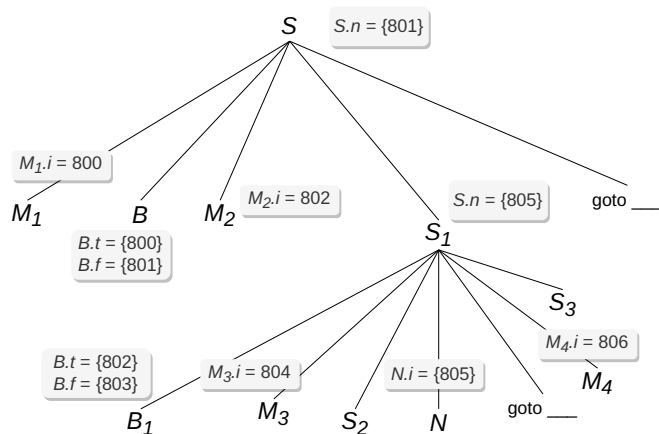
→ **while** $M_1(B) M_2 S_1$

→ **while** $M_1(E_1 \text{ relop } E_2) M_2 S_1$

→ **while** $M_1(E_1 \text{ relop } E_2) M_2$ **if** $(B_1) M_3 S_2 N$ **else** $M_4 S_3$

→ ...

Example of Backpatching Control Flow Statements



```
800: if (a < b) goto 802
801: goto ...
802: if (c < d) goto 804
803: goto 806
804: x = y + z
805: goto 800
806: x = y - z
807: goto ...
808:
```

Intermediate 3AC for Procedures

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

\Rightarrow

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

$D \rightarrow \mathbf{define } T \mathbf{id } (F) \{S\}$

$F \rightarrow T \mathbf{id }, F \mid \epsilon$



$S \rightarrow \mathbf{return } E;$

$E \rightarrow \mathbf{id } (A)$

$A \rightarrow E, A \mid \epsilon$

- (i) Generate function types
 - Example: *func pop(): void → integer*
- (ii) Check for correct usage of the function type
- (iii) Start a new symbol table after seeing **define** and **id**

References

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