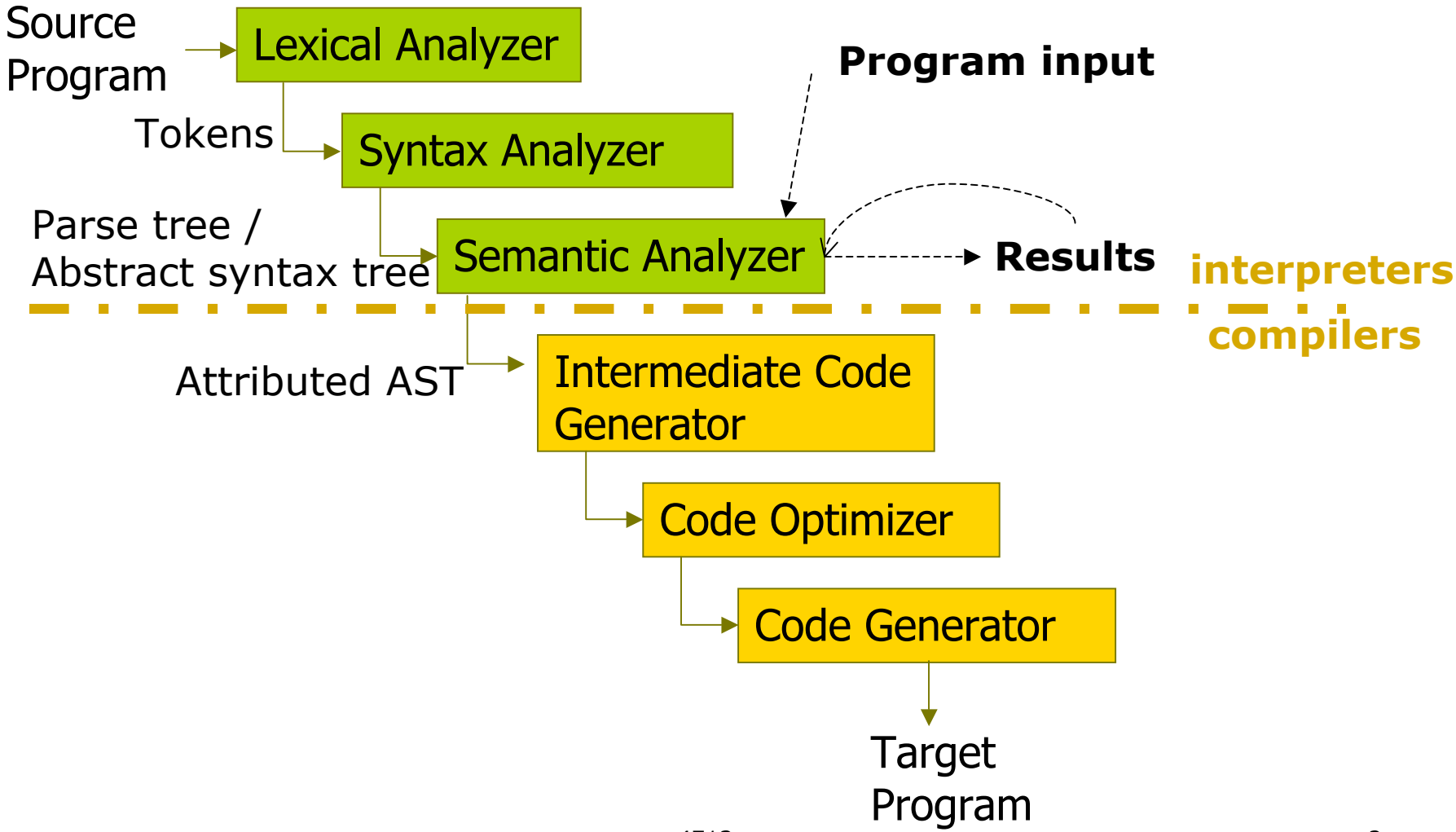
Intermediate Code Generation

Abstract syntax tree, three-address code, and type checking

Compile-time semantic evaluation



Intermediate code generation

- Static checker
  - Type checking, context-sensitive analysis
- Intermediate language between source and target
  - Multiple machines can be targeted
    - Attaching a different backend for each machine
    - Intel, PowerPC, UltraSparc can all share the same parser for C/C++
  - Multiple source languages can be supported
    - Attaching a different frontend (parser) for each language
    - Eg. C and C++ can share the same backend
  - Allow independent code optimizations
    - Multiple levels of intermediate representation
      - Low-level intermediate language: close to target machine
      - AST, post-fix, three-address code, stack-based code, …

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

  function call x() where x is not a function
  - may cause jump to a illegal op code

  int_add(3, 4.5)
  - It is an error to interpret bit pattern of 4.5 as an integer

- 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

Type expressions

A type expression is

- a basic type (eg. bool, char, float, int, void)
- a type name
- or formed by applying type constructor to other expressions
  - Array type: array(I,T) ➔ arrays with elements of type T and indices of type I.
    - float a[100]; ➔ a : array(int, float)
  - Tuple type: T1*T2*…*Tn ➔ cartesian product of types T1,T2…Tn
    - (int a,float b) ➔ (a,b) : int * float
  - Record type: record((fd1*T1)*(fd2*T2)…*(fdn*Tn)) ➔ records with a sequence of fields fd1,fd2,…,fdn of types T1,…,Tn
    - struct {int a,b;} xyz; ➔ xyz : record(a:int * b:int)
  - Pointer type: pointer(T) : pointer to an object of type T
    - double *p; ➔ p : pointer(double)
  - Function type D→T: functions that map values of type D to values of type T
    - int f (char* a, int b); ➔ f : pointer(char)*int →int

Structural equivalence of type expressions

- Two type expressions s and t are structurally equivalent if
  - s and t are the same basic type or
  - s and t are built using the same compound type constructor with the same components

```
Function structure-equiv(s, t) : boolean
  if s and t are the same basic type
    return true;
  else if s == array(s1,s2) and t == array(t1,t2)
    return structure-equiv(s1,t1) and structure-equiv(t1,t2)
  else if s == record(s1) and t == record(t1)
    return structure-equiv(s1, t1)
  else if s == s1 * s2 and t == t1 * t2  then
    return structure-equiv(s1,t1) and structure-equiv(t1,t2)
  else if s == pointer(s1) and t == pointer(t1)
   return structure-equiv(s1,t1)
  else if s == s1 → s2 and t == t1 → t2
   return structure-equiv(s1,t1) and structure-equiv(t1,t2)
  else     return false
```

Names for type expressions

- Type expressions can be given names and names can be used to define type expressions
  - struct XYZ { int a, b,c; };
  - Struct abc { XYZ* p1, p2; };
- Name equivalence
  - Each type name represent a different type
  - struct XYZ {int a,b,c; } and struct ABC {int a,b,c;} are different types

```
typedef  Cell*  Link;
Link  next, last;
Cell* p, q, r;
```

Do the variables all have identical types?
Yes if structural equivalence; no if name equivalence.

Evaluating types of expressions

```
P ::= D ;  E
D ::= D ; D | id : T
T ::= char | integer | T [ num ]
E ::= literal | num | id | E mod E | E[E]
```

```
P ::= D ; E
D ::= D ; D  |    id : T  { addtype(id.entry, T.type); }
T ::= char { T.type = char; } | integer { T.type = integer ;}
      | T1[num]  { T.type = array(num.val, T1.type);}
E ::= literal      { E.type = char;} |  num  { E.type = num;}
      | id  { E.type = lookupType(id.entry); }
      | E1 mod E2  {if (E1.type == integer && E2.type==integer)
                              E.type = integer; else E.type = type_error;}
      | E1[E2]  { if (E2.type == integer && E1.type==array(s,t))
                             E.type = t;   else   E.type = type_error; }
```

Type checking with coercion

- Implicit type conversion
  - When type mismatch happens, compilers automatically convert inconsistent types into required types
    - 2 + 3.5: convert 2 to 2.0 before adding 2.0 with 3.5

```
E ::= ICONST   { E.type = integer;}
E ::= FCONST { E.type = real; }
E ::= id          { E.type = lookup(id.entry); }
E ::= E1 op E2   { if (E1.type==integer and E2.type==integer)
                        E.type = integer;
                   else if (E1.type==integer and E2.type==real)
                        E.type=real;
                   else if (E1.type==real and E2.type==integer)
                         E.type=real;
                   else if (E1.type==real and E2.type==real)
                        E.type=real;
                   }
```

Type checking of statements

```
P ::= D ;  S
D ::= D ; D | id : T
T ::= char | integer | T [ num ]
S ::= E ; | {S S} | if (E) S | while (E) S
E ::= literal | num | id | E mod E | E[E]
```

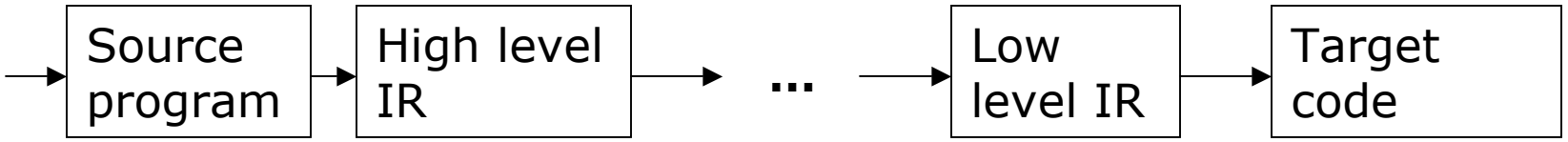
```
S ::= E ;   { if (E.type!=type_error) S.type = void;
              else S.type = type_error; }
    | '{' S1 S2 '}' {  if (S1.type == void)  S.type = S2.type;
                       else S.type = type_error; }
    | if '(' E ')' S1 { if (E.type == integer) S.type=S1.type;
                       else S.type=type_error; }
    | while '(' E ')' S1 { if (E.type == integer) S.type=S1.type;
                       else S.type=type_error; }
```

Type checking of function calls

```
P ::= D ;  E
D ::= D ; D | id : T | T id (Tlist)
Tlist ::= T, Tlist | T
T ::= char | integer | T [ num ]
E ::= literal | num | id | E mod E | E[E]  | E(Elist)
Elist ::= E, Elist | E
```

```
……
D ::= T1 id (Tlist)  { addtype(id.entry, fun(T1.type,Tlist.type)); }
Tlist ::= T, Tlist1  { Tlist.type = tuple(T1.type, Tlist1.type); }
        | T  { Tlist.type = T.type }
E ::= E1 ( Elist )   { if (E1.type == fun(r, p) && p ==Elist.type)
                          E.type = r ;  else E.type = type_error; }
Elist ::= E, Elist1   { Elist.type = tuple(E1.type, Elist1.type); }
        | E           { Elist.type = E.type; }
```

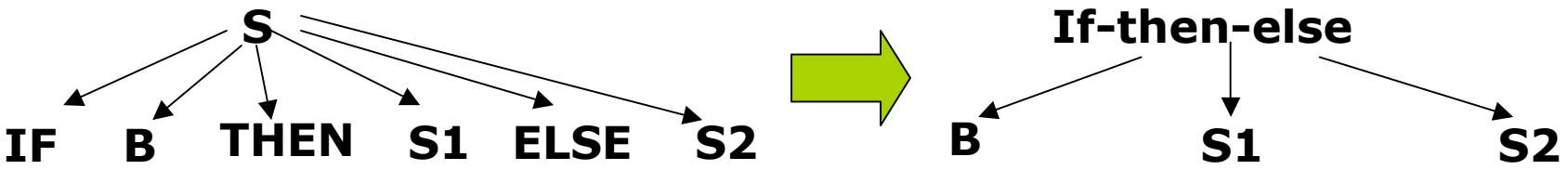
Intermediate representation



- A compiler might use a sequence of different IRs
  - High level IRs preserve high-level program structure
    - Eg., classes, loops, statements, expressions
  - Low level IRs support explicit expression and optimization of implementation details
- Selecting IR --- depends on the goal of each pass
  - Source-to-source translation: close to source language
    - Parse trees and abstract syntax trees
  - Translating to machine code: close to machine code
    - Linear three-address code
- External format of IR
  - Allows independent passes over IR

Abstract syntax tree

- Condensed form of parse tree for representing language constructs
  - Operators and keywords do not appear as leaves
    - They define the meaning of the interior (parent) node



  - Chains of single productions may be collapsed

Constructing AST

Grammar:

E ::= E + T | E – T | T
T ::= (E) | id | num

- Use syntax-directed definitions
  - Problem: construct an AST for each expression
  - Attribute grammar approach
    - Associate each non-terminal with an AST
      - Each AST: a pointer to a node in AST

        E.nptr T.nptr
  - Definitions: how to compute attribute?
    - Bottom-up: synthesized attribute

      if we know the AST of each child, how to compute the AST of the parent?

Constructing AST for expressions

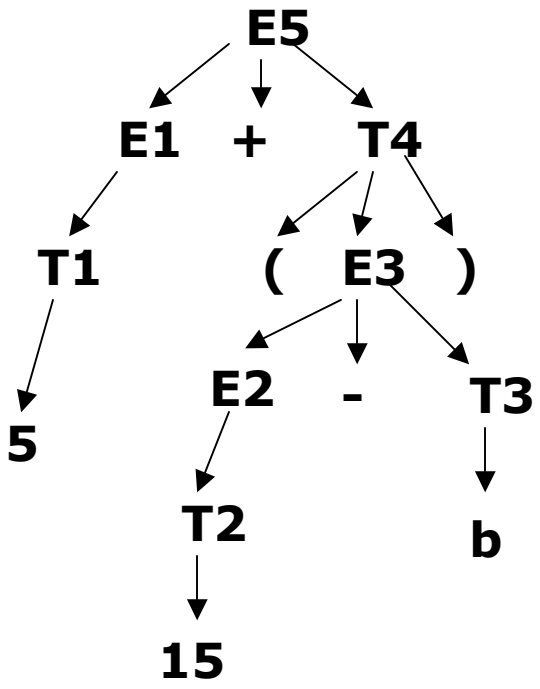
- Associate each non-terminal with an AST
  - E.nptr, T.nptr: a pointer to ASTtree
- Synthesized attribute definition:
  - If we know the AST of each child, how to compute the AST of the parent?

Production	Semantic rules
E ::= E1 + T	E.nptr=mknode_plus(E1.nptr,T.nptr)
E ::= E1 – T	E.nptr=mknode_minus(E1.nptr,T.nptr)
E ::= T	E.nptr=T.nptr
T ::= (E)	T.nptr=E.nptr
T ::= id	T.nptr=mkleaf_id(id.entry)
T ::= num	T.nptr=mkleaf_num(num.val)

Example: constructing AST

Bottom-up parsing: evaluate attribute at each reduction

Parse tree for 5+(15-b)



1. reduce 5 to T1 using T::=num:
   T1.nptr = leaf(5)
2. reduce T1 to E1 using E::=T:
   E1.nptr = T1.nptr = leaf(5)
3. reduce 15 to T2 using T::=num:
   T2.nptr=leaf(15)
4. reduce T2 to E2 using E::=T:
   E2.nptr=T2.nptr = leaf(15)
5. reduce b to T3 using T::=num:
   T3.nptr=leaf(b)
6. reduce E2-T3 to E3 using E::=E-T:
   E3.nptr=node('-',leaf(15),leaf(b))
7. reduce (E3) to T4 using T::=(E):
   T4.nptr=node('-',leaf(15),leaf(b))
8. reduce E1+T4 to E5 using E::=E+T:
   E5.nptr=node('+',leaf(5),
   node('-',leaf(15),leaf(b)))

Implementing AST in C

Grammar:

E ::= E + T | E – T | T
T ::= (E) | id | num

- Define different kinds of AST nodes
  - typedef enum {PLUS, MINUS, ID, NUM} ASTNodeTag;
- Define AST node

```
typedef struct ASTnode {
    AstNodeTag kind;
    union { symbol_table_entry* id_entry;
            int  num_value;
            struct ASTnode* opds[2];
          } description;
};
```

- Define AST node construction routines
  - ASTnode* mkleaf_id(symbol_table_entry* e);
  - ASTnode* mkleaf_num(int n);
  - ASTnode* mknode_plus(struct ASTnode* opd1, struct ASTNode* opd2);
  - ASTnode* mknode_minus(struct ASTnode* opd1, struct ASTNode* opd2);

Implementing AST in Java

Grammar:

E ::= E + T | E – T | T
T ::= (E) | id | num

- Define AST node

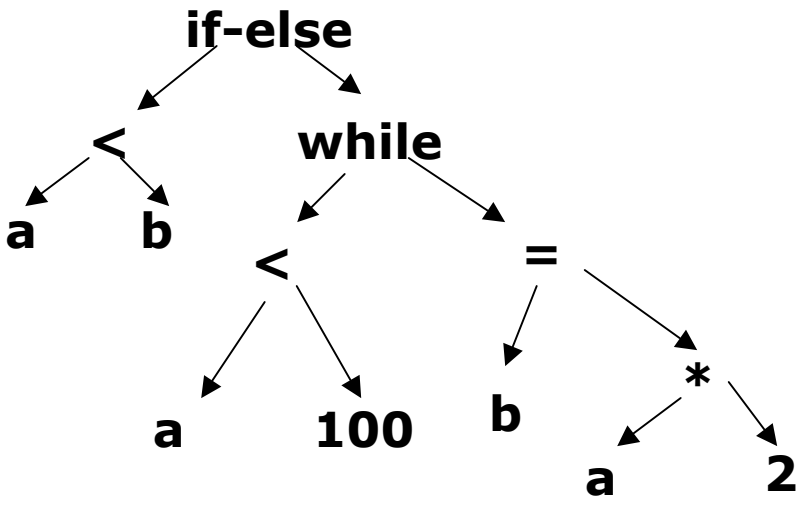
```
abstract class ASTexpression {
    public System.String toString();
}
class ASTidentifier extends ASTexpression
  {  private symbol_table_entry id_entry;  … }
class ASTvalue extends ASTexpression
  { private int num_value; … }
class ASTplus extends ASTexpression
  { private ASTnode opds[2]; … }
Class ASTminus extends ASTexpression
  { private ASTnode opds[2]; ... }
```

More ASTs

Abstract syntax:

S::= if-else E S S | while E S | E | _
E::= var | num | true | false | E bop E | uop E
bop ::= < | <= | > | >= | && | = | + | * | ….
uop ::= - | * | & | …

Abstract syntax tree




```
class ASTstmt {…}
class ASTifElse extends ASTstmt {
    private ASTexpr* cond;
    private ASTstmt* tbranch;
    private ASTstmt* fbranch; …}
class ASTwhile extends ASTstmt {
    private ASTexpr* cond;
    private ASTstmt* body;…}
class ASTexpr extends ASTstmt {…}
class ASTvar extends ASTexpr {…}
```

Three address code

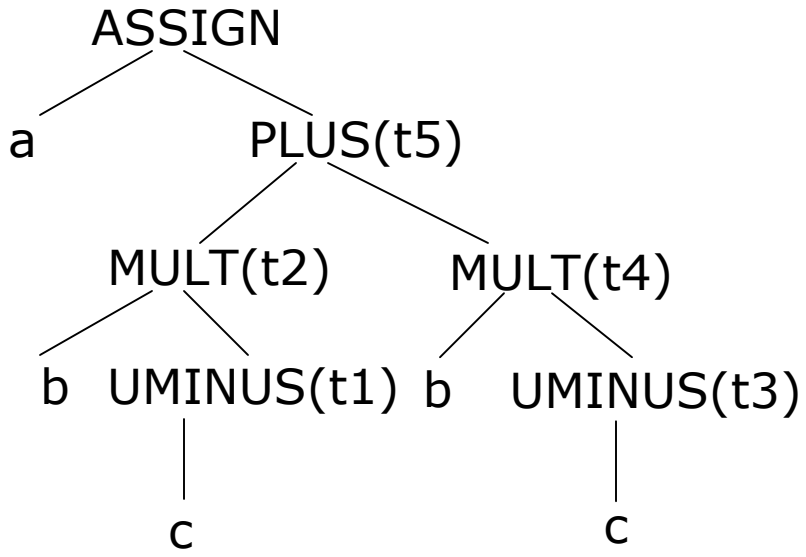
- Low level IL before final code generation
  - Linear representation of AST
  - Every instruction manipulates at most two operands and one result
- Assignment statements
  - x := y op z, where op is a binary operation
  - x := op y, where op is a unary operation
  - Copy statement: x:=y
  - Indexed assignments: x:=y[i] and x[i]:=y
  - Pointer assignments: x:=&y and x:=*y
- Control flow statements
  - Unconditional jump: goto L
  - Conditional jump: if x relop y goto L ; if x goto L; if False x goto L
  - Procedure calls: call procedure p with n parameters

```
param x1
param x2
  …
param xn
call p, n
```

Example: translating expressions

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

Abstract syntax tree:



Three-address code:

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

Storing three-address code

- Store all instructions in a quadruple table
  - Every instruction has four fields: op, arg1, arg2, result
  - The label of instructions ➔ index of instruction in table

Three-address code

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

Quadruple entries

	op	arg1	arg2	result
(0)	Uminux	c		t1
(1)	Mult	b	t1	t2
(2)	Uminus	c		t3
(3)	Mult	b	t3	t4
(4)	Plus	t2	t4	t5
(5)	Assign	t5		a

Translating assignment statement

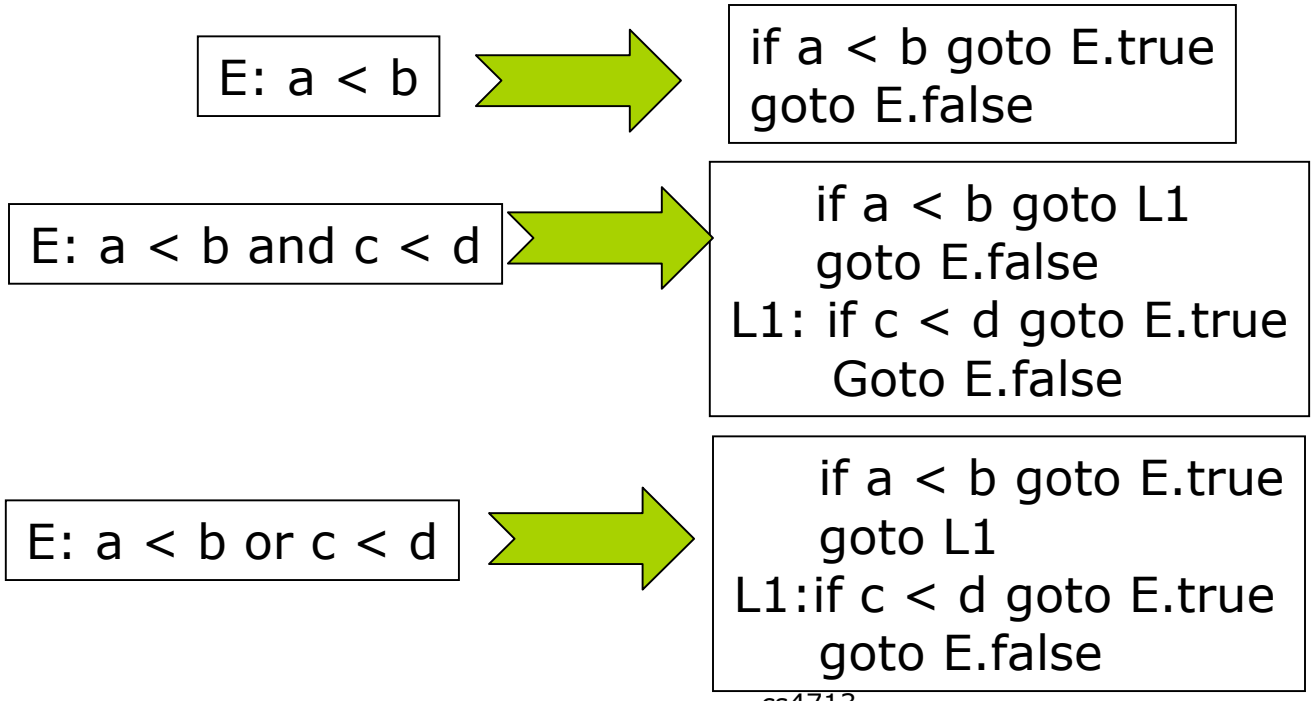
- For every non-terminal expression E
  - E.place: temporary variable used to store result
- Synthesized attributes
  - Bottom up traversal ensures E.place assigned before used
  - Can reuse temporary variables to reduce size of symbol table

Code concatenation

```
S ::= id '=' E   {gen(ASSIGN, E.place,0,lookup_place(id);}
E ::= E1 '+' E2
  {E.place=new_tmp(); gen(ADD,E1.place,E2.place,E.place);}
E ::= E1'*'E2
  {E.place=new_tmp(); gen(MULT,E1.place,E2.place,E.place);}
E ::= '-' E1
  {E.place = new_tmp(); gen(UMINUS,E1.place,0,E.place); }
E ::= (E1)  { E.place = E1.place; }
E ::= id  { E.place = lookup_place(id);  }
```

Control-flow translation of boolean expressions

- Two translation options
  - Same as translating regular expressions
  - Translate into control-flow branches
- For every boolean expression E
  - E.true: the label to goto if E is true
  - E.false: the label to goto if E is false



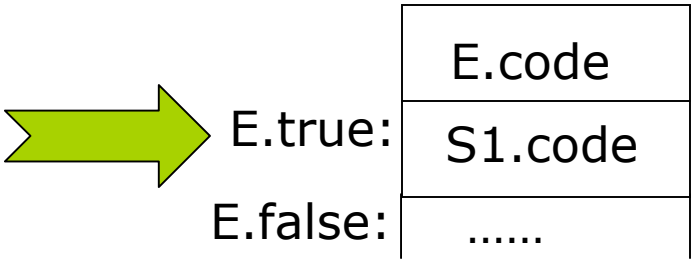
Translation schemes for boolean expressions

- For every boolean expression E
  - E.true: the label to goto if E is true
  - E.false: the label to goto if E is false

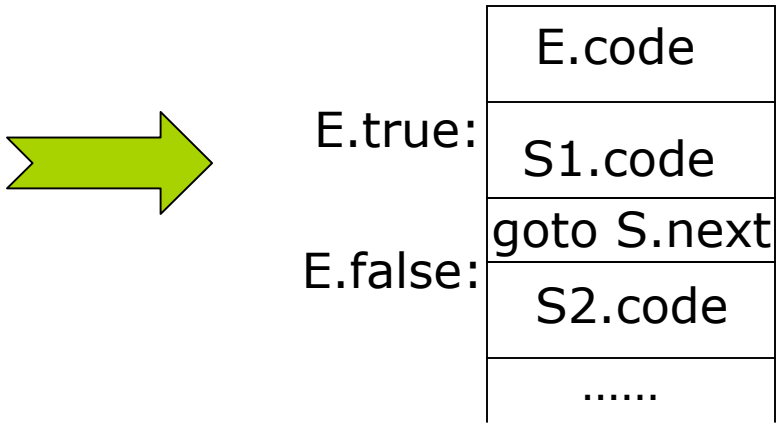
```
E ::= {E1.true=E.true; E1.false=new_label(); } E1 or
      {E2.true=E.true; E2.false=E.false; gen_label(E1.false) } E2
E ::= {E1.true=new_label(); E1.false=E.false; } E1 and
      {E2.true=E.true; E2.false=E.false; gen_label(E1.true) } E2
E ::= {E1.true=E.false; E1.false=E.true;} not E1
E ::= '(' {E1.true=E.true; E1.false=E.false; } E1 ')'
E ::= id1 relop id2
  { gen(IF,id1.place,id2.place,E.true); gen(GOTO,0,0,E.false);}
E ::= true { gen(GOTO,0,0,E.true); }
E ::= false { gen(GOTO,0,0,E.false); }
```

Translating control-flow statements

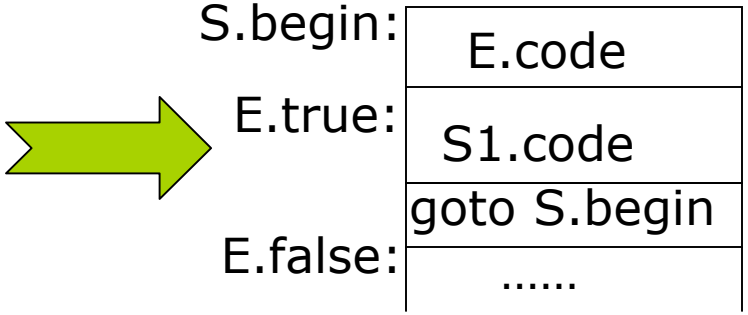
S::= if E then S1



S::= if E then S1 else S2



S::= While E do S1



Translating control-flow statements

- For every statement S
  - S.begin: the label of S; S.next: the label of statement following S
- For every boolean expression E
  - E.true: the goto label if E is true; E.false: the goto label if E is false

```
S ::= IF {E.true=new_label(); E.false=S.next; } E  THEN
          {S1.next=S.next; gen_label(E.true); } S1
S::= IF { E.true=new_label(); E.false=new_label();}  E THEN
          { S1.next=S.next; gen_label(E.true)} S1 ELSE
      {S2.next=S.next; gen(GOTO,0,0,S.next); gen_label(E.false) } S2
S ::= WHILE {S.begin=new_label(); E.true=new_label(); E.false=S.next;
                  gen_label(S.begin) } E
            DO {S1.next=S.begin; gen_label(E.true)} S1
              { gen(GOTO,0,0,S.begin); }
S ::= {S1.next=new_label(); } S1
       {S2.next=S.next; gen_label(S1.next) } S2
```

Translating control-flow statements

- Make two passes of the AST
  - First pass
    - Generate three-address code with symbolic labels
      - new_label(): create a new symbolic label (placeholder)
    - Determine place of every label
      - gen_label(symLabel): set place of symLabel
  - Second pass
    - Replace every symbolic label (placeholder) with the corresponding address in quadruple table