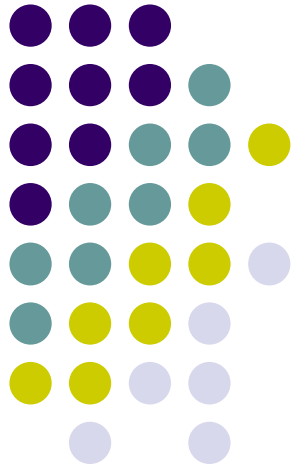
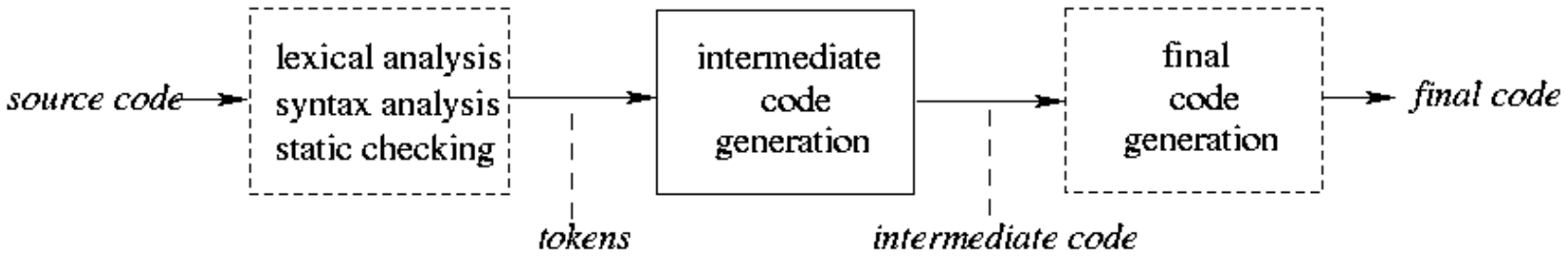
Intermediate Code Generation



Overview

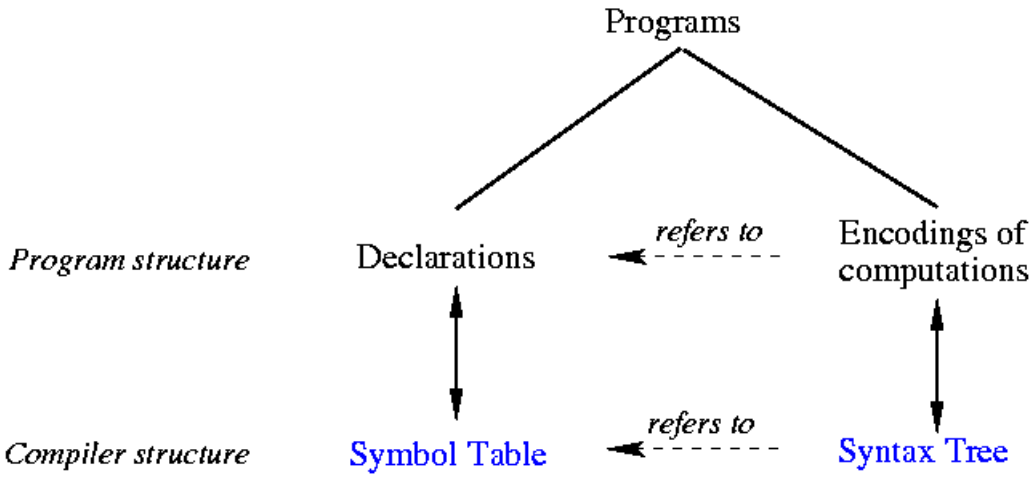


- Intermediate representations span the gap between the source and target languages:
  - closer to target language;
  - (more or less) machine independent;
  - allows many optimizations to be done in a machine-independent way.
- Implementable via syntax directed translation, so can be folded into the parsing process.

Types of Intermediate Languages

- *High Level Representations* (e.g., syntax trees):
  - closer to the source language
  - easy to generate from an input program
  - code optimizations may not be straightforward.
- *Low Level Representations* (e.g., 3-address code, RTL):
  - closer to the target machine;
  - easier for optimizations, final code generation;

Syntax Trees



A *syntax tree* shows the structure of a program by abstracting away irrelevant details from a parse tree.

- Each node represents a computation to be performed;
- The children of the node represents what that computation is performed on.

Syntax trees decouple parsing from subsequent processing.

Syntax Trees: Example

Grammar :

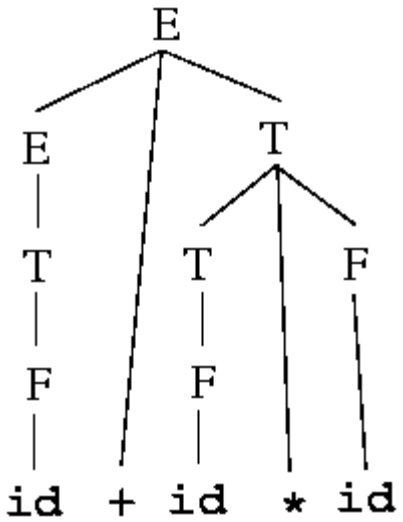
$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

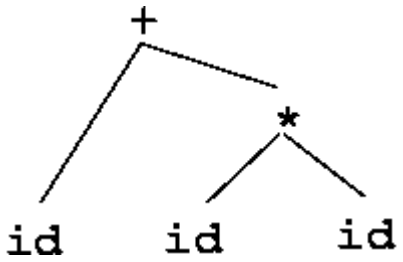
$F \rightarrow (E) \mid \mathbf{id}$

Input: **id** + **id** * **id**

Parse tree:

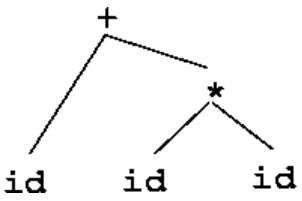


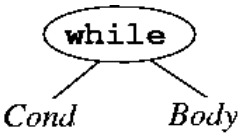
Syntax tree:



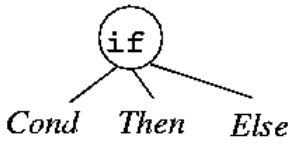
Syntax Trees: Structure

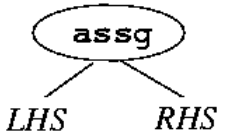
- Expressions:
  - leaves: identifiers or constants;
  - internal nodes are labeled with operators;
  - the children of a node are its operands.

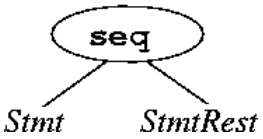




- Statements:
  - a node's label indicates what kind of statement it is;
  - the children correspond to the components of the statement.







Constructing Syntax Trees

General Idea: construct bottom-up using synthesized attributes.

E → E + E { $$ = *mkTree*(PLUS, $1, $3); }

S → if '(' E ')' S OptElse { $$ = *mkTree*(IF, $3, $5, $6); }

OptElse → else S { $$ = $2; }

| /* epsilon */ { $$ = NULL; }

S → while '(' E ')' S { $$ = *mkTree*(WHILE, $3, $5); }

mkTree(NodeType, Child1, Child2, …) allocates space for the tree node and fills in its node type as well as its children.

Three Address Code

- Low-level IR
- instructions are of the form '**x = y *op* z**,' where **x**, **y**, **z** are variables, constants, or "temporaries".

- At most one operator allowed on RHS, so no 'built-up" expressions.

  Instead, expressions are computed using temporaries (compiler-generated variables).

Three Address Code: Example

- *Source*:

```
if ( x + y*z > x*y + z)
    a = 0;
```

- *Three Address Code*:

```
tmp1 = y*z
tmp2 = x+tmp1          // x + y*z
tmp3 = x*y
tmp4 = tmp3+z          // x*y + z
if (tmp2 <= tmp4) goto L
a = 0
L:
```

An Intermediate Instruction Set

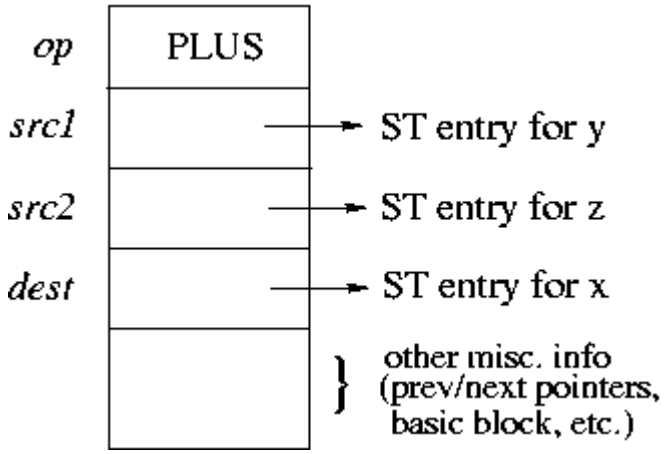
- *Assignment*:
  - x = y *op* z (*op* binary)
  - x = *op* y (*op* unary);
  - x = y
- *Jumps*:
  - if (x *op* y) goto L (L a label);
  - goto L
- *Pointer and indexed assignments*:
  - x = y[z]
  - y[z] = x
  - x = &y
  - x = *y
  - *y = x.
- *Procedure call/return*:
  - param x, k (x is the k^{th} param)
  - retval x
  - call p
  - enter p
  - leave p
  - return
  - retrieve x
- *Type Conversion*:
  - x = cvt_A_to_B y (A, B base types)
    e.g.: cvt_int_to_float
- *Miscellaneous*
  - label L

Three Address Code: Representation

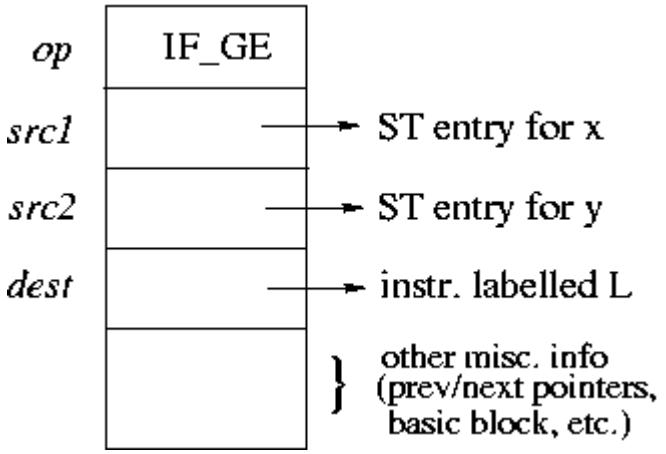
- Each instruction represented as a structure called a *quadruple* (or "*quad*"):
  - contains info about the operation, up to 3 operands.
  - for operands: use a bit to indicate whether constant or ST pointer.

E.g.:

x = y + z



if (x ≥ y) goto L



Code Generation: Approach

- function prototypes, global declarations:
  - save information in the global symbol table.
- function definitions:
  - function name, return type, argument type and number saved in global table (if not already there);
  - process formals, local declarations into local symbol table;
  - process body:
    - construct syntax tree;
    - traverse syntax tree and generate code for the function;
    - deallocate syntax tree and local symbol table.

Code Generation: Approach

Recursively traverse syntax tree:

- Node type determines action at each node;
- Code for each node is a (doubly linked) list of three-address instructions;
- Generate code for each node after processing its children

```
codeGen_stmt(synTree_node S)
{
    switch (S.nodetype) {
      case FOR:    … ; break;
      case WHILE : … ; break;
      case IF:     … ; break;
      case '=' :   … ; break;
    …
}
```

```
codeGen_expr(synTree_node E)
{
    switch (E.nodetype) {
      case '+':    … ; break;
      case '*' :   … ; break;
      case '–':    … ; break;
      case '/' :   … ; break;
    …
}
```

recursively process the children, then generate code for this node and glue it all together.

Intermediate Code Generation

Auxiliary Routines:

- *struct symtab_entry *newtemp(typename t)*

  creates a symbol table entry for new temporary variable each time it is called, and returns a pointer to this ST entry.

- *struct instr *newlabel()*

  returns a new label instruction each time it is called.

- *struct instr *newinstr(arg_1, arg_2, …)*

  creates a new instruction, fills it in with the arguments supplied, and returns a pointer to the result.

Intermediate Code Generation…

- struct symtab_entry *newtemp(t)
  {
  struct symtab_entry *ntmp = malloc(…); /* check: ntmp == NULL? */
  ntmp->name = *…create a new name that doesn't conflict…*
  ntmp->type = t;
  ntmp->scope = LOCAL;
  return ntmp;
  }
- struct instr *newinstr(opType, src1, src2, dest)
  {
  struct instr *ninstr = malloc(…); /* check: ninstr == NULL? */
  ninstr->op = opType;
  ninstr->src1 = src1; ninstr->src2 = src2; ninstr->dest = dest;
  return ninstr;
  }

❖ Advantages of intermediate code

- There are certain advantages of generating machine independent intermediate code:

1. A compiler for a different machine can be created by attaching a different back end to an existing front ends of each machine.
2. A compiler for a different source language(on same machine) can be created by proving different front ends for corresponding source languages to existing backend.
3. A machine independent code optimizer can be applied to intermediate co[illegible] order to optimize the code generation.

❖ Different intermediate forms / representation

- There are three types of intermediate representation,

1. Abstract syntax tree
2. Postfix notation
3. Three address code

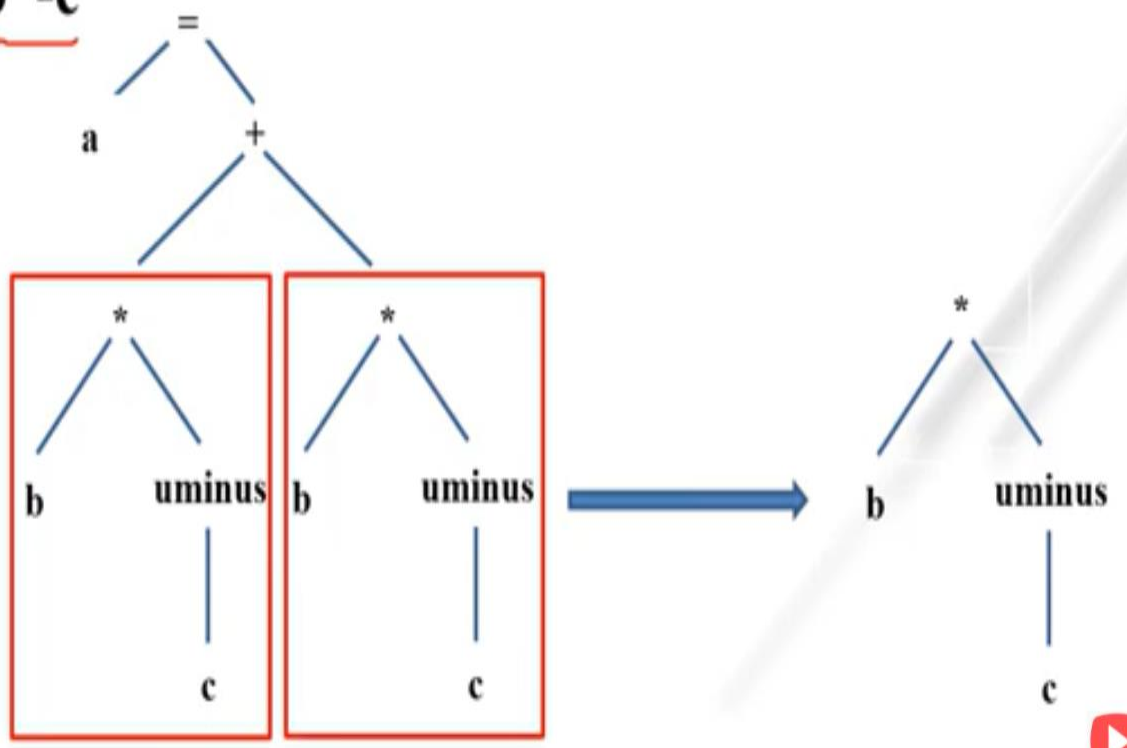
1. Abstract syntax tree & DAG

- ✓ A syntax tree depicts the natural hierarchical structure of a source program.
- ✓ A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified.
- Example: String: **a = b*-c+b*-c**

$t_1 = uminus\ c$

$t_2 = b * t_1$

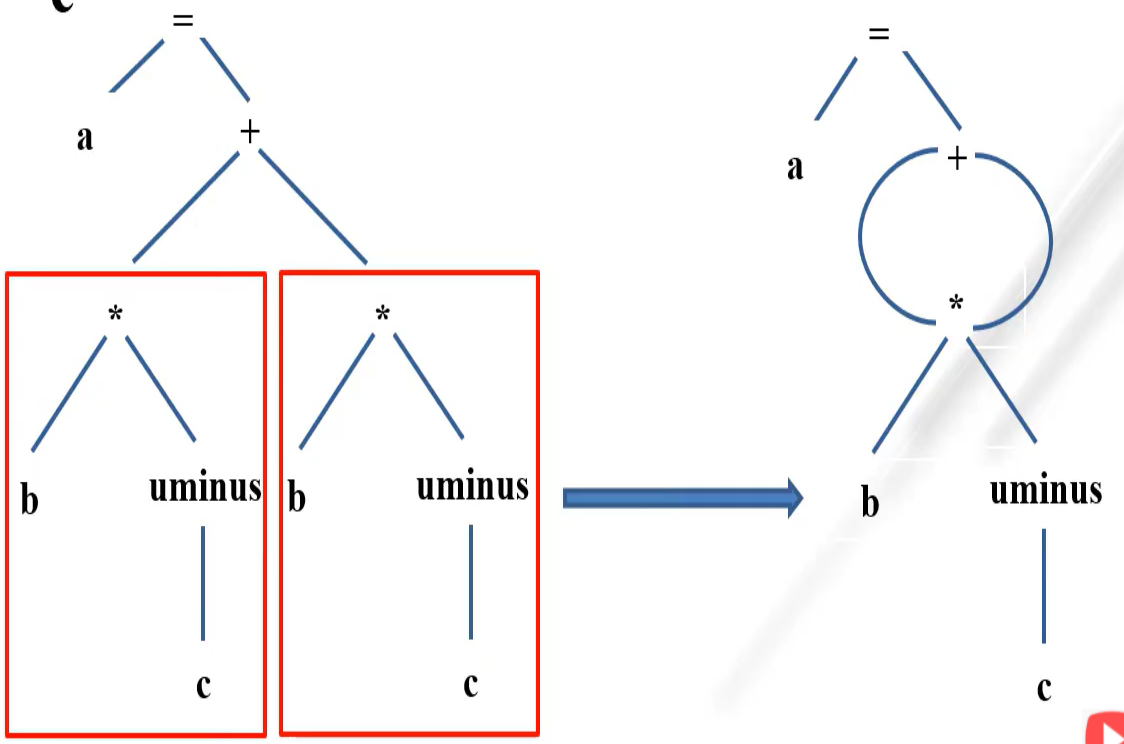
$t_3 = t_2 + t_2$



Syntax Tree

1. Abstract syntax tree & DAG

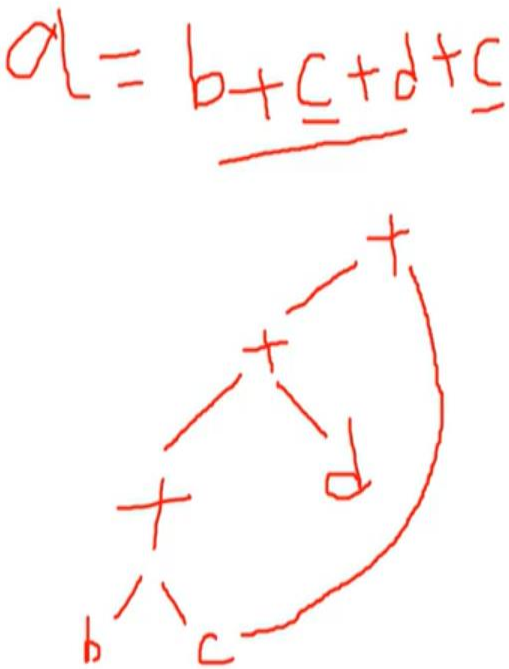
- ✓ A syntax tree depicts the natural hierarchical structure of a source program.
- ✓ A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified.
- Example: String: **a = b*-c+b*-c**

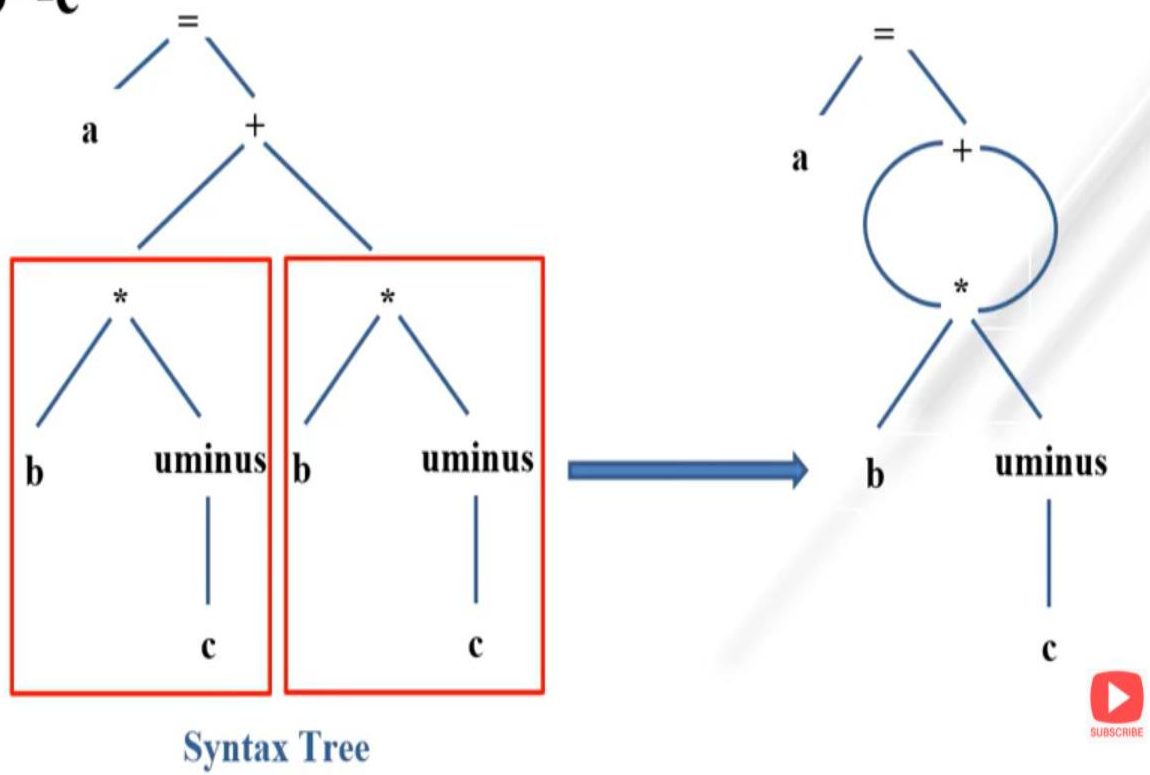


Syntax Tree

1. Abstract syntax tree & DAG

- ✓ A syntax tree depicts the natural hierarchical structure of a source program.
- ✓ A DAG (Directed Acyclic Graph) gives the same information but in a more compact way because common sub-expressions are identified.
- Example: String: **a = b*-c+b*-c**





Syntax Tree

2. Postfix notation

- ✓ This is most natural way of representation in expression evaluation in compiler.
- ✓ In postfix notation the operands occurs first and then operators are arranged.
- Example: Consider the following string: **a + b * c + d * e ↑ f**
- **The postfix notation of the string is: abc*+def↑*+**

2. Postfix notation

- string: a + b * c + d * e ↑ f
- postfix notation is: abc*+def↑*+

✓ *To obtain the postfix notation, evaluate the expression according to precedence.*

a + b * c + d * e ↑ f	where,
a + T1 + d * e ↑ f	T1 = bc*
T2 + d * e ↑ f	T2 = aT1+
T2 + d * T3	T3 = ef↑
T2 + T4	T4 = dT3*
T5	T5 = T2T4+

backward Substitution the value

of temporary variables

2. Postfix notation

- string: a + b * c + d * e ↑ f
- postfix notation is: abc*+def↑*+

✓ *To obtain the postfix notation, evaluate the expression according to precedence.*

a + b * c + d * e ↑ f	where,
a + T1 + d * e ↑ f	T1 = bc*
T2 + d * e ↑ f	T2 = aT1+
T2 + d * T3	T3 = ef↑
T2 + T4	T4 = dT3*
T5	T5 = T2T4+

backward Substitution the value of temporary variables

T5
T2T4+
T2dT3*+
T2def↑*+
aT1+def↑*+
abc*+def↑*+

Postfix notation

3. Three address code

- ✓ Three address code is a sequence of statements of the general form,

$$a = b \text{ op } c$$

- ✓ Where a, b or c are the operands that can be names or constants and op stands for any operator.

- Example: a = b + c + d

Three address code for above expression is:

$t_1 = b+c$

$t_2 = t_1+d$

$a = t_2$

$t_1 = a+b+c$

- ✓ Here t_1 and t_2 are the temporary names generated by the compiler.
- ✓ Only single operator on the right side of the expression is allowed at a time.
- ✓ There are at most three addresses allowed (two for operands and one for result). Hence, this representation is called three-address code.

❖ Different Representation of Three Address Code / Implementation of Three Address Code

- There are three types of representation used for three address code,
  1. Quadruples
  2. Triples
  3. Indirect triples
- Ex: x= -a*b + -a*b

  t_1= uminus a

  $t_2 = t_1 * b$

  t_3= uminus a

  $t_4 = t_3 * b$

  $t_5 = t_2 + t_4$

  x= t_5

1. Quadruple

- ✓ The quadruple is a structure with at the most four fields such as op, arg1, arg2 and result.
- ✓ The op field is used to represent the internal node for operator. The arg1 and arg2 represent the two operands. And result field is used to store the result of an expression.
- ✓ The three address statements with unary operator like a = -y or a = y do not use agr2.

x= -a*b + -a*b

t_1= uminus a

$t_2 = t_1 * b$

t_3= uminus a

$t_4 = t_3 * b$

$t_5 = t_2 + t_4$

x= t_5

Quadruple representation

	Operator	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

2. Triple

- ✓ In triple, temporaries are not used instead of that pointers in the symbol table are used directly
- ✓ If we do so, three address statements can be represented by records with only three fields: op, arg1 and arg2.
- ✓ Numbers in the round bracket ( ) are used to represent pointers into the triple structure.

Quadruple representation

	Operator	Arg1	Arg2	Result
(0)	uminus	a		t_1
(1)	*	t_1	b	t_2
(2)	uminus	a		t_3
(3)	*	t_3	b	t_4
(4)	+	t_2	t_4	t_5
(5)	=	t_5		x

Triple representation

	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

3. Indirect Triples

- ✓ In the indirect triple representation the listing of triples has been done. And listing pointers are used instead of using statement.
- ✓ This implementation is called indirect triples.

Triple representation

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(0)	b
(2)	uminus	a	
(3)	*	(2)	b
(4)	+	(1)	(3)
(5)	=	x	(4)

Indirect Triple representation

	Statement
(0)	(14)
(1)	(15)
(2)	(16)
(3)	(17)
(4)	(18)
(5)	(19)

No.	Operator	Arg1	Arg2
(0)	uminus	a	
(1)	*	(14)	b
(2)	uminus	a	
(3)	*	(16)	b
(4)	+	(15)	(17)
(5)	=	x	(18)

❑ Example

✓ Translate the arithmetic expression (a + b * c) + d + (a + b * c) – d + e

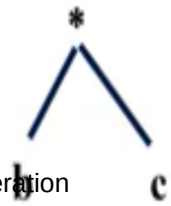
into:

1) Syntax tree
2) Postfix notation
3) Three- address code

Solution:

1) Syntax tree for the expression: (a + b * c) + d + (a + b * c) – d + e

Evaluating expression according to precedence
$ (+ *) + + (+ *) - + $
$ < (< + < * >) + + (+ *) - + $

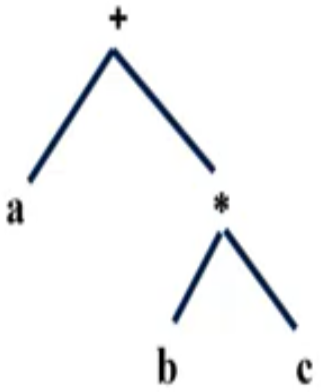


Syntax tree

Solution:

1) Syntax tree for the expression: **(a + b * c) + d + (a + b * c) – d + e**

Evaluating expression according to precedence
$ (+ *) + + (+ *) - + $
$ < (< + < * >) + + (+ *) - + $
$ < (< + >) + + (+ *) - + $
$ < + > + (+ *) - + $

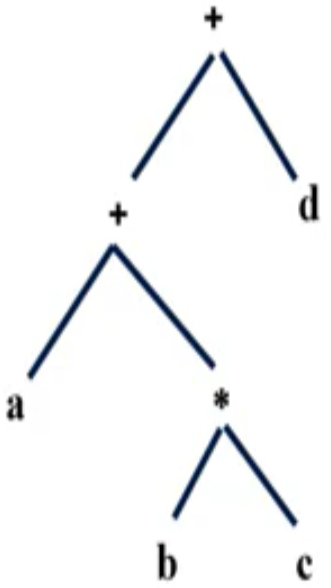


Syntax tree

Solution:

1) Syntax tree for the expression: **(a + b * c) + d + (a + b * c) – d + e**

Evaluating expression according to precedence
$ (+ *) + + (+ *) - + $
$ < (< + < * >) + + (+ *) - + $
$ < (< + >) + + (+ *) - + $
$ < + > + (+ *) - + $
$ < + < (< + < * >) - + $

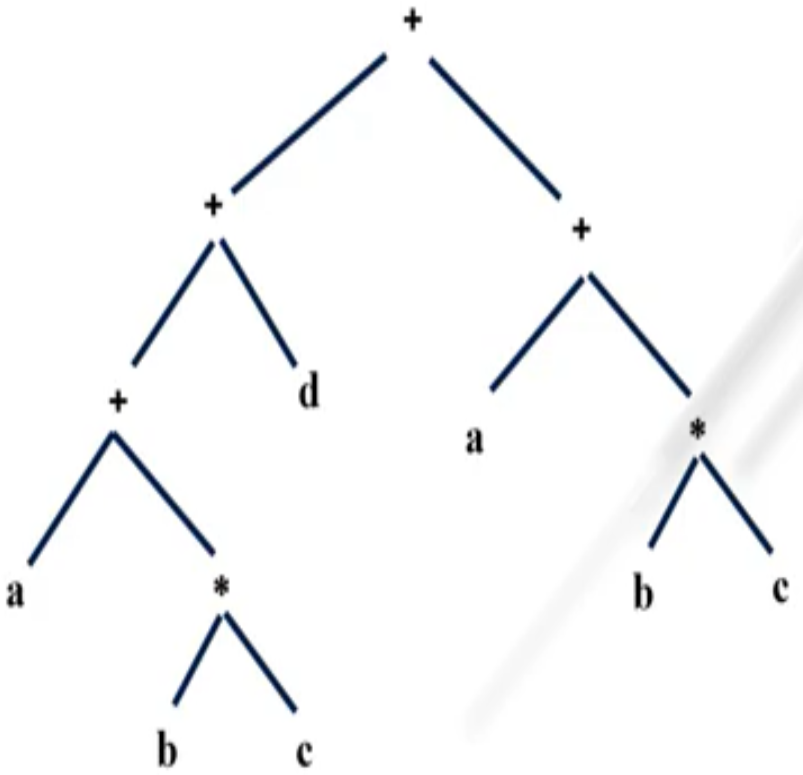


Syntax tree

Solution:

1) Syntax tree for the expression: **(a + b * c) + d + (a + b * c) – d + e**

Evaluating expression according to precedence
$ (+ *) + + (+ *) - + $
$ < (< + < * >) + + (+ *) - + $
$ < (< + >) + + (+ *) - + $
$ < + > + (+ *) - + $
$ < + < (< + < * >) - + $
$ < + < (< + >) - + $
$ < + > - + $
$ < - > + $

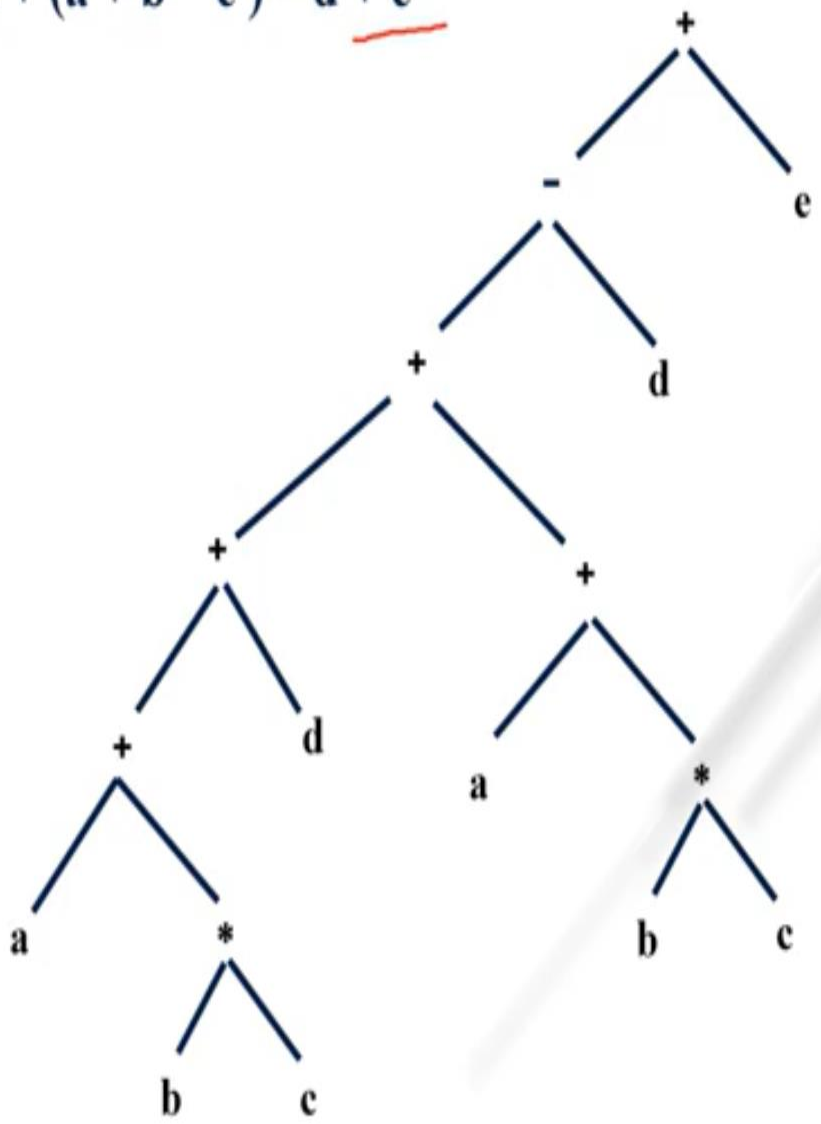


Syntax tree

Solution:

1) Syntax tree for the expression: **(a + b * c) + d + (a + b * c) – d + e**

Evaluating expression according to precedence
$ (+ *) + + (+ *) - + $
$ < (< + < * >) + + (+ *) - + $
$ < (< + >) + + (+ *) - + $
$ < + > + (+ *) - + $
$ < + < (< + < * >) - + $
$ < + < (< + >) - + $
$ < + > - + $
$ < - > + $
$ < + > $



Syntax tree

2) Postfix notation for the expression: (a + b * c) + d + (a + b * c) – d + e

(a + b * c) + d + (a + b * c) – d + e	where,
(a + T1) + d + (a + b * c) – d + e	T1 = bc*
T2 + d + (a + b * c) – d + e	T2 = aT1+
T3 + (a + b * c) – d + e	T3 = T2d+
T3 + (a + T4) – d + e	T4 = bc*
T3 + T5 – d + e	T5 = aT4+
T6 – d + e	T6 = T3T5+
T7 + e	T7 = T6d-
T8	T8 = T7e+

T8
T7 e +
T6 d - e +
T3 T5 + d – e +
T3 aT4 + + d – e +
T3 abc * + + d - e +
T2 d + abc * + + d – e +
aT1+ d + abc * + + d – e +
abc * + d + abc * + + d – e +

backward Substitution the value

of temporary variables

Postfix notation

3) Three- address code for the expression: (a + b * c) + d + (a + b * c) – d + e

$t_1 = b * c$
$t_2 = a + t1$
$t_3 = t2 + d$
$t_4 = b * c$
$t_5 = a + t4$
$t_6 = t3 + t5$
$t_7 = t6 - d$
$t_8 = t7 + e$

Three - address Code

❑ Example

✓ **Draw DAG for the expression (a + b * c) + d + (a + b * c) – d + e**

Three address code for DAG

$t_1 = b * c$

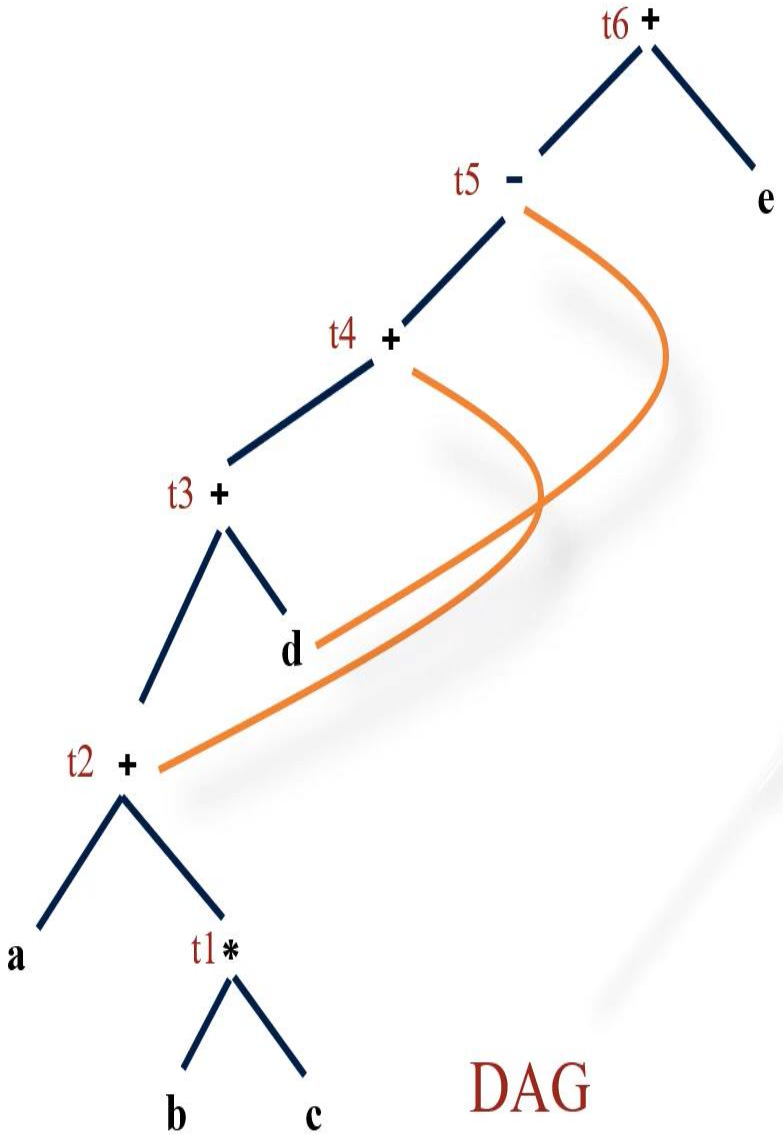
$t_2 = a + t1$

$t_3 = t2 + d$

$t_4 = t3 + t2$

$t_5 = t4 - d$

$t_6 = t5 + e$



DAG

Intermediate Code for a Function

Code generated for a function *f*:

- begin with 'enter *f*', where *f* is a pointer to the function's symbol table entry:
  - this allocates the function's activation record;
  - activation record size obtained from *f*'s symbol table information;
- this is followed by code for the function body;
  - generated using codeGen_stmt(…) [to be discussed soon]
- each return in the body (incl. any implicit return at the end of the function body) are translated to the code

  leave *f* /* clean up: *f* a pointer to the function's symbol table entry */

  return /* + associated return value, if any */

Simple Expressions

Syntax tree node for expressions augmented with the following fields:

- type: the type of the expression (or “error”);
- code: a list of intermediate code instructions for evaluating the expression.
- place: the location where the value of the expression will be kept at runtime:

Simple Expressions

Syntax tree node for expressions augmented with the following fields:

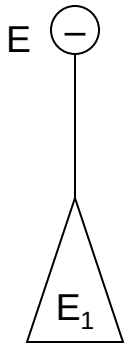
- type: the type of the expression (or “error”);
- code: a list of intermediate code instructions for evaluating the expression.
- place: the location where the value of the expression will be kept at runtime:
  - When generating intermediate code, this just refers to a symbol table entry for a variable or temporary that will hold that value;
  - The variable/temporary is mapped to an actual memory location when going from intermediate to final code.

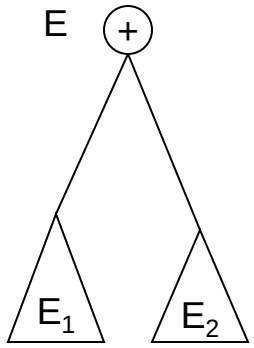
Simple Expressions 1

Syntax tree node E	***Action during intermediate code generation***
E **intcon**	codeGen_expr(E) { /* E.nodetype == INTCON; */ E.place = *newtemp*(E.type); E.code = 'E.place = intcon.val'; }
E **id**	codeGen_expr(E) { /* E.nodetype == ID; */ /* E.place is just the location of **id** (nothing more to do) */ E.code = NULL; }

Simple Expressions 2

Syntax tree node E	***Action during intermediate code generation***

	codeGen_expr(E)
	{
	/* E.nodetype == UNARY_MINUS */
	codeGen_expr(E_1); /* recursively traverse E_1, generate code for it */
	E.place = *newtemp*(E.type); /* allocate space to hold E's value */
	E.code = E_1.code ⊕ *newinstr*(UMINUS, E_1.place, NULL, E.place);
	}

	codeGen_expr(E)
	{
	/* E.nodetype == '+' … other binary operators are similar */
	codeGen_expr(E_1);
	codeGen_expr(E_2); /* generate code for E_1 and E_2 */
	E.place = *newtemp*(E.type); /* allocate space to hold E's value */
	E.code = E_1.code ⊕ E_2.code ⊕ *newinstr*(PLUS, E_1.place, E_2.place, E.place);
	}

Accessing Array Elements 1

- Given:
  - an array $A[lo \ldots hi]$ that starts at address b;
  - suppose we want to access $A[\ i\ ]$.
- We can use indexed addressing in the intermediate code for this:
  - $A[\ i\ ]$ is the $(i + lo)^{th}$ array element starting from address b.
  - Code generated for $A[\ i\ ]$ is:

```
t1 = i + lo
t2 = A[ t1 ]    /* A being treated as a 0-based array at this level. */
```

Accessing Array Elements 2

- In general, address computations can't be avoided, due to pointer and record types.
- Accessing $A[\ i\ ]$ for an array $A[lo \ldots hi]$ starting at address b, where each element is w bytes wide:

  Address of $A[\ i\ ]$ is $b + (\ i - lo\ ) * w$

  $= (b - lo * w) + i * w$

  $= k_A + i * w.$

  k_A depends only on A, and is known at compile time.

- Code generated:

```
t1 = i * w
t2 = k_A + t1     /* address of  A[ i ]  */
t3 = *t2
```

Accessing Structure Fields

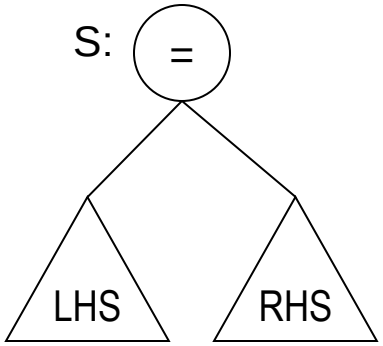
- Use the symbol table to store information about the order and type of each field within the structure.
  - Hence determine the distance from the start of a struct to each field.
  - For code generation, add the displacement to the base address of the structure to get the address of the field.
- *Example*: Given

```
struct s { … }  *p;
…
x = p→a;        /* a is at displacement δa within struct s */
```

The generated code has the form:

```
t1 = p + δa     /* address of p→a */
x = *t1
```

Assignments




```
codeGen_stmt(S):
/* base case: S.nodetype = 'S' */
    codeGen_expr(LHS);
    codeGen_expr(RHS);
    S.code = LHS.code
                ⊕ RHS.code
                ⊕ newinstr(ASSG,
                            LHS.place,
                            RHS.place) ;
```

Code structure:

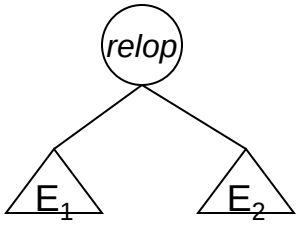
evaluate LHS

evaluate RHS

copy value of RHS into LHS

Logical Expressions 1

- *Syntax tree node*:



- *Naïve but Simple Code* (TRUE=1, FALSE=0):

```
        t1 = { evaluate E1
        t2 = { evaluate E2
        t3 = 1          /* TRUE */
        if ( t1 relop t2 ) goto L
        t3 = 0          /* FALSE */
    L: …
```

- *Disadvantage*: lots of unnecessary memory references.

Logical Expressions 2

- *Observation*: Logical expressions are used mainly to direct flow of control.
- *Intuition*: "tell" the logical expression where to branch based on its truth value.
  - When generating code for *B*, use two inherited attributes, *trueDst* and *falseDst*. Each is (a pointer to) a *label* instruction.

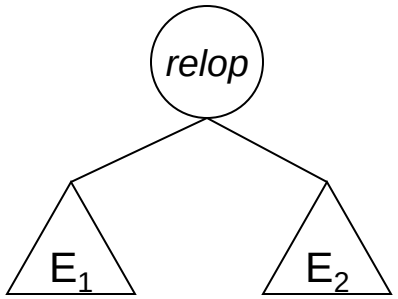
    E.g.: for a statement **if** (*B*) S_1 **else** S_2 :

    B.trueDst = start of S_1

    B.falseDst = start of S_2
  - The code generated for *B* jumps to the appropriate label.

Logical Expressions 2: cont'd

Syntax tree:



codeGen_bool(B, *trueDst*, *falseDst*):

/* base case: B.nodetype == *relop* */

B.code = E1.code

⊕ E2.code

⊕ *newinstr*(*relop,* E1.place, E2.place, *trueDst*)

⊕ *newinstr*(GOTO, *falseDst*, NULL, NULL);

Example: $B \Rightarrow$ x+y > 2*z.

Suppose *trueDst* = Lbl1, *falseDst* = Lbl2.

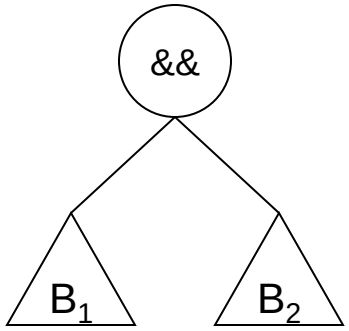
$E_1 \equiv$ x+y, E_1.place = tmp_1, E_1.code $\equiv \langle$ 'tmp_1 = x + y' $\rangle$

$E_2 \equiv$ 2*z, E_2.place = tmp_2, E_2.code $\equiv \langle$ 'tmp_2 = 2 * z' $\rangle$

B.code = E_1.code ⊕ E_2.code ⊕ 'if (tmp_1 > tmp_2) goto Lbl1' ⊕ goto Lbl2

= $\langle$ 'tmp_1 = x + y' , 'tmp_2 = 2 * z', 'if (tmp_1 > tmp_2) goto Lbl1' , goto Lbl2 $\rangle$

Short Circuit Evaluation



codeGen_bool (B, *trueDst*, *falseDst*):

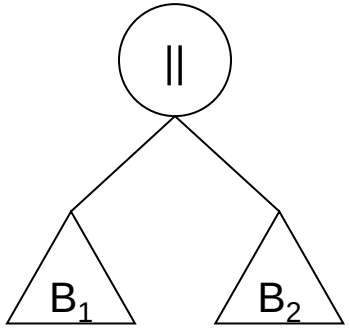
/* recursive case 1: B.nodetype == '&&' */

L_1 = *newlabel*( );

codeGen_bool(B_1, L_1, *falseDst*);

codeGen_bool(B_2, *trueDst*, *falseDst*);

B.code = B_1.code ⊕ L_1 ⊕ B_2.code;



codeGen_bool (B, *trueDst*, *falseDst*):

/* recursive case 2: B.nodetype == '||' */

L_1 = *newlabel*( );

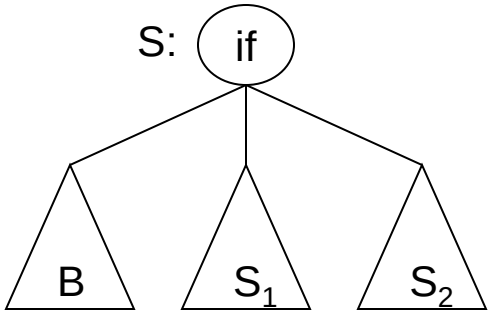
codeGen_bool(B_1, *trueDst*, L_1);

codeGen_bool(B_2, *trueDst*, *falseDst*);

B.code = B_1.code ⊕ L_1 ⊕ B_2.code;

Conditionals

Syntax Tree:

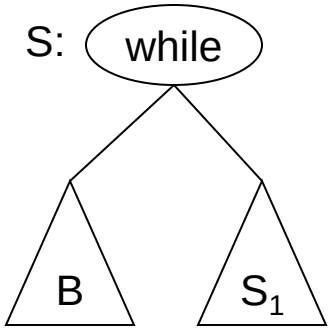


- *Code Structure*:

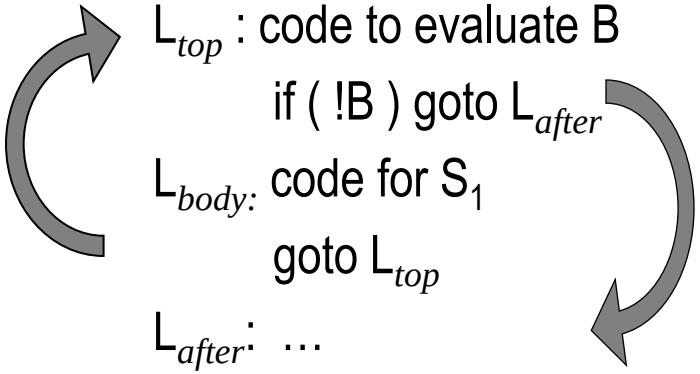
code to evaluate B
L_{then}: code for S1
goto L_{after}
L_{else}: code for S2
L_{after} : …

codeGen_stmt(S):
/* S.nodetype == 'IF' */
L_{then} = *newlabel*();
L_{else} = *newlabel*();
L_{after} = *newlabel*();
codeGen_bool(B, L_{then} , L_{else});
codeGen_stmt(S_1);
codeGen_stmt(S_2);
S.code = B.code
⊕ L_{then}
⊕ S_1.code
⊕ *newinstr*(GOTO, L_{after})
⊕ L_{else}
⊕ S_2.code
⊕ L_{after} ;

Loops 1

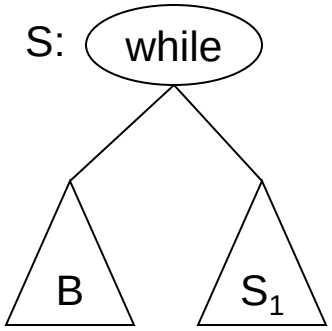


Code Structure:

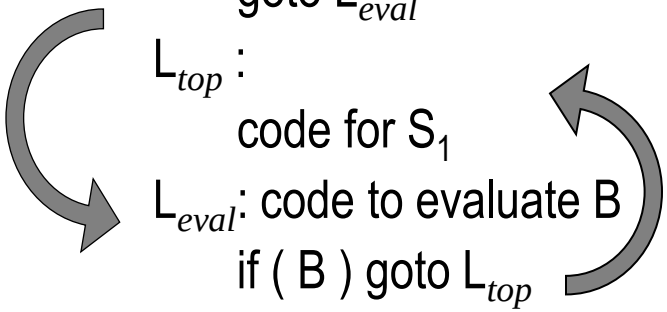



```
codeGen_stmt(S):
/* S.nodetype == 'WHILE' */
    L_top = newlabel();
    L_body = newlabel();
    L_after = newlabel();
    codeGen_bool(B, L_body, L_after);
    codeGen_stmt(S_1);
    S.code = L_top
           ⊕ B.code
           ⊕ L_body
           ⊕ S1.code
           ⊕ newinstr(GOTO, L_top)
           ⊕ L_after ;
```

Loops 2



Code Structure:



This code executes fewer branch ops.

codeGen_stmt(S):

/* S.nodetype = 'WHILE' */

L_{top} = *newlabel*();

L_{eval} = *newlabel*();

L_{after} = *newlabel*();

codeGen_bool(B, $L_{top,}$ L_{after});

codeGen_stmt(S_1);

S.code =

newinstr(GOTO, L_{eval})

⊕ L_{top}

⊕ S_1.code

⊕ L_{eval}

⊕ B.code

⊕ L_{after} ;

Multi-way Branches: switch statements

- *Goal*:

  generate code to (efficiently) choose amongst a fixed set of alternatives based on the value of an expression.

- *Implementation Choices*:
  - *linear search*
    - best for a small number of case labels ($\approx$ 3 or 4)
    - cost increases with no. of case labels; later cases more expensive.
  - *binary search*
    - best for a moderate number of case labels ($\approx$ 4 – 8)
    - cost increases with no. of case labels.
  - *jump tables*
    - best for large no. of case labels ($\geq$ 8)
    - may take a large amount of space if the labels are not well-clustered.

Background: Jump Tables

- A jump table is an array of code addresses:
  - *Tbl*[i] is the address of the code to execute if the expression evaluates to i.
  - if the set of case labels have “holes”, the correspond jump table entries point to the default case.
- *Bounds checks*:
  - Before indexing into a jump table, we must check that the expression value is within the proper bounds (if not, jump to the default case).
  - The check

    $$lower\_bound \leq exp\_value \leq upper\ bound$$

    can be implemented using a single unsigned comparison.

Jump Tables: cont'd

- Given a ***switch*** with max. and min. case labels c_{max} and c_{min}, the jump table is accessed as follows:

Instruction	***Cost*** (cycles)
$t_0 \leftarrow$ value of expression	…
$t_0 = t_0 - c_{min}$	1
if $\neg(t_0 \leq_u c_{max} - c_{min})$ goto *DefaultCase*	4 to 6
t_1 = JmpTbl_BaseAddr	1
t_1 += 4*t_0	1
jmp *t1	3 to 5
	$\sum$: 10 to 14

Jump Tables: Space Costs

- A jump table with max. and min. case labels c_{max} and c_{min} needs $\approx c_{max} - c_{min}$ entries.

  This can be wasteful if the entries aren't "dense enough", e.g.:

```
switch (x) {
  case 1: …
  case 1000: …
  case 1000000: …
}
```

- Define the ***density*** of a set of case labels as

  density = no. of case labels / ($c_{max} - c_{min}$)

- Compilers will not generate a jump table if density below some threshold (typically, 0.5).

Switch Statements: Overall Algorithm

- if no. of case labels is small ($\leq$ ~ 8), use linear or binary search.
  - use no. of case labels to decide between the two.
- if density $\geq$ threshold (~ 0.5) :
  - generate a jump table;

  else :
  - divide the set of case labels into sub-ranges s.t. each sub-range has density $\geq$ threshold;
  - generate code to use binary search to choose amongst the sub-ranges;
  - handle each sub-range recursively.

Function Calls

- *Caller*:
  - evaluate actual parameters, place them where the callee expects them:
    - param x, k /* x is the k^{th} actual parameter of the call */
  - save appropriate machine state (e.g., return address) and transfer control to the callee:
    - call p
- *Callee*:
  - allocate space for activation record, save callee-saved registers as needed, update stack/frame pointers:
    - enter p

Function Returns

- *Callee*:
  - restore callee-saved registers; place return value (if any) where caller can find it; update stack/frame pointers:
    - retval x;
    - leave p
  - transfer control back to caller:
    - return
- *Caller*:
  - save value returned by callee (if any) into x:
    - retrieve x

Function Call/Return: Example

- *Source*: x = f(0, y+1) + 1;
- *Intermediate Code: Caller*:

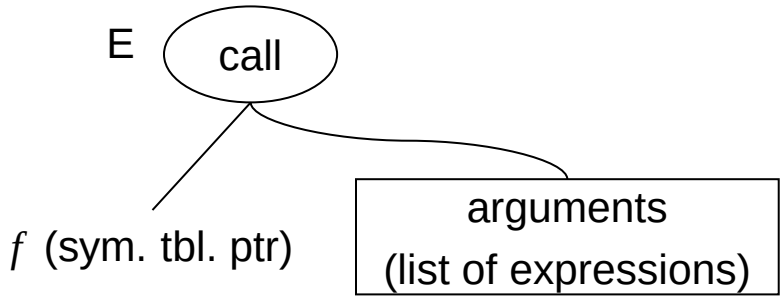
```
t1 = y+1
param t1, 2
param 0, 1
call f
retrieve t2
x = t2+1
```

- *Intermediate Code: Callee*:

```
enter f          /* set up activation record */
…                /* code for f's body */
retval t27       /* return the value of t27 */
leave f          /* clean up activation record */
return
```

Intermediate Code for Function Calls

- non-void return type:



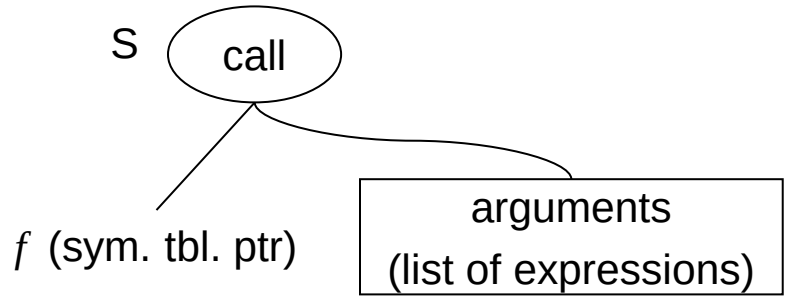
Code Structure:

… evaluate actuals …
param x_k
…
param x_1
} R-to-L
call *f*
retrieve t0 /* t0 a temporary var */

codeGen_expr(E):
/* E.nodetype = FUNCALL */
codeGen_expr_list(arguments);
E.place = newtemp(f.returnType);
E.code = …code to evaluate the arguments…
⊕ param x_k
…
⊕ param x_1
⊕ call f, k
⊕ retrieve E.place;

Intermediate Code for Function Calls

- void return type:



Code Structure:

… evaluate actuals …
param x_k
…
param x_1
call *f*
retrieve t0 /* t0 a temporary var */

R-to-L

```
codeGen_stmt(S):
/* S.nodetype = FUNCALL */
    codeGen_expr_list(arguments);
    E.place = newtemp( f.returnType );
S.code = …code to evaluate the arguments…
        ⊕ param x_k
        …
        ⊕ param x_1
        ⊕ call f, k
        ⊕ retrieve E.place;
```

void return type ⇒ f has no return value
⇒ no need to allocate space for one, or
to retrieve any return value.

Reusing Temporaries

Storage usage can be reduced considerably by reusing space for temporaries:

- For each type T, keep a “free list” of temporaries of type T;
- *newtemp*(T) first checks the appropriate free list to see if it can reuse any temps; allocates new storage if not.
- putting temps on the free list:
  - distinguish between user variables (not freed) and compiler-generated temps (freed);
  - free a temp after the point of its last use (i.e., when its value is no longer needed).