Compiler Optimization and Code Generation

Professor: Sc.D., Professor

Vazgen Melikyan

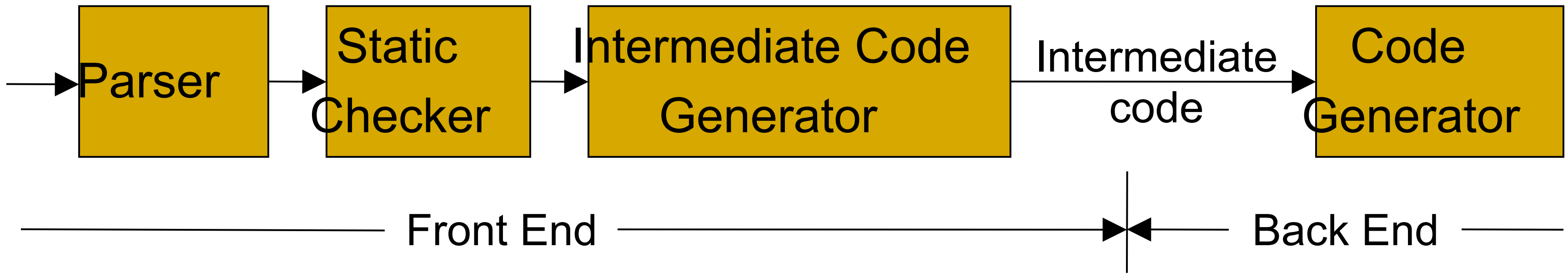
Course Overview

- Introduction: Overview of Optimizations
  - 1 lecture
- Intermediate-Code Generation
  - 2 lectures
- Machine-Independent Optimizations
  - 3 lectures
- Code Generation
  - 2 lectures

Intermediate-Code Generation

Logical Structure of a Compiler Front End

- In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code.



- Static checking:
  - Type checking: ensures that operators are applied to compatible operands
  - Any syntactic checks that remain after parsing

Type Checking

- Each operation in a language
  - Requires the operands to be predefined types of values
  - Returns an expected type of value as result
- When operations misinterpret the type of their operands, the program has a type error
- Compilers must determine a unique type for each expression
  - Ensure that types of operands match those expected by an operator
  - Determine the size of storage required for each variable
    - Calculate addresses of variable and array accesses

Value of Intermediate Code Generation

- Typically the compiler needs to produce machine code or assembler for several target machines.
- The intermediate code representation is neutral in relation to target machine, so the same intermediate code generator can be shared for all target languages.
- Less work in producing a compiler for a new machine.
- Machine independent code optimization can be applied.

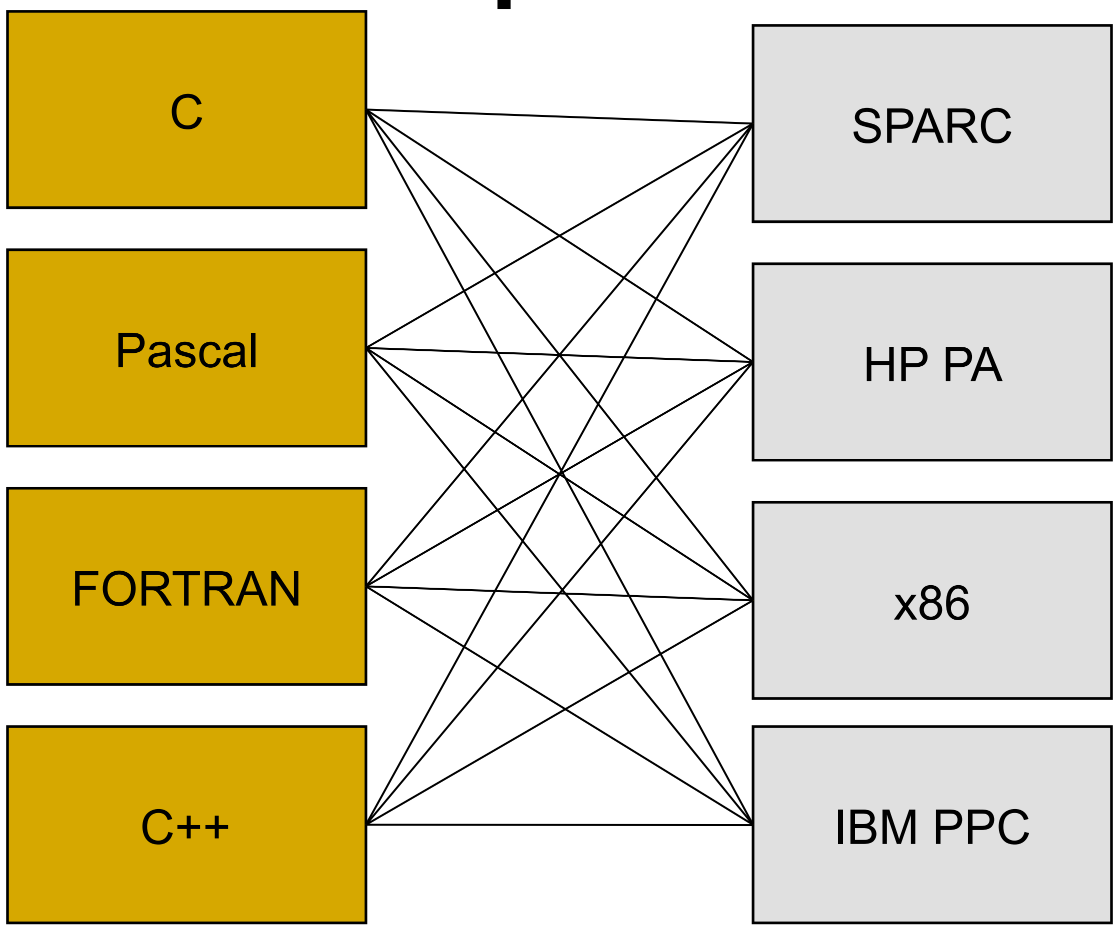
Main Methods of Intermediate Code (IC) Generation

- Two main forms used for representing intermediate code:
  - Postfix Notation: the abstract syntax tree is linearized as a sequence of data references and operations.
    - For instance, the tree for : a * (9 + d) can be mapped to the equivalent postfix notation: a9d+*
  - Quadruples: All operations are represented as a 4-part list:
    - (op, arg1, arg2, result)
      - E.g., x := y + z -> (+ y z x)

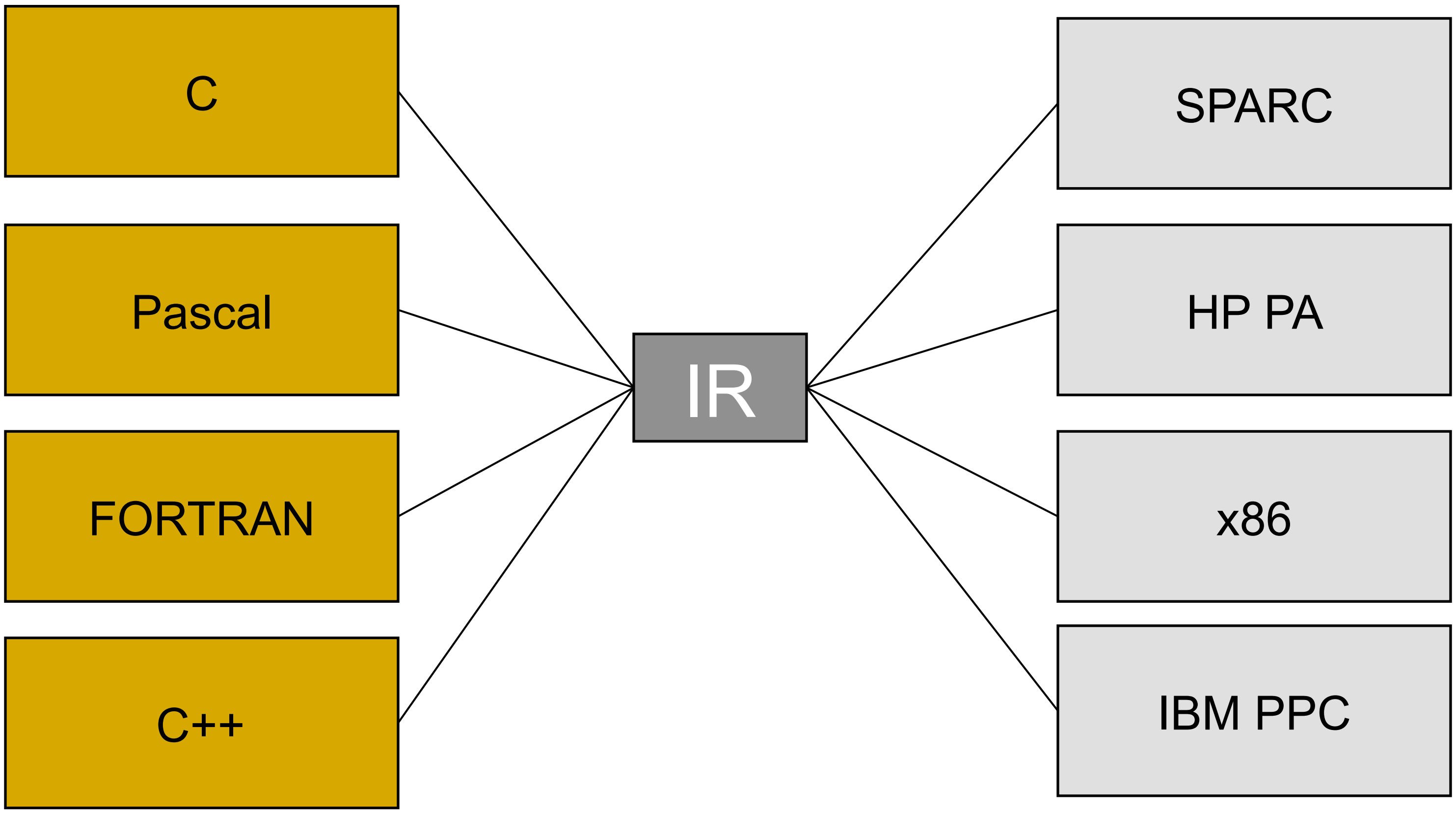
Commonly Used Intermediate Representations

- Possible IR forms
  - Graphical representations: such as syntax trees, AST (Abstract Syntax Trees), DAG
  - Postfix notation
  - Three address code
  - SSA (Static Single Assignment) form

Compiling Process without Intermediate Representation

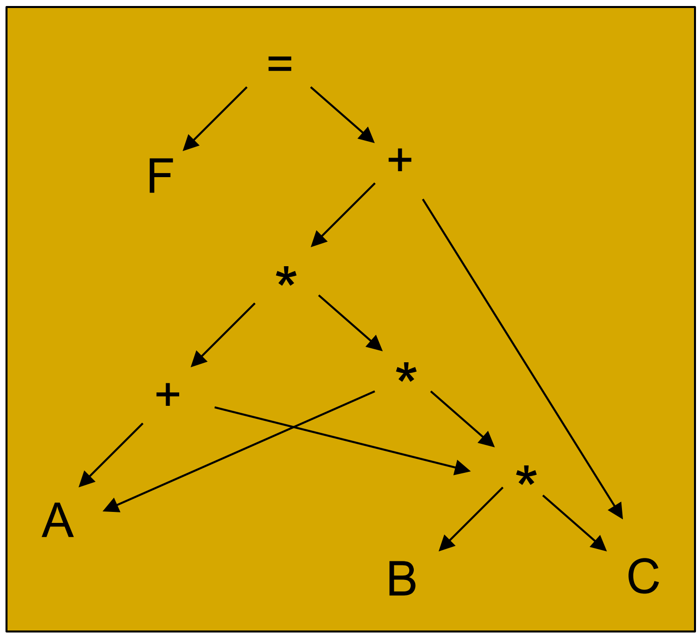


Compiling Process with Intermediate Representation



Direct Acyclic Graph (DAG) Representation

- Example: F = ((A+B*C) * (A*B*C))+C



Postfix Notation: PN

- A mathematical notation wherein every operator follows all of its operands.

  Example: PN of expression a* (b+a) is abc+*
- Form Rules:
  - If E is a variable/constant, the PN of E is E itself.
  - If E is an expression of the form E1 op E2, the PN of E is E1 ’E2 ’op (E1 ’ and E2 ’ are the PN of E1 and E2, respectively.)
  - If E is a parenthesized expression of form (E1), the PN of E is the same as the PN of E1.

Three Address Code

- The general form: x = y **op** z
  - x,y,and z are names, constants, compiler-generated temporaries
  - op stands for any operator such as +,-,…
- A popular form of intermediate code used in optimizing compilers is three-address statements.
  - Source statement: f = a+b*c+e

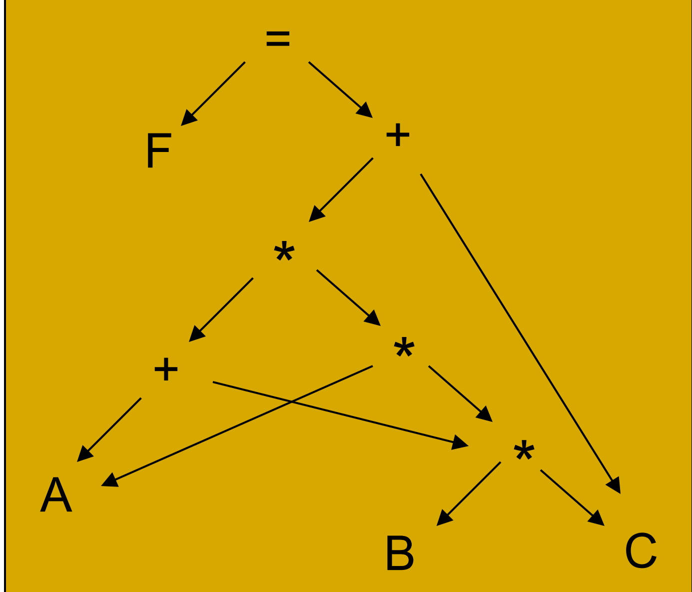
    Three address statements with temporaries t1 and t2:

    t1 = b* c
    t2 = a + t1
    f = t2 + e

DAG vs. Three Address Code

- Three address code is a linearized representation of a syntax tree (or a DAG) in which explicit names (temporaries) correspond to the interior nodes of the graph.

Expression: F = ((A+B*C) * (A*B*C))+C




```
T1 := A
T2 := C
T3 := B * T2
T4 := T1+T3
T5 := T1*T3
T6 := T4 * T5
T7 := T6 + T2
F := T7
```

```
T1 := B * C
T2 := A+T1
T3 := A*T1
T4 := T2*T3
T5 := C
T6 := T4 + T5
D := T6
```

Types of Three-Address Statements

- Assignment statements:
  - x := y op z, where op is a binary operator
  - x := y op z, where op is a binary operator
- Copy statements
  - x := y
- The unconditional jumps:
  - goto L
- Conditional jumps:
  - if x relop y goto L
- param x and call p, n and return y relating to procedure calls
- Assignments:
  - x := y[i]
  - x[i] := y
- Address and pointer assignments:
  - x := &y, x := *y, and *x = y

Generating Three-Address Code

- Temporary names are made up for the interior nodes of a syntax tree
- The synthesized attribute S.code represents the code for the assignment S
- The nonterminal E has attributes:
  - E.place is the name that holds the value of E
  - E.code is a sequence of three-address statements evaluating E
- The function newtemp returns a sequence of distinct names
- The function newlabel returns a sequence of distinct labels

Assignments

Production	Semantic Rules
S -> **id** := E	S.code := E.code || gen(**id**.place ':=' E.place)
E -> E1 + E2	E.place := newtemp; E.code := E1.code || E2.code || gen(E.place ':=' E1.place '+' E2.place)
E -> E1 * E2	E.place := newtemp; E.code := E1.code || E2.code || gen(E.place ':=' E1.place '*' E2.place)
E -> -E1	E.place := newtemp; E.code := E1.code || gen(E.place ':=' 'uminus'E1.place)
E -> (E1)	E.place := E1.place; E.code := E1.code
E -> **id**	E.place := **id.place;** E.code := "

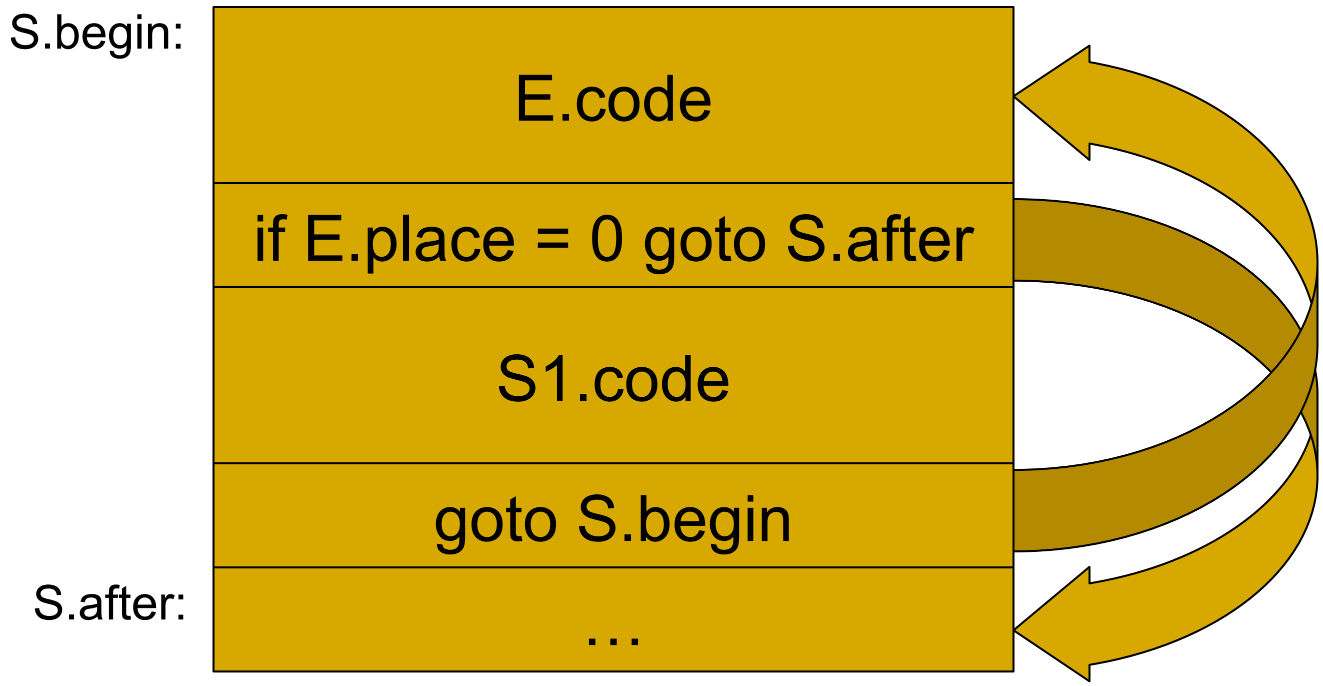
Incremental Translation

- Code attributes can be long strings, so they are usually generated incrementally.
- Instead of building up E.code only the new three-address instructions are generated.
- In the incremental approach, gen not only constructs a three-address instruction, it appends the instruction to the sequence of instructions generated so far.

Incremental Translation: Examples

Production	Semantic Rules
S -> **id** := E	gen(top.gen(**id**.lexeme) ':=' E.addr);
E -> E1 + E2	E.addr := new Temp(); gen(E.addr ':=' E1.addr '+' E2.addr);
E -> -E1	E. addr := new Temp(); gen(E. addr ':=' 'minus' E1. addr) ;
E -> (E1)	E.addr := E1.addr
E -> **id**	E.addr := top.get(**id**.lexeme);

While Statement



Quadruples

- A quadruple is a record structure with four fields: op, arg1, arg2, and result
  - The op field contains an internal code for an operator
  - Statements with unary operators do not use arg2
  - Operators like param use neither arg2 nor result
  - The target label for conditional and unconditional jumps are in result
- The contents of fields arg1, arg2, and result are typically pointers to symbol table entries
  - If so, temporaries must be entered into the symbol table as they are created
  - Obviously, constants need to be handled differently

Quadruples: Example

	op	arg1	arg2	result
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	assign	t5		a

Triples

- Triples refer to a temporary value by the position of the statement that computes it
  - Statements can be represented by a record with only three fields: op, arg1, and arg2
  - Avoids the need to enter temporary names into the symbol table
- Contents of arg1 and arg2:
  - Pointer into symbol table (for programmer defined names)
  - Pointer into triple structure (for temporaries)
  - Of course, still need to handle constants differently

Triples : Example

	op	arg1	result
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Declarations

- A symbol table entry is created for every declared name
- Information includes name, type, relative address of storage, etc.
- Relative address consists of an offset:
  - Offset is from the field for local data in an activation record for locals to procedures
- Types are assigned attributes type and width (size)
- Becomes more complex if dealing with nested procedures or records

Declarations: Example

Production	Semantic Rules
P -> D	offset := 0
D -> D ; D	
D -> id : T	enter(id.name, T.type, offset); offset := offset + T.width
T -> integer	T.type := integer; T.width := 4
T -> real	T.type := real T.width := 8
T -> array[num] of T1	T.type := array(num, T1.type); T.width := num * T1.width
T -> ↑T1	T.type := pointer(T1.type); T.width := 4

Translating Assignments

Production	Semantic Rules
S -> id := E	p := lookup(id.name); if p != NULL then emit(p ':=' E.place) else error
E -> E1 + E2	E.place := newtemp; emit(E.place ':=' E1.place '+' E2.place)
E -> E1 * E2	E.place := newtemp; emit(E.place ':=' E1.place '*' E2.place)
E -> -E1	E.place := newtemp; emit(E.place ':=' 'uminus' E1.place)
E -> (E1)	E.place := E1.place
E -> id	p := lookup(id.name); if p != NULL then E.place := p else error

Addressing Array Elements

- The location of the i-th element of array A is:

  $$\text{base} + (i - \text{low}) * w$$

  - w is the width of each element
  - Low is the lower bound of the subscript
  - Base is the relative address of a[low]
- The expression for the location can be rewritten as:

  $$i * w + (\text{base} - \text{low} * w)$$

  - The subexpression in parentheses is a constant
  - That subexpression can be evaluated at compile time

Semantic Actions for Array References

Production	Semantic Rules
S -> id := E	gen(top.get(id.lexeme) ':=' E.addr)
E -> E1 + E2	E.addr=newTemp(); gen(E. addr '=' E1. addr '+' E2. addr) ;
| L = E	gen(L. addr. base '['L. addr ']' '=' E. addr);
| id	E.addr = top.get(id.lexeme)
L -> id [E]	L.array = top.get(id.lexeme); L.type = L.array.type.elem; L. addr = new Temp 0; gen(L.addr '=' E.addr '*' L.type.width);

Type Conversions

- There are multiple types (e.g. integer, real) for variables and constants
  - Compiler may need to reject certain mixed-type operations
  - At times, a compiler needs to general type conversion instructions
- An attribute E.type holds the type of an expression

Boolean Expressions

- Boolean expressions compute logical values
- Often used with flow-of-control statements
- Methods of translating Boolean expression:
  - Numerical:
    - True is represented as 1 and false is represented as 0
    - Nonzero values are considered true and zero values are considered false
  - Flow-of-control:
    - Represent the value of a Boolean by the position reached in a program
    - Often not necessary to evaluate entire expression

Boolean Expressions: Examples

Production	Semantic Rules
E -> E1 or E2	E1.true := E.true; E1.false := newlabel; E2.true := E.true; E2.false := E.false; E.code := E1.code || gen(E1.false ':') || E2.code
E -> E1 and E2	E1.true := newlabel; E1.false := E.false; E2.true := E.true; E2.false := E.false; E.code := E1.code || gen(E1.true ':') || E2.code
E -> not E1	E1.true := E.false; E1.false := E.true; E.code := E1.code

Boolean Expressions: Examples (2)

Production	Semantic Rules
E -> (E1)	E1.true := E.true; E1.false := E.false; E.code := E1.code
E -> id1 relop id2	E.code := gen('if' id.place relop.op id2.place 'goto' E.true) || gen('goto' E.false)
E -> true	E.code := gen('goto' E.true)
E -> false	E.code := gen('goto' E.false)

Flow-of-Control

- The function newlabel returns a new symbolic label each time it is called
- Each Boolean expression has two new attributes:
  - E.true is the label to which control flows if E is true
  - E.false is the label to which control flows if E is false
- Attribute S.next of a statement S:
  - Inherited attribute whose value is the label attached to the first instruction to be executed after the code for S
  - Used to avoid jumps

Flow-of-Control: Examples

Production	Semantic Rules
S -> if E then S1	E.true := newlabel; E.false := S.next; S1.next := S.next; S.code := E.code || gen(E.true ':') || S1.code
S -> if E then S1 else S2	E.true := newlabel; E.false := newlabel; S1.next := S.next; S2.next := S.next; S.code := E.code || gen(E.true ':') || S1.code || gen('goto' S.next) || gen(E.false ':') || S2.code
S -> while E do S1	S.begin := newlabel; E.true := newlabel; E.false := S.next; S1.next := S.begin; S.code := gen(S.begin ':') || E.code || gen(E.true ':') || S1.code || gen('goto' S.begin)

Labels and Goto Statements

- The definition of a label is treated as a declaration of the label
- Labels are typically entered into the symbol table
  - Entry is created the first time the label is seen
  - This may be before the definition of the label if it is the target of any forward goto
- When a compiler encounters a goto statement:
  - It must ensure that there is exactly one appropriate label in the current scope
  - If so, it must generate the appropriate code; otherwise, an error should be indicated

Return Statements

- Several actions must also take place when a procedure terminates
  - If the called procedure is a function, the result must be stored in a known place
  - The activation record of the calling procedure must be restored
  - A jump to the calling procedure's return address must be generated
- No exact division of run-time tasks between the calling and called procedure

Pass by Reference

- The param statements can be used as placeholders for arguments
- The called procedure is passed a pointer to the first of the param statements
- Any argument can by obtained by using the proper offset from the base pointer
- Arguments other than simple names:
  - First generate three-address statements needed to evaluate these arguments
  - Follow this by a list of param three-address statements

Pass by Reference Using a Queue

Production	Semantic Rules
S -> call id (Elist)	for each item p on queue do emit('param' p); emit('call' id.place)
Elist -> Elist, E	push E.place to queue
Elist -> E	initialize queue to contain E

- The code to evaluate arguments is emitted first, followed by param statements and then a call
- If desired, could augment rules to count the number of parameters

Backpatching

- A key problem when generating code for Boolean expressions and flow-of-control statements is that of matching a jump instruction with the target of the jump.
- Backpatching uses lists of jumps which are passed as synthesized attributes.
- Specifically, when a jump is generated, the target of the jump is temporarily left unspecified. Each such jump is put on a list of jumps whose labels are to be filled in when the proper label can be determined.

One-Pass Code Generation using Backpatching

- Generate instructions into an instruction array, and labels will be indices into this array. To manipulate lists of jumps, three functions are used:
  - makelist(i) creates a new list containing only i, an index into the array of instructions; makelist returns a pointer to the newly created list.
  - merge(pl , p2) concatenates the lists pointed to by pl and p2 , and returns a pointer to the concatenated list.
  - backpatch(p, i) inserts i as the target label for each of the instructions on the list pointed to by p.

SYNOPSYS®

Predictable Success