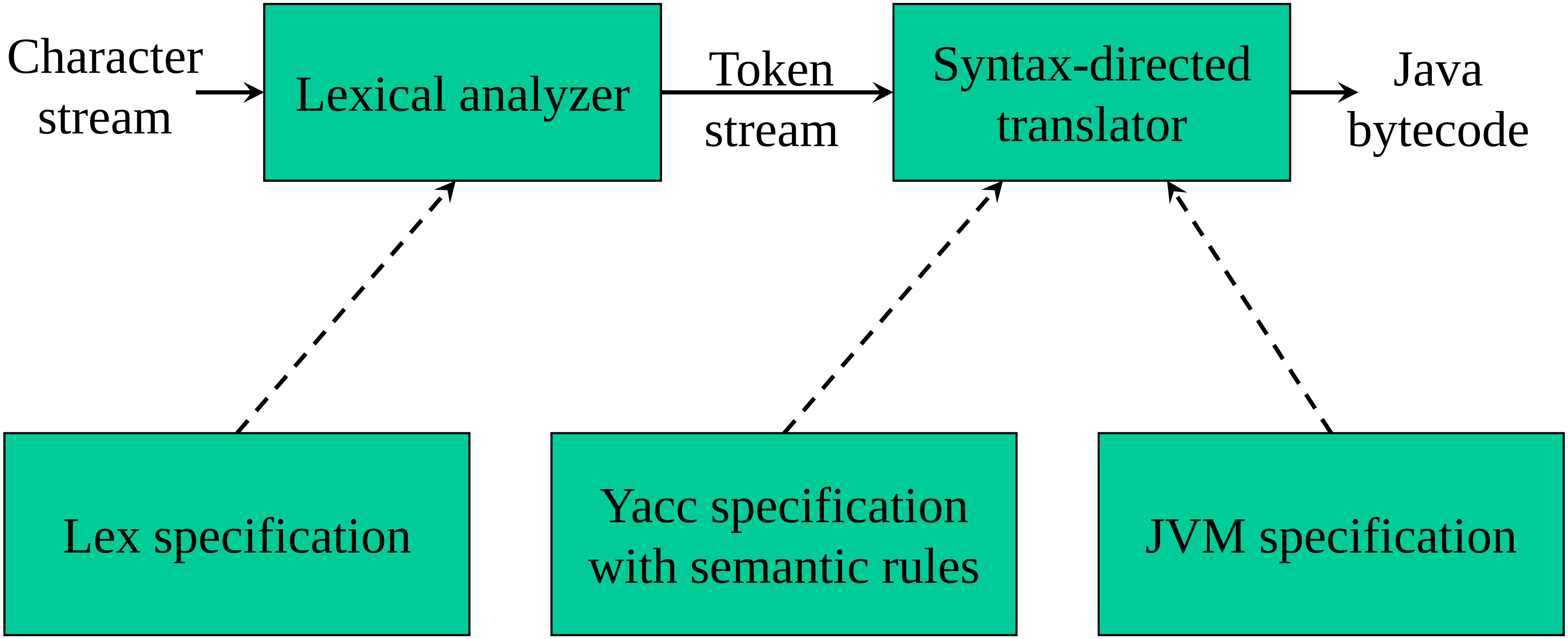
Syntax-Directed Translation
&
Intermediate Code Generation

The Structure of our Compiler Revisited



Character stream

Lexical analyzer

Token stream

Syntax-directed translator

Java bytecode

Lex specification

Yacc specification with semantic rules

JVM specification

Syntax-Directed Translation

- Grammar symbols are associated with **attributes** to associate information with the programming language constructs that they represent.
- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.
- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may put information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities
  - in fact, they may perform almost any activities.
- An attribute may hold almost any thing.
  - a string, a number, a memory location, a complex record.

Syntax-Directed Definitions and Translation Schemes

- When we associate semantic rules with productions, we use two notations:
  - **Syntax-Directed Definitions**
  - **Translation Schemes**
- **Syntax-Directed Definitions:**
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.
- **Translation Schemes:**
  - indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

Example Attribute Grammar in Yacc

```
%token DIGIT
%%
L : E ‘\n’          { printf(“%d\n”, $1); }
  ;
E : E ‘+’ T         { $$ = $1 + $3; }
  | T               { $$ = $1; }
  ;
T : T ‘*’ F         { $$ = $1 * $3; }
  | F               { $$ = $1; }
  ;
F : ‘(’ E ‘)’       { $$ = $2; }
  | DIGIT           { $$ = $1; }
  ;
%%
```

Syntax-Directed Definitions

- A syntax-directed definition is a generalization of a context-free grammar in which:
  - Each grammar symbol is associated with a set of attributes.
  - This set of attributes for a grammar symbol is partitioned into two subsets called **synthesized** and **inherited** attributes of that grammar symbol.
  - Each production rule is associated with a set of semantic rules.
- *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.
- This *dependency graph* determines the evaluation order of these semantic rules.
- Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) of the parse tree.
- Of course, the order of these computations depends on the dependency graph induced by the semantic rules.

Syntax-Directed Definition

- In a syntax-directed definition, each production $A \rightarrow \alpha$ is associated with a set of semantic rules of the form:

  $b=f(c_1,c_2,\ldots,c_n)$ where f is a function,

  and b can be one of the followings:

  ➔ b is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

  OR

  ➔ b is an inherited attribute one of the grammar symbols in α (on the right side of the production), and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production ($A \rightarrow \alpha$).

Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends* o*n* attributes $c_1,c_2,\ldots,c_n$.
- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

- An **attribute grammar** is a syntax-directed definition in which the functions in the semantic rules cannot have side effects (they can only evaluate values of attributes).

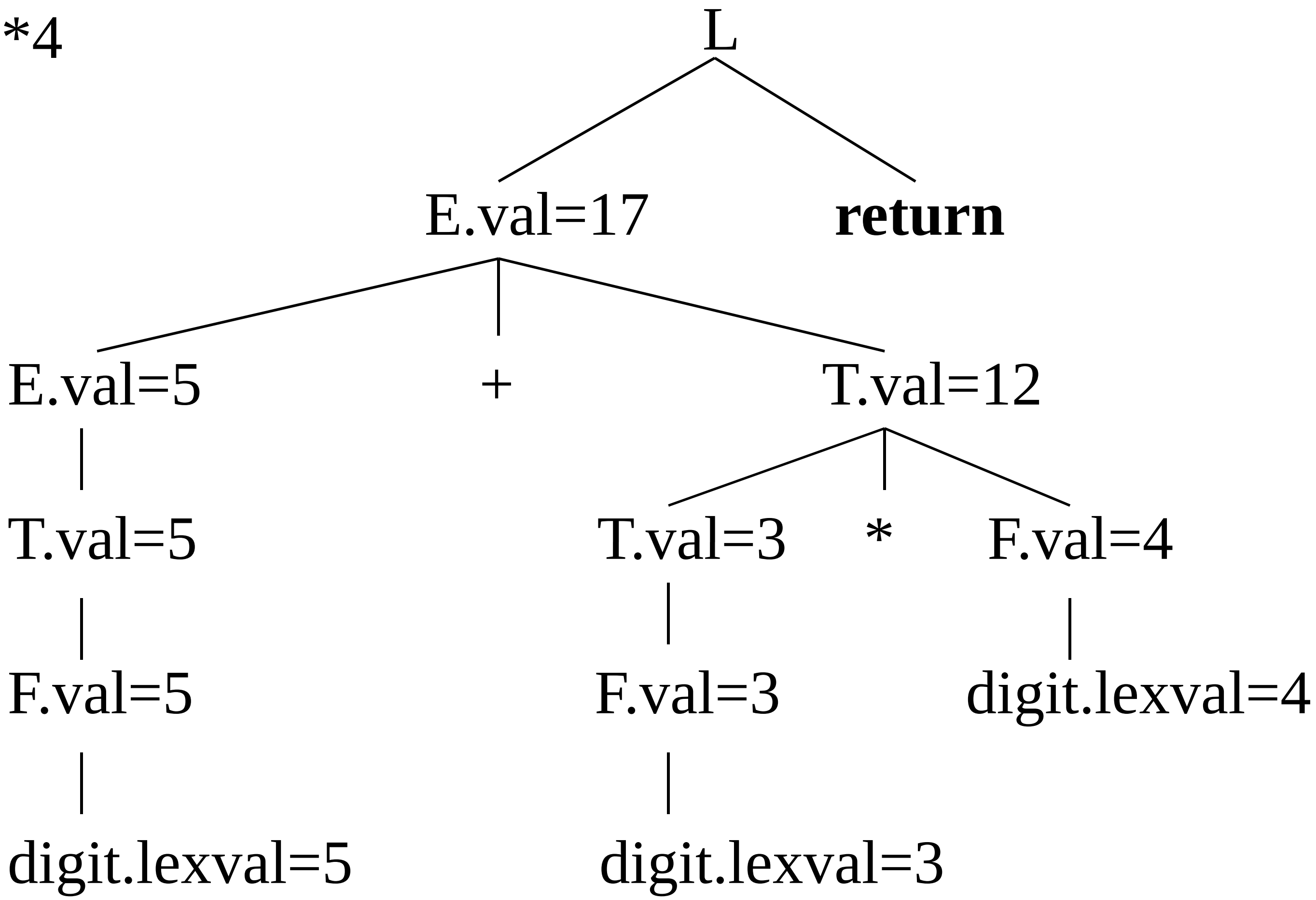
Syntax-Directed Definition -- Example

Production	**Semantic Rules**
L → E **return**	print(E.val)
$E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
E → T	E.val = T.val
$T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = **digit**.lexval

- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

Annotated Parse Tree -- Example

Input: 5+3*4



L

E.val=17 **return**

E.val=5 + T.val=12

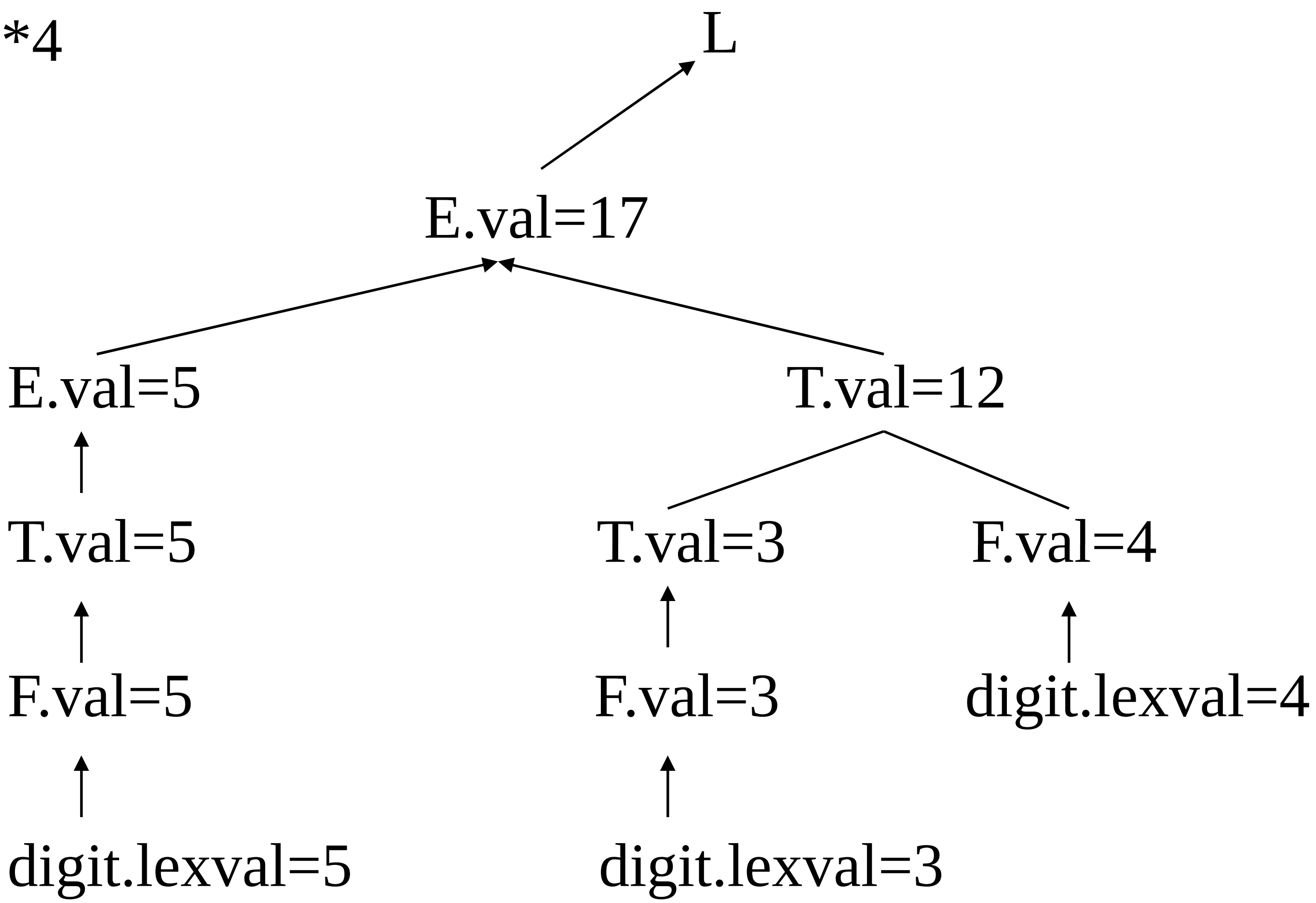
T.val=5 T.val=3 * F.val=4

F.val=5 F.val=3 digit.lexval=4

digit.lexval=5 digit.lexval=3

Dependency Graph

Input: 5+3*4



L

E.val=17

E.val=5

T.val=12

T.val=5

T.val=3

F.val=4

F.val=5

F.val=3

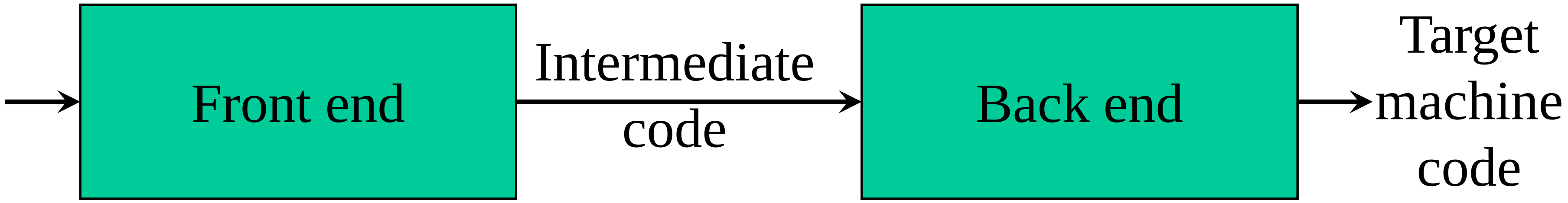
digit.lexval=4

digit.lexval=5

digit.lexval=3

Intermediate Code Generation

- Facilitates *retargeting*: enables attaching a back end for the new machine to an existing front end


Front end
Intermediate code
Back end
Target machine code

- Enables machine-independent code optimization

Intermediate Representations

- *Graphical representations* (e.g. AST)
- *Postfix notation*: operations on values stored on operand stack (similar to JVM bytecode)
- *Three-address code*: (e.g. *triples* and *quads*)

  $x := y \text{ op } z$
- *Two-address code*:

  $x := \text{op } y$

  which is the same as $x := x \text{ op } y$

Implementing Syntax Trees

- Each node can be represented by a record with several fields
- Example: node representing an operator used in an expression:
  - One field indicates the operator and others point to records for nodes representing operands
  - The operator is referred to as the “label” of the node
- If being used for translation, records can have additional fields for attributes

Syntax Trees for Expressions

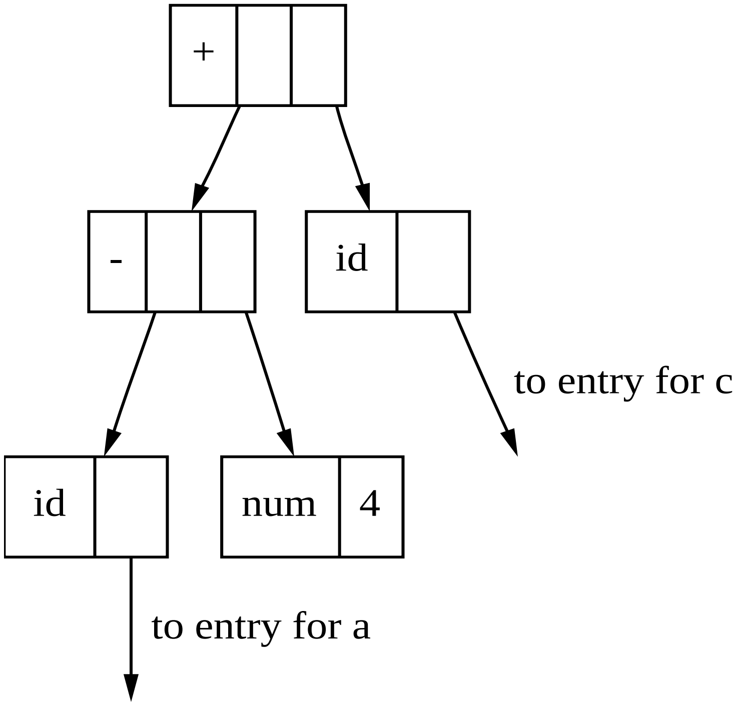
- Functions will create nodes for the syntax tree
  - `mknode (op, left, right)` – creates an operator node with label `op` and pointers `left` and `right` which point to operand nodes
  - `mkleaf(id, entry)` – creates an identifier node with label `id` and a pointer to the appropriate symbol table entry
  - `Mkleaf(num, val)` – creates a number node with label `num` and value `val`
- Each function returns pointer to created node

Syntax-Directed Translation of Abstract Syntax Trees

Production	Semantic Rule
$S \rightarrow$ **id :=** E	S.nptr := *mknode*(':=', *mkleaf*(**id**, **id**.entry), E.nptr)
$E \rightarrow E_1$ **+** E_2	E.nptr := *mknode*('+', E_1.nptr, E_2.nptr)
$E \rightarrow E_1$ ***** E_2	E.nptr := *mknode*('*', E_1.nptr, E_2.nptr)
$E \rightarrow$ **-** E_1	E.nptr := *mknode*('uminus', E_1.nptr)
$E \rightarrow$ **(** E_1 **)**	E.nptr := E_1.nptr
$E \rightarrow$ **id**	E.nptr := *mkleaf*(**id**, **id**.entry)

Example: `a - 4 + c`

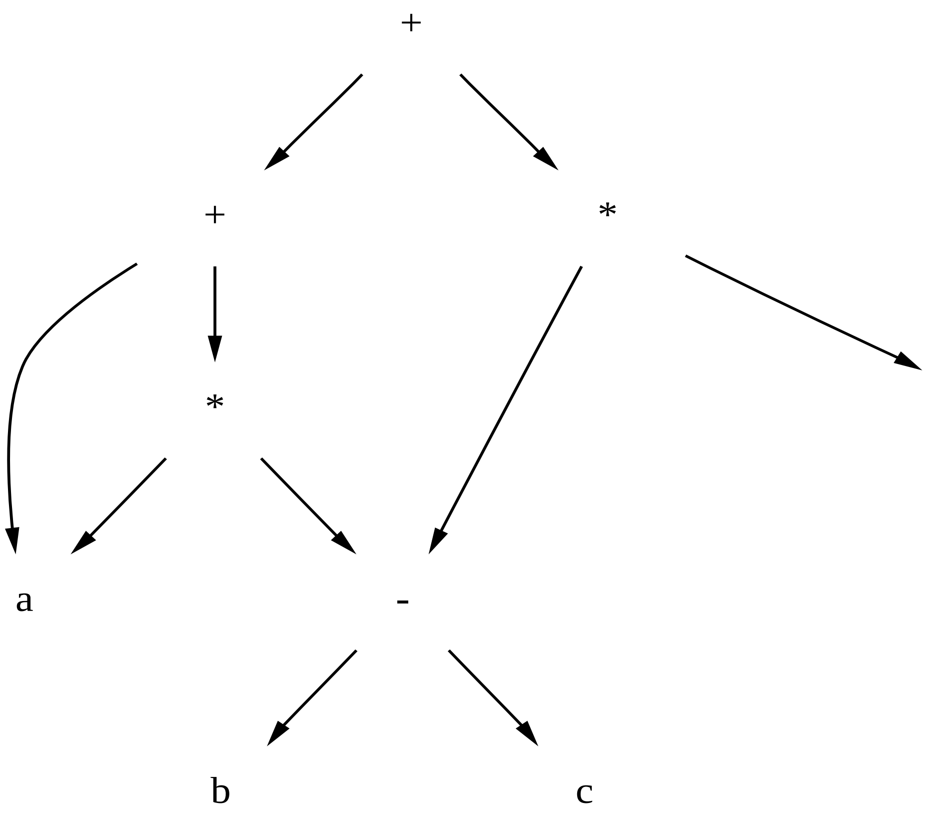
```
$p_1$ := mkleaf(**id**, $p_a$);
$P_2$ := mkleaf(num, 4);
$p_3$ := mknode('-', $p_1$, $p_2$);
$p_4$ := mkleaf(**id**, $p_c$);
$p_5$ := mknode('+', $p_3$, $p_4$);
```



Directed Acyclic Graphs

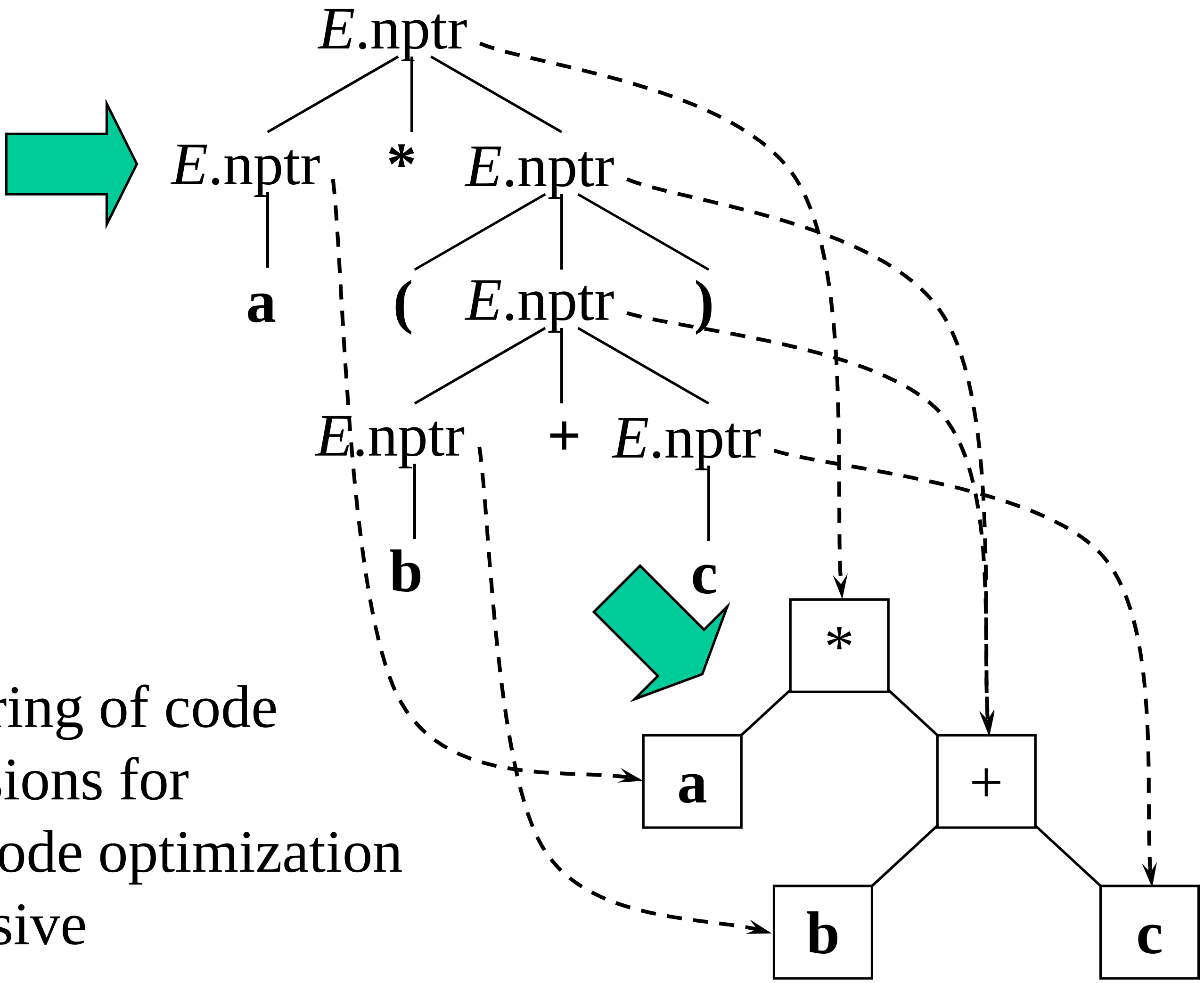
- Called a *dag* for short
- Convenient for representing expressions
- As with syntax trees:
  - Every subexpression will be represented by a node
  - Interior nodes represent operators, children represent operands
- Unlike syntax trees, nodes may have more than one parent
- Can be created automatically (discussed in textbook)

Example: a + a * (b – c) + (b – c) * d



Abstract Syntax Trees

a * (b + c)

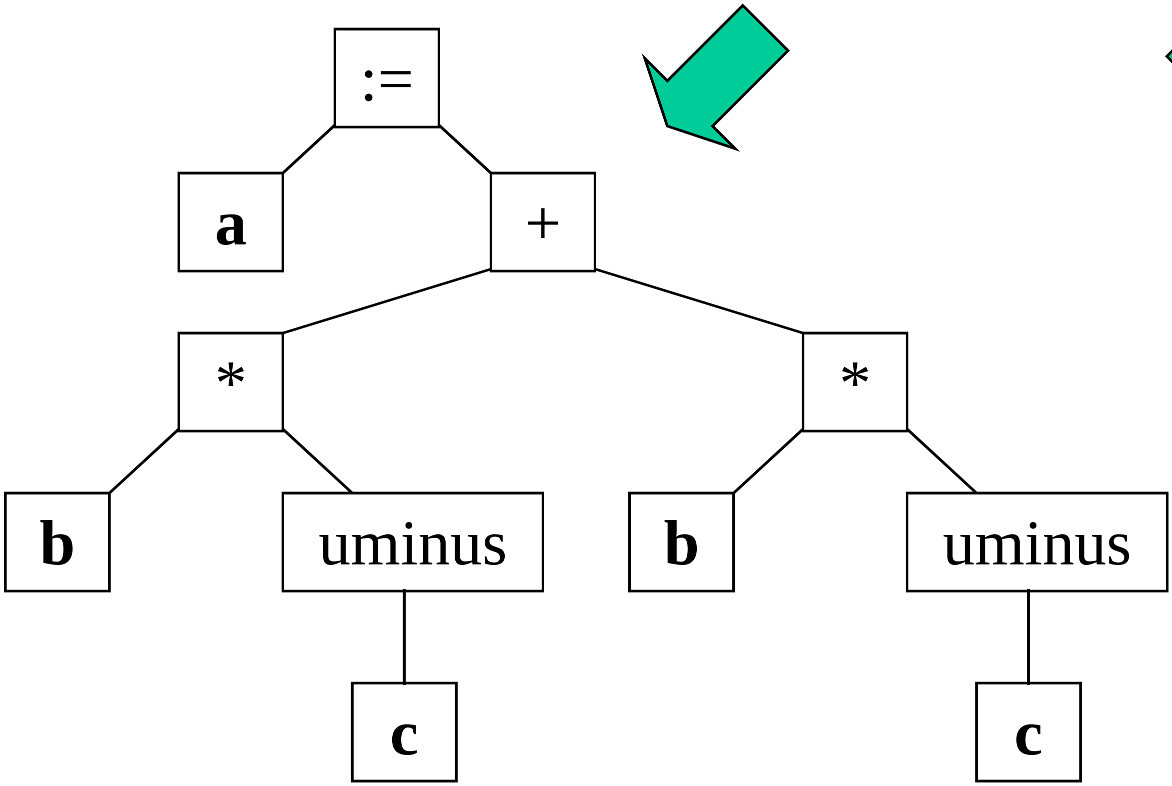


Pro: easy restructuring of code and/or expressions for intermediate code optimization

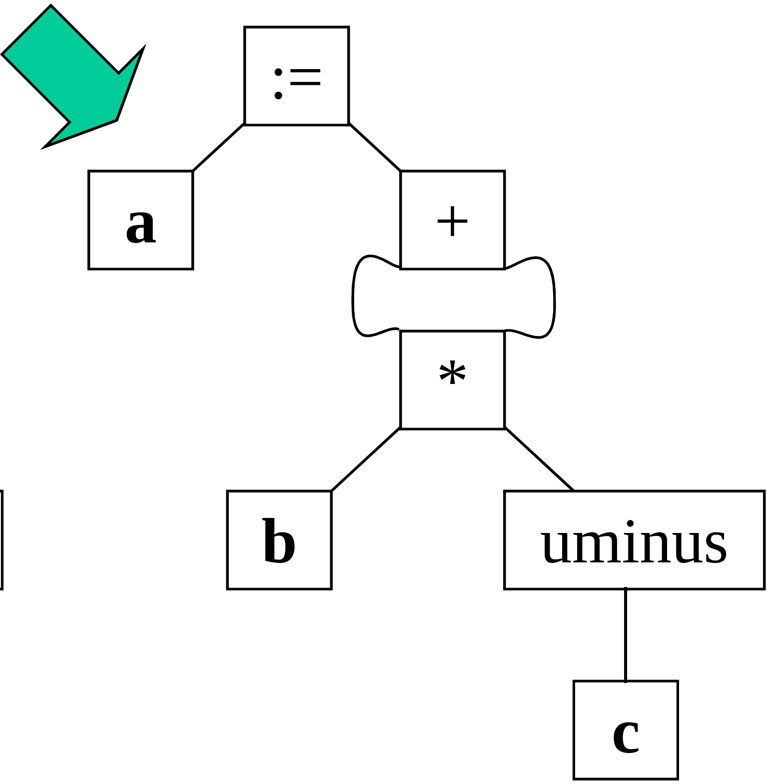
Cons: memory intensive

Abstract Syntax Trees versus DAGs

a := b * -c + b * -c



Tree



DAG

Postfix Notation

a := b * -c + b * -c

a b c uminus * b c uminus * + assign

Postfix notation represents operations on a stack

Pro: easy to generate

Cons: stack operations are more difficult to optimize

Bytecode (for example)

```
iload 2       // push b
iload 3       // push c
ineg          // uminus
imul          // *
iload 2       // push b
iload 3       // push c
ineg          // uminus
imul          // *
iadd          // +
istore 1      // store a
```

Three-Address Code

a := b * -c + b * -c

```
t1 := - c
t2 := b * t1
t3 := - c
t4 := b * t3
t5 := t2 + t4
a  := t5
```

Linearized representation of a syntax tree

```
t1 := - c
t2 := b * t1
t5 := t2 + t2
a  := t5
```

Linearized representation of a syntax DAG

Three-Address Statements

- Assignment statements: *x* `:=` *y op z*, *x* `:=` *op y*
- Indexed assignments: *x* `:=` *y*[*i*], *x*[*i*] `:=` *y*
- Pointer assignments: *x* `:=` `&`*y*, *x* `:=` `*`*y*, `*`*x* `:=` *y*
- Copy statements: *x* `:=` *y*
- Unconditional jumps: **`goto`** *lab*
- Conditional jumps: **`if`** *x relop y* **`goto`** *lab*
- Function calls: **`param`** *x…* **`call`** *p, n*
  `return` *y*

Syntax-Directed Translation into Three-Address Code

Productions

$S \rightarrow \textbf{id} := E$

$\quad \mid \textbf{while}\ E\ \textbf{do}\ S$

$E \rightarrow E + E$

$\quad \mid E\ \textbf{*}\ E$

$\quad \mid \textbf{-}\ E$

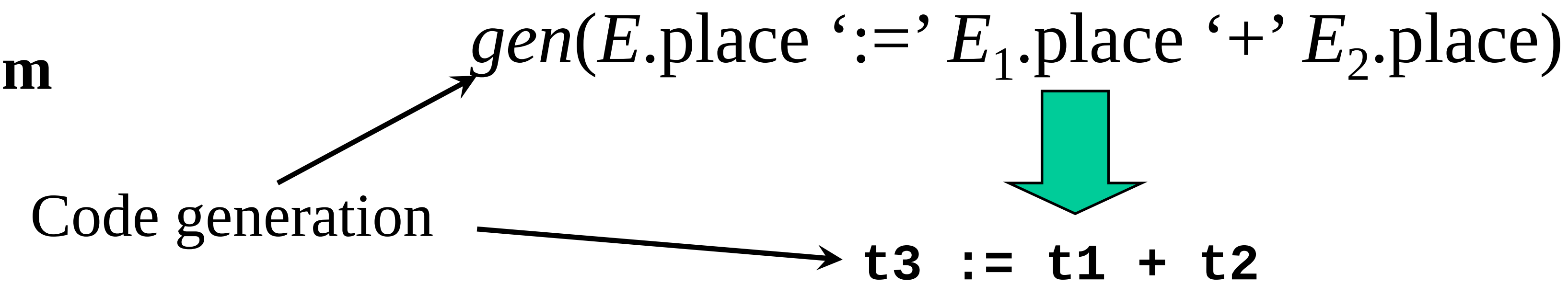
$\quad \mid \textbf{(}\ E\ \textbf{)}$

$\quad \mid \textbf{id}$

$\quad \mid \textbf{num}$

Synthesized attributes:

S.code	three-address code for *S*
S.begin	label to start of *S* or nil
S.after	label to end of *S* or nil
E.code	three-address code for *E*
E.place	a name holding the value of *E*



$gen(E.\text{place}\ ‘{:=}’\ E_1.\text{place}\ ‘{+}’\ E_2.\text{place})$

Code generation

`t3 := t1 + t2`

Syntax-Directed Translation into Three-Address Code (cont'd)

Productions	Semantic rules
$S \rightarrow$ **id** **:=** E	*S*.code := *E*.code || *gen*(**id**.place ‘:=’ *E*.place); *S*.begin := *S*.after := nil
$E \rightarrow E_1$ **+** E_2	*E*.place := *newtemp*(); *E*.code := E_1.code || E_2.code || *gen*(*E*.place ‘:=’ E_1.place ‘+’ E_2.place)
$E \rightarrow E_1$ ***** E_2	*E*.place := *newtemp*(); *E*.code := E_1.code || E_2.code || *gen*(*E*.place ‘:=’ E_1.place ‘*’ E_2.place)
$E \rightarrow$ **-** E_1	*E*.place := *newtemp*(); *E*.code := E_1.code || *gen*(*E*.place ‘:=’ ‘uminus’ E_1.place)
$E \rightarrow$ **(** E_1 **)**	*E*.place := E_1.place *E*.code := E_1.code
$E \rightarrow$ **id**	*E*.place := **id**.name *E*.code := ‘’
$E \rightarrow$ **num**	*E*.place := *newtemp*(); *E*.code := *gen*(*E*.place ‘:=’ **num**.value)

Syntax-Directed Translation into Three-Address Code (cont'd)

Production

$S \rightarrow$ **while** E **do** S_1

Semantic rule

S.begin := *newlabel*()
S.after := *newlabel*()
S.code := *gen*(*S*.begin ':') ||
E.code ||
gen('if' *E*.place '=' '0' 'goto' *S*.after) ||
S_1.code ||
gen('goto' *S*.begin) ||
gen(*S*.after ':')

S.begin:	*E*.code
	`if` *E*.place `= 0 goto` *S*.after
	S.code
	`goto` *S*.begin
S.after:	…

Example

```
i := 2 * n + k
while i do
   i := i - k
```

```
    t1 := 2
    t2 := t1 * n
    t3 := t2 + k
    i  := t3
L1: if i = 0 goto L2
    t4 := i - k
    i  := t4
    goto L1
L2:
```

Implementation of Three-Address Statements: Quads

#	*Op*	*Arg1*	*Arg2*	*Res*
(0)	uminus	c		t1
(1)	*	b	t1	t2
(2)	uminus	c		t3
(3)	*	b	t3	t4
(4)	+	t2	t4	t5
(5)	:=	t5		a

Quads (quadruples)

Pro: easy to rearrange code for global optimization

Cons: lots of temporaries

Implementation of Three-Address Statements: Triples

#	*Op*	*Arg1*	*Arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	:=	a	(4)

Triples

Pro: temporaries are implicit

Cons: difficult to rearrange code

Implementation of Three-Address Stmts: Indirect Triples

#	*Stmt*
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

Program

#	*Op*	*Arg1*	*Arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	:=	a	(18)

Triple container

Pro: temporaries are implicit & easier to rearrange code

Names and Scopes

- The three-address code generated by the syntax-directed definitions shown on the previous slides is somewhat simplistic, because it assumes that the names of variables can be easily resolved by the back end in global or local variables
- We need local symbol tables to record global declarations as well as local declarations in procedures, blocks, and structs to resolve names

Symbol Tables for Scoping

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void somefunc()
{ …
  swap(s.a, s.b);
  …
}
```

We need a symbol table for the *fields* of struct S

Need symbol table for *global* variables and functions

Need symbol table for *arguments* and *locals* for each function

Check: **`s`** is global and has fields **`a`** and **`b`**

Using symbol tables we can generate code to access **`s`** and its fields

Offset and Width for Runtime Allocation

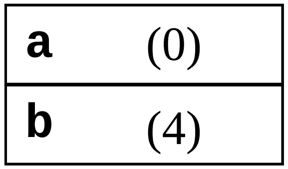
```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void somefunc()
{ …
  swap(s.a, s.b);
  …
}
```

The fields **a** and **b** of struct S are located at *offsets* 0 and 4 from the start of S

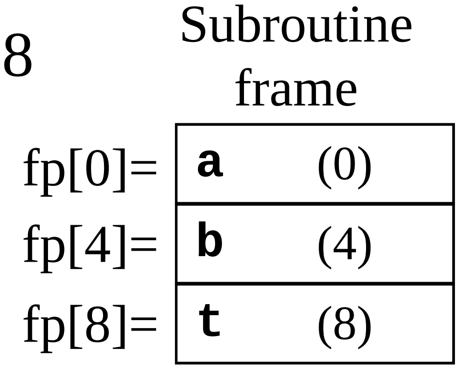
The *width* of S is 8



a	(0)
b	(4)

Subroutine frame holds arguments **a** and **b** and local **t** at *offsets* 0, 4, and 8

The *width* of the frame is 12



Subroutine frame

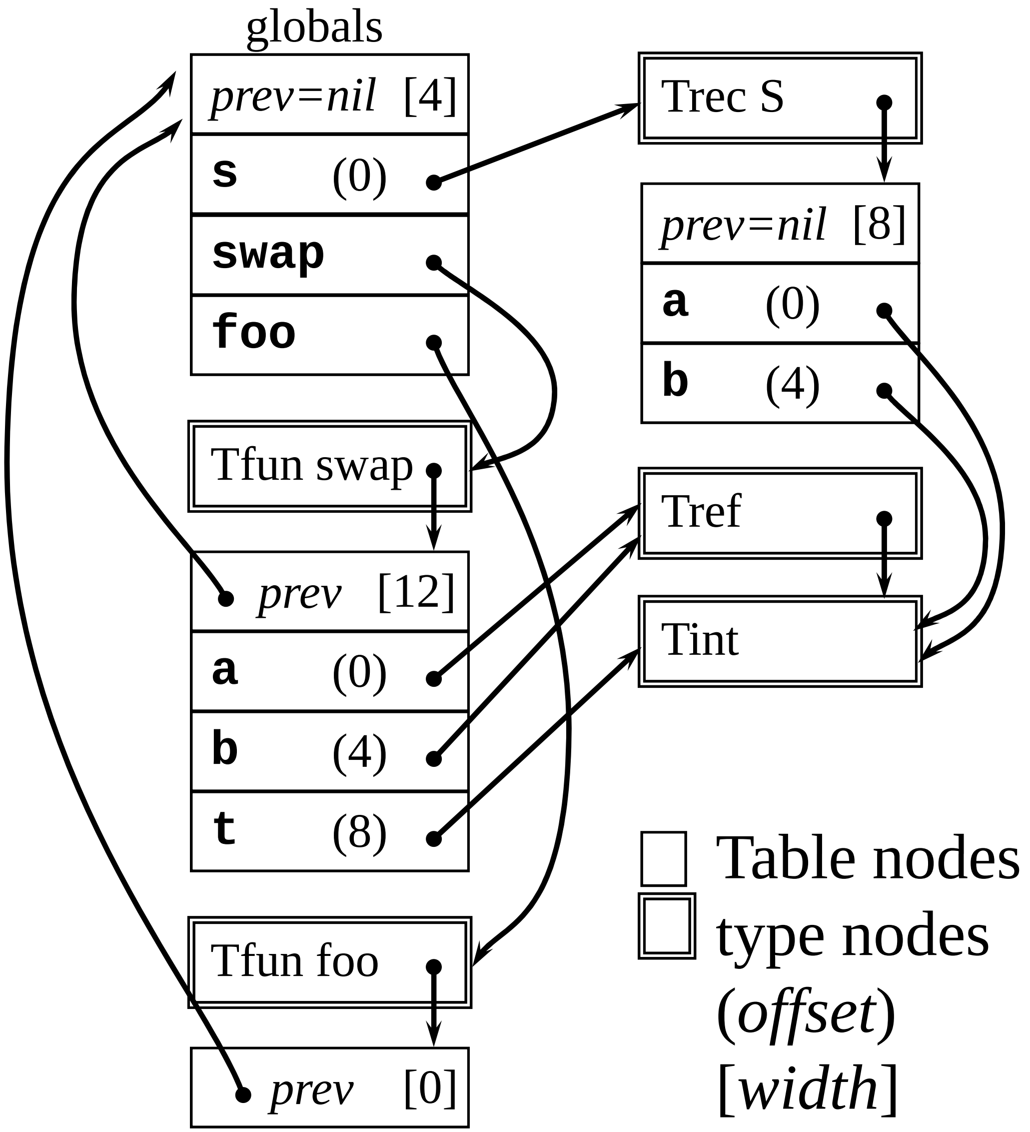
fp[0]=	**a**	(0)
fp[4]=	**b**	(4)
fp[8]=	**t**	(8)

Example

```
struct S
{ int a;
  int b;
} s;

void swap(int& a, int& b)
{ int t;
  t = a;
  a = b;
  b = t;
}

void foo()
{ …
  swap(s.a, s.b);
  …
}
```



Hierarchical Symbol Table Operations

- *mktable*(*previous*) returns a pointer to a new table that is linked to a previous table in the outer scope
- *enter*(*table*, *name*, *type*, *offset*) creates a new entry in *table*
- *addwidth*(*table*, *width*) accumulates the total width of all entries in *table*
- *enterproc*(*table*, *name*, *newtable*) creates a new entry in *table* for procedure with local scope *newtable*
- *lookup*(*table*, *name*) returns a pointer to the entry in the table for *name* by following linked tables

Syntax-Directed Translation of Declarations in Scope

Productions

$P \rightarrow D \;\textbf{;}\; S$

$D \rightarrow D \;\textbf{;}\; D$

$\quad \mid \textbf{id :}\; T$

$\quad \mid \textbf{proc id ;}\; D \;\textbf{;}\; S$

$T \rightarrow \textbf{integer}$

$\quad \mid \textbf{real}$

$\quad \mid \textbf{array [num] of}\; T$

$\quad \mid \wedge\, T$

$\quad \mid \textbf{record}\; D \;\textbf{end}$

$S \rightarrow S \;\textbf{;}\; S$

$\quad \mid \textbf{id :=}\; E$

$\quad \mid \textbf{call id (}\; A \;\textbf{)}$

Productions *(cont'd)*

$E \rightarrow E \;\textbf{+}\; E$

$\quad \mid E \;\textbf{*}\; E$

$\quad \mid \textbf{-}\; E$

$\quad \mid \textbf{(}\; E \;\textbf{)}$

$\quad \mid \textbf{id}$

$\quad \mid E\, \wedge$

$\quad \mid \textbf{\&}\; E$

$\quad \mid E \;\textbf{. id}$

$A \rightarrow A \;\textbf{,}\; E$

$\quad \mid E$

Synthesized attributes:

T.type	pointer to type
T.width	storage width of type (bytes)
E.place	name of temp holding value of *E*

Global data to implement scoping:

tblptr	stack of pointers to tables
offset	stack of offset values

Syntax-Directed Translation of Declarations in Scope (cont'd)

$P \rightarrow \quad \{\ t := mktable(\text{nil});\ push(t, tblptr);\ push(0, offset)\ \}$

$\quad D\ \mathbf{;}\ S$

$D \rightarrow \mathbf{id} : T$

$\quad \{\ enter(top(tblptr), \mathbf{id}.\text{name}, T.\text{type}, top(offset));$

$\quad\ top(offset) := top(offset) + T.\text{width}\ \}$

$D \rightarrow \mathbf{proc\ id\ ;}$

$\quad \{\ t := mktable(top(tblptr));\ push(t, tblptr);\ push(0, offset)\ \}$

$\quad D_1\ \mathbf{;}\ S$

$\quad \{\ t := top(tblptr);\ addwidth(t, top(offset));$

$\quad\ pop(tblptr);\ pop(offset);$

$\quad\ enterproc(top(tblptr), \mathbf{id}.\text{name}, t)\ \}$

$D \rightarrow D_1\ \mathbf{;}\ D_2$

Syntax-Directed Translation of Declarations in Scope (cont'd)

$T \rightarrow$ **integer** { 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 * T_1.width }

$T \rightarrow$ ^ T_1

{ T.type := *pointer*(T_1.type); T.width := 4 }

$T \rightarrow$ **record**

{ *t* := *mktable*(nil); *push*(*t*, *tblptr*); *push*(0, *offset*) }

D **end**

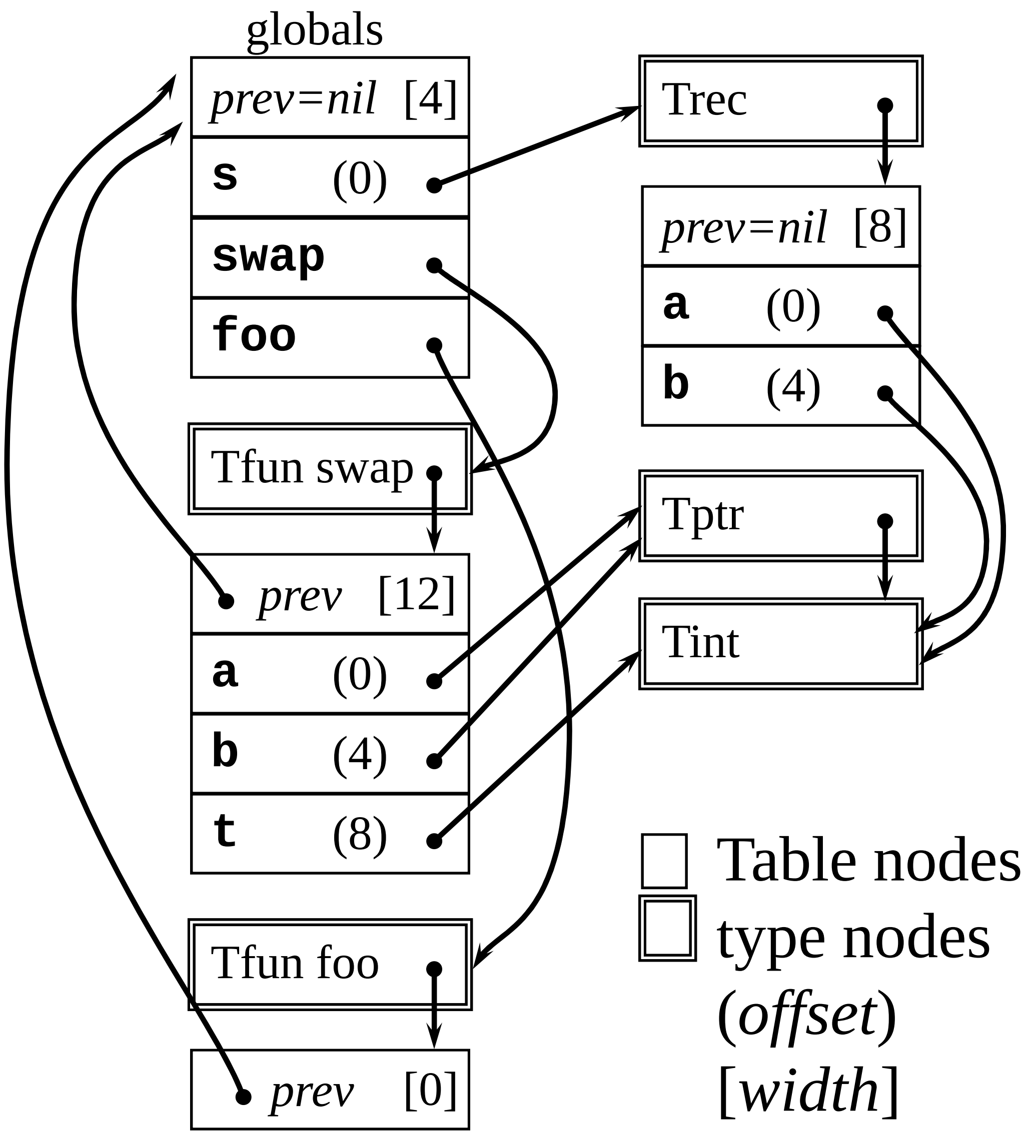
{ T.type := *record*(*top*(*tblptr*)); T.width := *top*(*offset*);
addwidth(*top*(*tblptr*), *top*(*offset*)); *pop*(*tblptr*); *pop*(*offset*) }

Example

```
s: record
      a: integer;
      b: integer;
   end;

proc swap;
  a: ^integer;
  b: ^integer;
  t: integer;
  t := a^;
  a^ := b^;
  b^ := t;

proc foo;
  call swap(&s.a, &s.b);
```



Syntax-Directed Translation of Statements in Scope

$S \rightarrow S \,;\, S$

$S \rightarrow \textbf{id} := E$

```
{ p := lookup(top(tblptr), id.name);
  if p = nil then
    error()
  else if p.level = 0 then // global variable
    emit(id.place ‘:=’ E.place)
  else // local variable in subroutine frame
    emit(fp[p.offset] ‘:=’ E.place) }
```

Globals

s	(0)
x	(8)
y	(12)

Subroutine frame

fp[0]=	**a**	(0)
fp[4]=	**b**	(4)
fp[8]=	**t**	(8)
…		

Syntax-Directed Translation of Expressions in Scope

```
$E \to E_1$ + $E_2$    { $E$.place := newtemp();
                          emit($E$.place ':=' $E_1$.place '+' $E_2$.place) }

$E \to E_1$ * $E_2$    { $E$.place := newtemp();
                          emit($E$.place ':=' $E_1$.place '*' $E_2$.place) }

$E \to$ - $E_1$        { $E$.place := newtemp();
                          emit($E$.place ':=' 'uminus' $E_1$.place) }

$E \to$ ( $E_1$ )      { $E$.place := $E_1$.place }

$E \to$ id { $p$ := lookup(top(tblptr), id.name);
             if $p$ = nil then error()
             else if $p$.level = 0 then // global variable
                 $E$.place := id.place
             else // local variable in frame
                 $E$.place := fp[$p$.offset] }
```

Syntax-Directed Translation of Expressions in Scope (cont'd)

```
$E \to E_1$ ^          { E.place := newtemp();
                         emit(E.place ':=' '*' $E_1$.place) }

$E \to$ & $E_1$        { E.place := newtemp();
                         emit(E.place ':=' '&' $E_1$.place) }

$E \to$ id$_1$ . id$_2$    { p := lookup(top(tblptr), id$_1$.name);
                         if p = nil or p.type != Trec then error()
                         else
                           q := lookup(p.type.table, id$_2$.name);
                           if q = nil then error()
                           else if p.level = 0 then // global variable
                             E.place := id$_1$.place[q.offset]
                           else // local variable in frame
                             E.place := fp[p.offset+q.offset] }
```

Advanced Intermediate Code Generation Techniques

- Reusing temporary names
- Addressing array elements
- Translating logical and relational expressions
- Translating short-circuit Boolean expressions and flow-of-control statements with backpatching lists
- Translating procedure calls

Reusing Temporary Names

$E_1 + E_2$ $\xrightarrow{\text{generate}}$

Evaluate E_1 into **t1**
Evaluate E_2 into **t2**
t3 := t1 + t2

↑

If **t1** no longer used, can reuse **t1** instead of using new temp **t3**

Modify *newtemp*() to use a "stack":
Keep a counter *c*, initialized to 0
newtemp() increments *c* and returns temporary **$***c*
Decrement counter on each use of a **$***i* in a three-address statement

Reusing Temporary Names (cont'd)

x := a * b + c * d - e * f

Statement	*c*
	0
`$0 := a * b`	1
`$1 := c * d`	2
`$0 := $0 + $1`	1
`$1 := e * f`	2
`$0 := $0 - $1`	1
`x  := $0`	0

Addressing Array Elements: One-Dimensional Arrays

`A : array [10..20] of integer;`

`… := A[i]` $= base_A + (i - low) * w$

$= i * w + c$

where $c = base_A - low * w$

with $low = 10; w = 4$

⇓

`t1 := c` `// c =` $base_A - 10 * 4$

`t2 := i * 4`

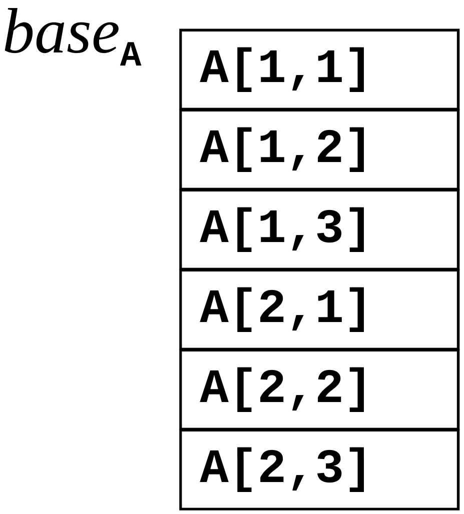
`t3 := t1[t2]`

`…  := t3`

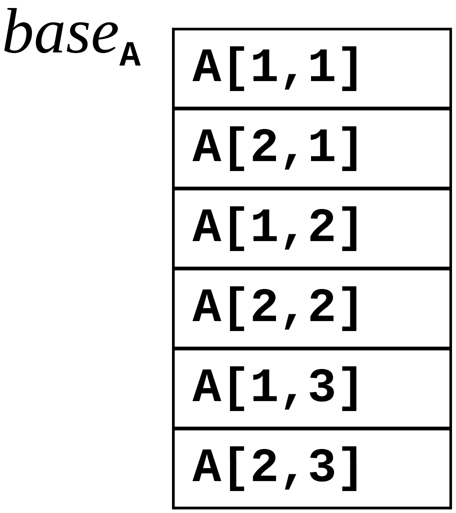
Addressing Array Elements: Multi-Dimensional Arrays

```
A : array [1..2,1..3] of integer;
```

$low_1 = 1, low_2 = 1, n_1 = 2, n_2 = 3, w = 4$



Row-major



Column-major

Addressing Array Elements: Multi-Dimensional Arrays

`A : array [1..2,1..3] of integer;` (Row-major)

`… := A[i,j]` $= base_{\mathtt{A}} + ((i_1 - low_1) * n_2 + i_2 - low_2) * w$

$= ((i_1 * n_2) + i_2) * w + c$

where $c = base_{\mathtt{A}} - ((low_1 * n_2) + low_2) * w$

with $low_1 = 1;\ low_2 = 1;\ n_2 = 3;\ w = 4$

⇩

```
t1 := i * 3
t1 := t1 + j
t2 := c       // c = baseA - (1 * 3 + 1) * 4
t3 := t1 * 4
t4 := t2[t3]
…  := t4
```

Addressing Array Elements: Grammar

$S \rightarrow L\ \mathbf{:=}\ E$

$E \rightarrow E\ \mathbf{+}\ E$

$\quad\mid \mathbf{(}\ E\ \mathbf{)}$

$\quad\mid L$

$L \rightarrow Elist\ \mathbf{]}$

$\quad\mid \mathbf{id}$

$Elist \rightarrow Elist\ \mathbf{,}\ E$

$\quad\quad\mid \mathbf{id}\ \mathbf{[}\ E$

Synthesized attributes:

E.place	name of temp holding value of *E*
Elist.array	array name
Elist.place	name of temp holding index value
Elist.ndim	number of array dimensions
L.place	lvalue (=name of temp)
L.offset	index into array (=name of temp) **null** indicates non-array simple **id**

Addressing Array Elements

```
$S \rightarrow L := E$           { if $L$.offset = null then
                                       emit($L$.place ‘:=’ $E$.place)
                                   else
                                       emit($L$.place[$L$.offset] ‘:=’ $E$.place) }
$E \rightarrow E_1 + E_2$        { $E$.place := newtemp();
                                   emit($E$.place ‘:=’ $E_1$.place ‘+’ $E_2$.place) }
$E \rightarrow ( E_1 )$          { $E$.place := $E_1$.place }
$E \rightarrow L$                { if $L$.offset = null then
                                       $E$.place := $L$.place
                                   else
                                       $E$.place := newtemp();
                                       emit($E$.place ‘:=’ $L$.place[$L$.offset] }
```

Addressing Array Elements

```
L → Elist ]        { L.place := newtemp();
                     L.offset := newtemp();
                     emit(L.place ‘:=’ c(Elist.array);
                     emit(L.offset ‘:=’ Elist.place ‘*’ width(Elist.array)) }
L → id             { L.place := id.place;
                     L.offset := null }
Elist → $Elist_1$ , E
                   { t := newtemp(); m := $Elist_1$.ndim + 1;
                     emit(t ‘:=’ $Elist_1$.place ‘*’ limit($Elist_1$.array, m));
                     emit(t ‘:=’ t ‘+’ E.place);
                     Elist.array := $Elist_1$.array; Elist.place := t;
                     Elist.ndim := m }
Elist → id [ E     { Elist.array := id.place; Elist.place := E.place;
                     Elist.ndim := 1 }
```

Translating Assignments

Production	Semantic Rules

S → id := E	p := lookup(id.name);
	if p != NULL then
	emit(p ':=' E.place)
	else
	error

E → E_1 + E_2	E.place := newtemp; emit(E.place ':=' E_1.place '+' E_2.place)
E → E_1 * E_2	E.place := newtemp; emit(E.place ':=' E_1.place '*' E_2.place)
E → -E_1	E.place := newtemp; emit(E.place ':=' 'uminus' E_1.place)
E → (E_1)	E.place := E_1.place

E → id	p := lookup(id.name);
	if p != NULL then
	E.place := p
	else
	error

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

Semantic Action: $E \rightarrow E_1 + E_2$

```
E.place := newtemp;
if E_1.type = integer and E_2.type = integer then
begin
  emit(E.place ':=' E_1.place 'int+' E_2.place);
  E.type := integer
end
else if E_1.type = real and E2.type = real then
  …
else if E_1.type = integer and E_2.type = real then
begin
  u := newtemp;
  emit(u ':=' 'inttoreal' E_1.place);
  emit(E.place ':=' u 'real+' E_2.place);
  E.type := real
end
else if E_1.type = real and E_2.type = integer then
  …
else E.type := type_error;
```

Example: x := y + i * j

- Without Type conversion

```
t1 := i * j
t2 := y + t1
x := t2
```

- With Type conversion
  - In this example, `x` and `y` have type `real`
  - `i` and `j` have type `integer`
  - The intermediate code is shown below:

```
t1 := i int* j
t3 := inttoreal t1
t2 := y real+ t3
x := t2
```

Boolean Expressions

- Boolean expressions compute logical values
- Often used with flow-of-control statements
- Methods of translating boolean expression:
  - Numerical methods:
    - 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 methods:
    - Represent the value of a boolean by the position reached in a program
    - Often not necessary to evaluate entire expression

Numerical Representation

- Expressions evaluated left to right using 1 to denote true and 0 to donate false
- Example: a or b and not c

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

- Another example: a < b

```
100:  if a < b goto 103
101:  t : = 0
102:  goto 104
103:  t : = 1
104:  …
```

Numerical Representation

Production	Semantic Rules
E → E_1 or E_2	E.place := newtemp; emit(E.place ':=' E_1.place 'or' E_2.place)
E → E_1 and E_2	E.place := newtemp; emit(E.place ':=' E_1.place 'and' E_2.place)
E → not E_1	E.place := newtemp; emit(E.place ':=' 'not' E_1.place)
E → (E_1)	E.place := E1.place;

E → id_1 relop id_2	E.place := newtemp;
	emit('if' id_1.place relop.op
	id_2.place 'goto' nextstat+3);
	emit(E.place ':=' '0');
	emit('goto' nextstat+2);
	emit(E.place ':=' '1');

E → true	E.place := newtemp; emit(E.place ':=' '1')
E → false	E.place := newtemp; emit(E.place ':=' '0')

Example: a<b or c<d and e<f

```
100:  if a < b goto 103
101:  t1 := 0
102:  goto 104
103:  t1 := 1
104:  if c < d goto 107
105:  t2 := 0
106:  goto 108
107:  t2 := 1
108:  if e < f goto 111
109:  t3 := 0
110:  goto 112
111:  t3:= 1
112:  t4 := t2 and t3
113:  t5 := t1 or t4
```

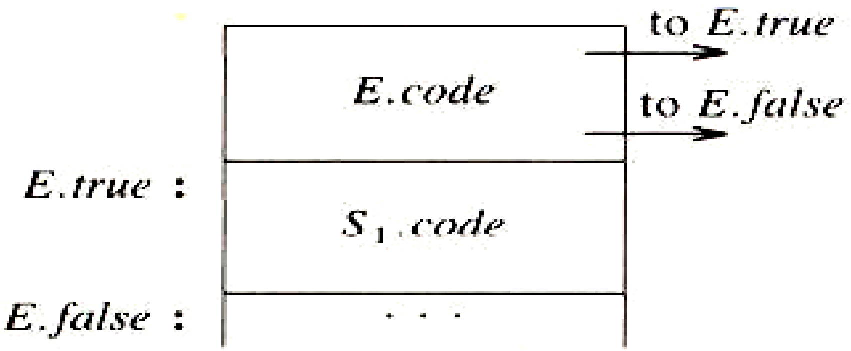
Flow-of-Control

Flow Control Statements

- S → if E then S1
- S → if E then S1 else S2
- S → while E do S1

- The function newlabel will return a new symbolic label each time it is called
- Each boolean expression will have 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 to jumps

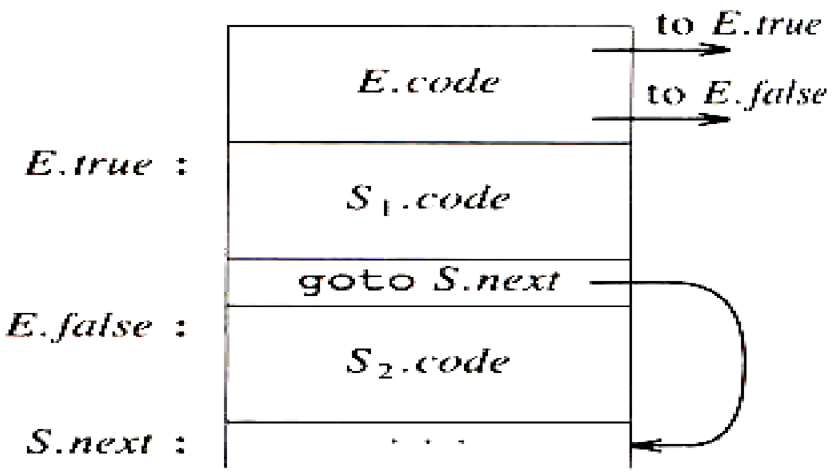
S → if E then S1



(a) if-then

```
E.true := newlabel;
E.false := S.next;
S1.next := S.next;
S.code := E.code ||  gen(E.true ':') ||  S1.code
```

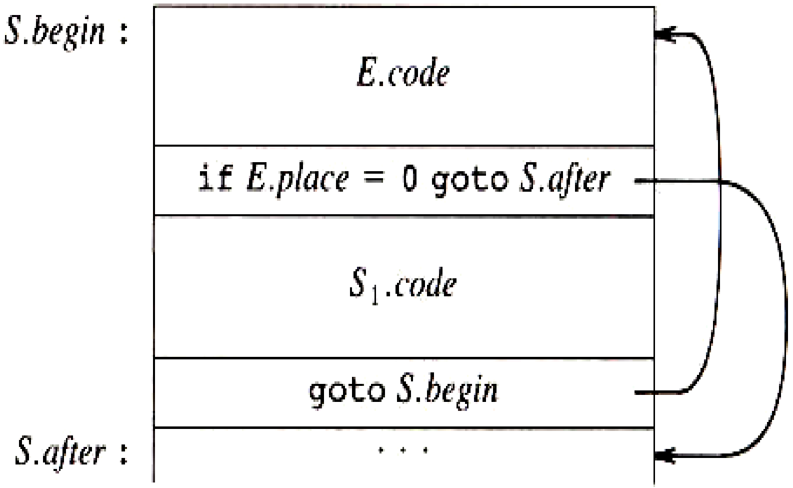
S → if E then S1 else S2



(b) if-then-else

```
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)
```

Boolean Expressions

Production	Semantic Rules
$E \rightarrow E_1$ or E_2	E_1.true := E.true; E_1.false := newlabel; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.false ':') || E_2.code
$E \rightarrow E_1$ and E_2	E_1.true := newlabel; E_1.false := E.false; E_2.true := E.true; E_2.false := E.false; E.code := E_1.code || gen(E_1.true ':') || E_2.code

Boolean Expressions

Production	Semantic Rules
E → not E_1	E_1.true := E.false; E_1.false := E.true; E.code := E1.code
E → (E_1)	E_1.true := E.true; E_1.false := E.false; E.code := E_1.code
E → id_1 relop id_2	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)

Examples:

```
a<b or c<d and e<f
```

```
        if a < b goto Ltrue
        goto L1
L1:     if c < d goto L2
        goto Lfalse
L2:     if e < f goto Ltrue
        goto Lfalse
```

```
while a < b do
  if c < d then
    x := y + z
  else
    x := y - z
```

```
L1:     if a < b goto L2
        goto Lnext
L2:     if c < d goto L3
        goto L4
L3:     t1 := y + z
        x:= t1
        goto L1
L4:     t2 := y – z
        X := t2
        goto L1
Lnext:
```

Mixed-Mode Expressions

- Boolean expressions often have arithmetic subexpressions, e.g. `(a + b) < c`
- If false has the value 0 and true has the value 1
  - arithmetic expressions can have boolean subexpressions
  - Example: `(a < b) + (b < a)` has value 0 if `a` and `b` are equal and 1 otherwise
- Some operators may require both operands to be boolean
- Other operators may take both types of arguments, including mixed arguments

Revisit: $E \rightarrow E_1 + E_2$

```
E.type := arith;
if E1.type = arith and E2.type = arith then
begin
  /* normal arithmetic add */
  E.place := newtemp;
  E.code := $E_1$.code || $E_2$.code ||
    gen(E.place ':=' $E_1$.place '+' $E_2$.place)
end
else if E1.type := arith and E2.type = bool then
begin
  E2.place := newtemp;
  E2.true := newlabel;
  E2.flase := newlabel;
  E.code := E1.code || E2.code ||
    gen(E2.true ':' E.place ':=' E1.place + 1) ||
    gen('goto' nextstat+1) ||
    gen(E2.false ':' E.place ':=' E1.place)
else if …
```

Translating Logical and Relational Expressions

a or b and not c →

```
t1 := not c
t2 := b and t1
t3 := a or t2
```

a < b →

```
    if a < b goto L1
    t1 := 0
    goto L2
L1: t1 := 1
L2:
```

Backpatching

$$
\begin{aligned}
E &\rightarrow E\ \textbf{or}\ M\ E \\
&\mid E\ \textbf{and}\ M\ E \\
&\mid \textbf{not}\ E \\
&\mid \textbf{(}\ E\ \textbf{)} \\
&\mid \textbf{id relop id} \\
&\mid \textbf{true} \\
&\mid \textbf{false} \\
M &\rightarrow \varepsilon
\end{aligned}
$$

Synthesized attributes:

E.code	three-address code
E.truelist	backpatch list for jumps on true
E.falselist	backpatch list for jumps on false
M.quad	location of current three-address quad

Backpatch Operations with Lists

- *makelist*(*i*) creates a new list containing three-address location *i*, returns a pointer to the list
- *merge*(p_1, p_2) concatenates lists pointed to by p_1 and p_2, returns a pointer to the concatenates list
- *backpatch*(*p*, *i*) inserts *i* as the target label for each of the statements in the list pointed to by *p*

Backpatching with Lists: Example

a < b or c < d and e < f

```
100: if a < b goto _
101: goto _
102: if c < d goto _
103: goto _
104: if e < f goto _
105: goto _
```

backpatch

```
100: if a < b goto TRUE
101: goto 102
102: if c < d goto 104
103: goto FALSE
104: if e < f goto TRUE
105: goto FALSE
```

Backpatching with Lists: Translation Scheme

$M \rightarrow \varepsilon$ { M.quad := $\textit{nextquad}$() }

$E \rightarrow E_1$ **or** $M\ E_2$

{ $\textit{backpatch}(E_1$.falselist, M.quad);
E.truelist := $\textit{merge}(E_1$.truelist, E_2.truelist);
E.falselist := E_2.falselist }

$E \rightarrow E_1$ **and** $M\ E_2$

{ $\textit{backpatch}(E_1$.truelist, M.quad);
E.truelist := E_2.truelist;
E.falselist := $\textit{merge}(E_1$.falselist, E_2.falselist); }

$E \rightarrow$ **not** E_1 { E.truelist := E_1.falselist;
E.falselist := E_1.truelist }

$E \rightarrow$ **(** E_1 **)** { E.truelist := E_1.truelist;
E.falselist := E_1.falselist }

Backpatching with Lists: Translation Scheme (cont'd)

```
$E \rightarrow \mathbf{id}_1$ relop $\mathbf{id}_2$
                  { E.truelist := makelist(nextquad());
                    E.falselist := makelist(nextquad() + 1);
                    emit('if' $\mathbf{id}_1$.place relop.op $\mathbf{id}_2$.place 'goto _');
                    emit('goto _') }
$E \rightarrow$ true      { E.truelist := makelist(nextquad());
                    E.falselist := nil;
                    emit('goto _') }
$E \rightarrow$ false     { E.falselist := makelist(nextquad());
                    E.truelist := nil;
                    emit('goto _') }
```

Flow-of-Control Statements and Backpatching: Grammar

$$
\begin{aligned}
S &\rightarrow \textbf{if } E \textbf{ then } S \\
&\mid \textbf{if } E \textbf{ then } S \textbf{ else } S \\
&\mid \textbf{while } E \textbf{ do } S \\
&\mid \textbf{begin } L \textbf{ end} \\
&\mid A \\
L &\rightarrow L \textbf{ ; } S \\
&\mid S
\end{aligned}
$$

Synthesized attributes:

S.nextlist	backpatch list for jumps to the next statement after S (or nil)
L.nextlist	backpatch list for jumps to the next statement after L (or nil)

S_1 ; S_2 ; S_3 ; S_4 ; S_4 …

```
100: Code for S1
200: Code for S2
300: Code for S3
400: Code for S4
500: Code for S5
```

Jumps out of S_1

backpatch(S_1.nextlist, 200)
backpatch(S_2.nextlist, 300)
backpatch(S_3.nextlist, 400)
backpatch(S_4.nextlist, 500)

Flow-of-Control Statements and Backpatching

$S \rightarrow A$ { S.nextlist := nil }

$S \rightarrow$ **begin** L **end**

{ S.nextlist := L.nextlist }

$S \rightarrow$ **if** E **then** $M\ S_1$

{ *backpatch*(E.truelist, M.quad);

S.nextlist := *merge*(E.falselist, S_1.nextlist) }

$L \rightarrow L_1$ **;** $M\ S$ { *backpatch*(L_1.nextlist, M.quad);

L.nextlist := S.nextlist; }

$L \rightarrow S$ { L.nextlist := S.nextlist; }

$M \rightarrow \varepsilon$ { M.quad := *nextquad*() }

Flow-of-Control Statements and Backpatching (cont'd)

$S \to$ **if** E **then** M_1 S_1 N **else** M_2 S_2
{ *backpatch*(E.truelist, M_1.quad);
backpatch(E.falselist, M_2.quad);
S.nextlist := *merge*(S_1.nextlist,
merge(N.nextlist, S_2.nextlist)) }

$S \to$ **while** M_1 E **do** M_2 S_1
{ *backpatch*(S_1,nextlist, M_1.quad);
backpatch(E.truelist, M_2.quad);
S.nextlist := E.falselist;
emit('**`goto`** _') }

$N \to \varepsilon$ { N.nextlist := *makelist*(*nextquad*());
emit('**`goto`** _') }

Translating Procedure Calls

$$\begin{aligned} S &\rightarrow \textbf{call id (}\ Elist\ \textbf{)} \\ Elist &\rightarrow Elist\ \textbf{,}\ E \\ &\quad\mid E \end{aligned}$$

foo(a+1, b, 7)

```
t1 := a + 1
t2 := 7
param t1
param b
param t2
call foo 3
```

Translating Procedure Calls

```
S → call id ( Elist )     { for each item p on queue do
                                emit('param' p);
                              emit('call' id.place |queue|) }
Elist → Elist , E         { append E.place to the end of queue }
Elist → E                 { initialize queue to contain only E.place }
```