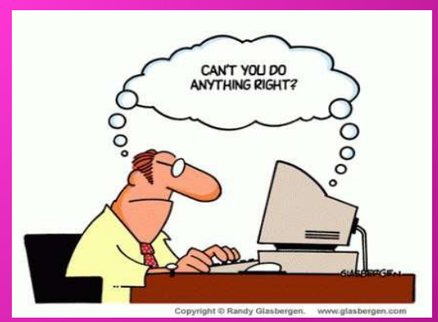
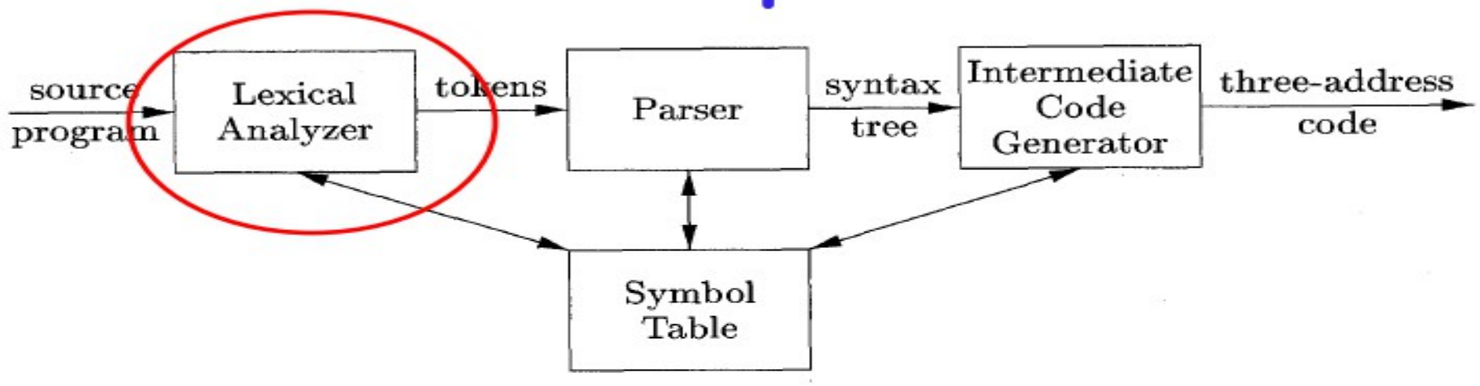
Unit IV


CAN'T YOU DO ANYTHING RIGHT?
Copyright © Randy Glasbergen. www.glasbergen.com

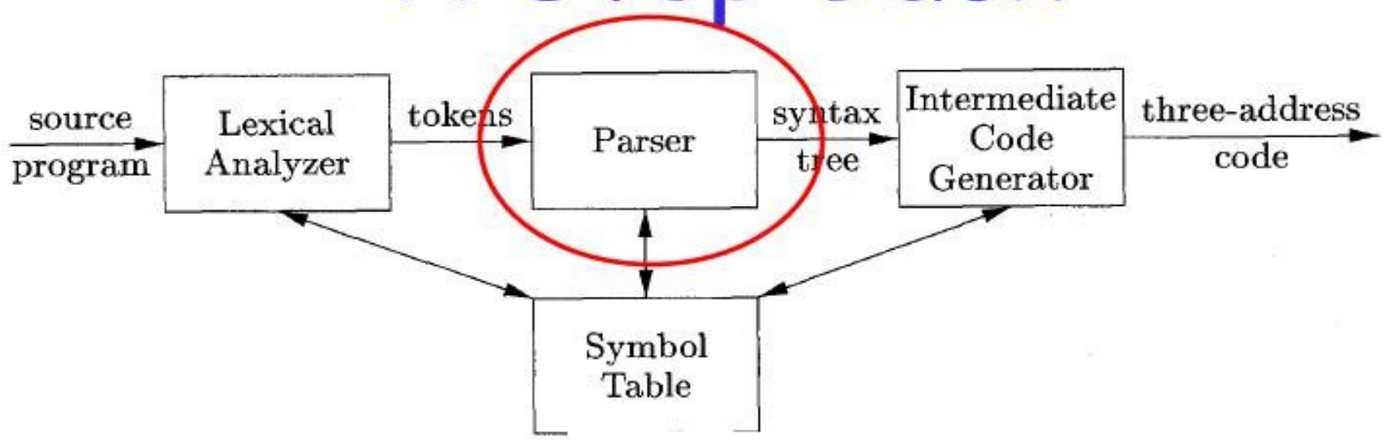
A Step-Back



Chapter 3

- Strings
- Regular expressions
- Tokens
- Transition diagrams
- Finite Automata

A Step-Back

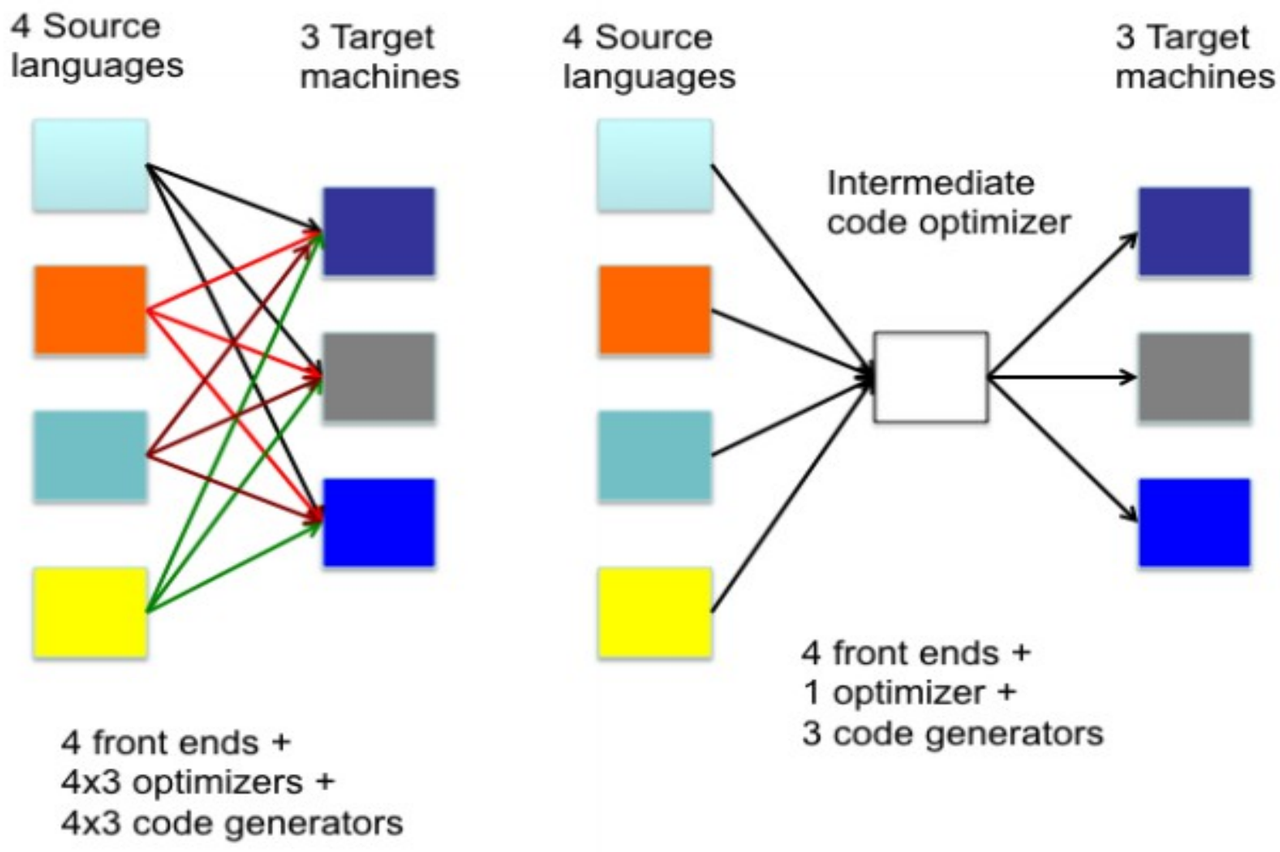


Chapter 4

- Grammars
- Derivations
- Parse-trees
- Top-down parsing (LL)
- Bottom-up paring (LR, SLR,LALR)

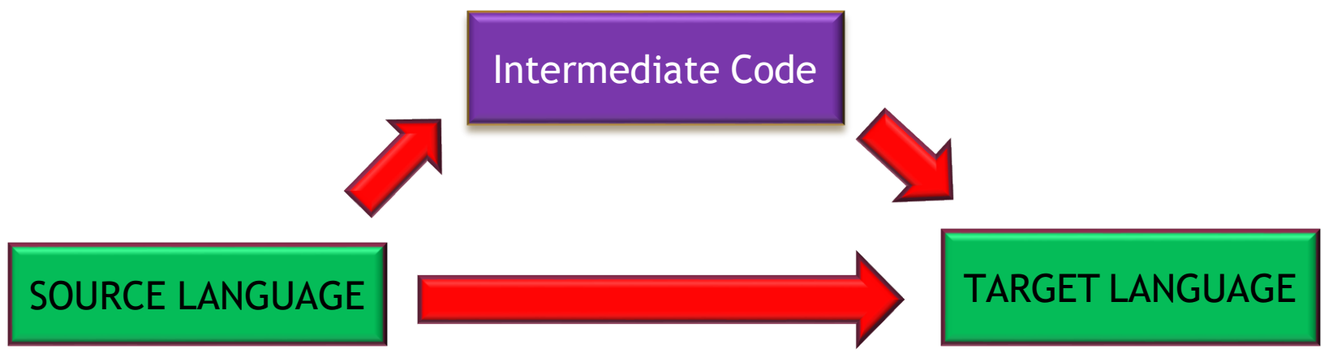
Now What?

Why Intermediate code?



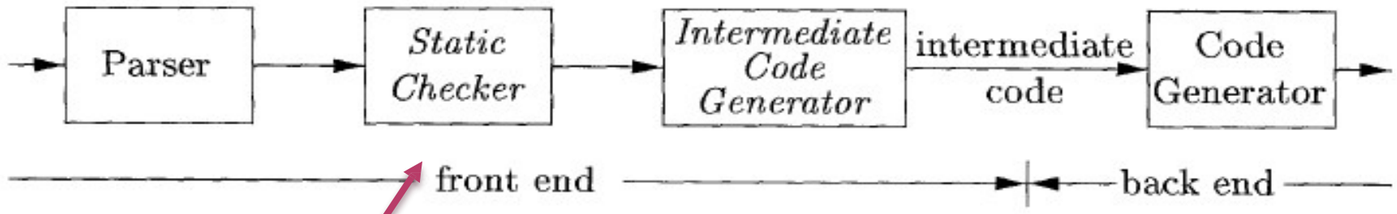
Intermediate Code

- An language b/w source and target language
- Provides an intermediate level of abstraction
  - » More details than the source
  - » Fewer details than the target



Necessity of Intermediate Code

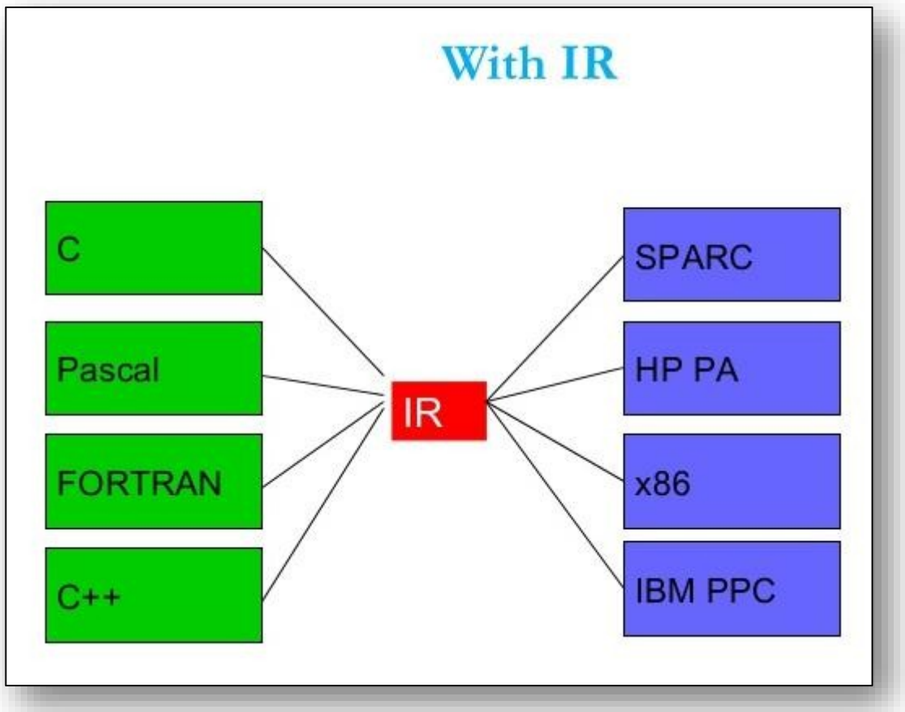
- With Intermediate code, the back end can be changed at any time
- Back end takes intermediate code as input

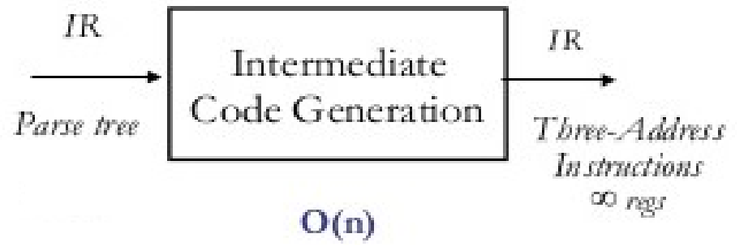


Includes:

• Type checking

• Any syntactic checks that remain after parsing (e.g. ensure *break statement is enclosed within* while-, for-, or switch statements).


With IR
C
Pascal
FORTRAN
C++
IR
SPARC
HP PA
x86
IBM PPC

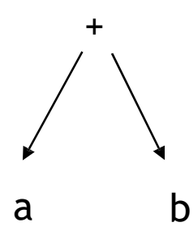

IR
Parse tree
Intermediate Code Generation
IR
Three-Address Instructions
∞ regs
O(n)

Intermediate Languages

- There are 3 types of intermediate representation
  - Syntax Tree
  - DAG
- Post fix notation
- Three address code
  - Quadruples
  - Triples
  - Indirect triples

Syntax Tree

- Represents hierarchical representation of the source program. Syntax tree is the Compressed representation of parse tree
- For example, a + b

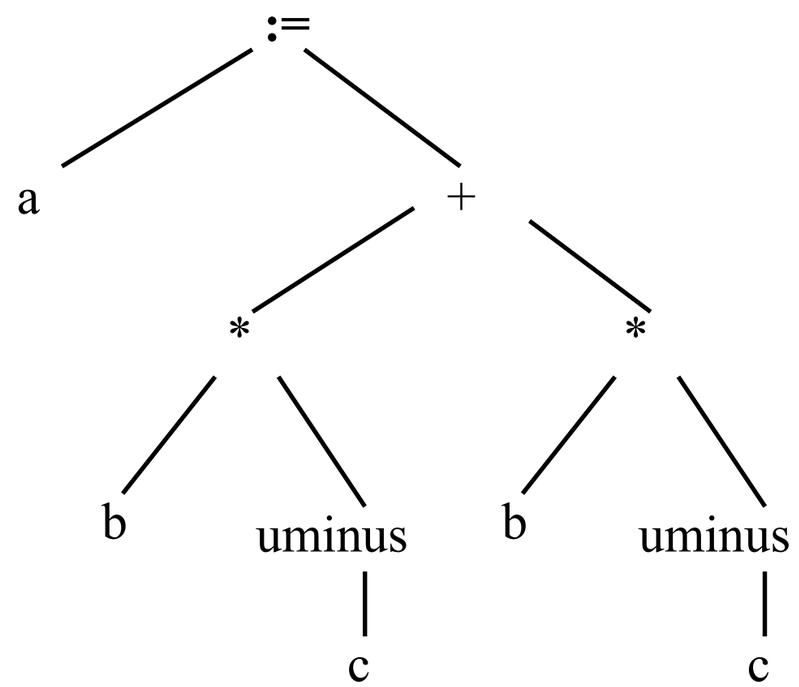


Operators – Interior node

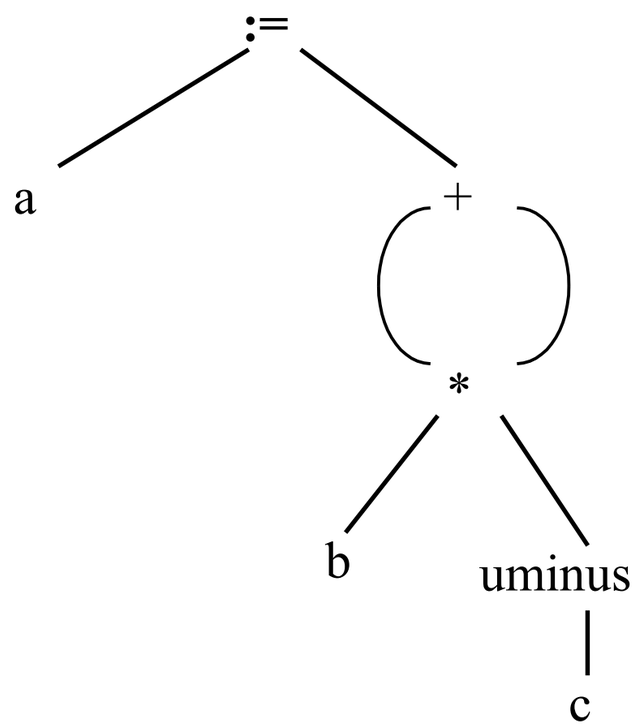
Operands – Leaf node

Intermediate Forms of a Program: Example a := b * -c + b * -c

ASTs: Abstract Syntax Trees



DAGs: Directed Acyclic Graphs
It's a variation of Syntax Tree



Merits of DAG

Directed Acyclic Graph (DAG):
- More compact representation
- Gives clues regarding generation of efficient code

- DAG is created by syntax directed translation.

Functions used for Constructing syntax trees

- Each node is represented as record in syntax tree. The function used is

1 mknode (op, left, right)

- The above function create a operator node with label as operator and extra 2 fields – left and right. For example,

mknode (+, a, b)

Node

+	a	b

- The 2nd function used is

2 mknum (num, value)

- The above function create a number node with label as identifier and a field, which represents the value of the number.

mknum (a, 15)

- The 3rd function used is

3 mkleaf (id, id.place)

- The above function create a leaf node with label as identifier and a pointer to specify the symbol table entry for that identifier.

mkleaf (a, 1006)

Symbol table entry

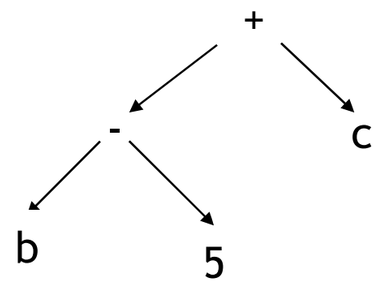
1000	x
1002	Y
1004	Z
1006	a

Syntax tree for assignment statements are produced by the syntax directed definition

Production	Semantic Rule
E → E1 + E2	E.nptr := mknode(‘+’, E1.nptr ,E2.nptr)
E → E1 * E2	E.nptr := mknode(‘* ’, E1.nptr ,E2.nptr)
E → - E1	E.nptr := mkunode(‘uminus’, E1.nptr)
E → (E1)	E.nptr := E1.nptr
E → id	E.nptr := mkleaf(id, id.place)

Creation of DAG - Example

Syntax tree
b – 5 + c



P_1 = mkleaf(id, symbol table entry for b)

P_2 = mknum(num, 5)

P_3 = mknode(‘-’, P_1 , P_2)

P_4 = mkleaf(id, symbol table entry for c)

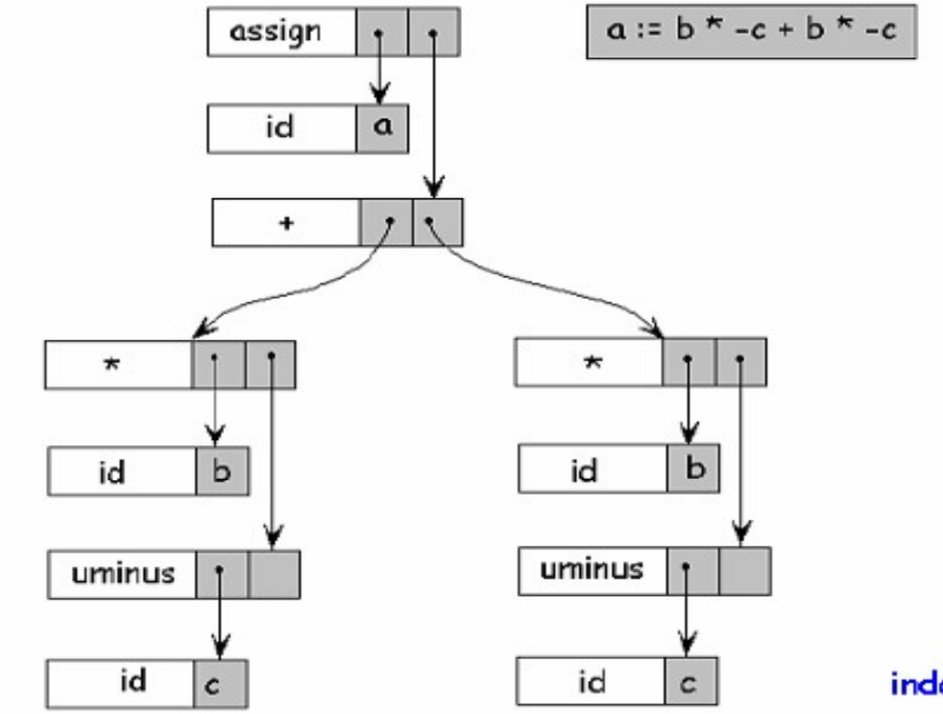
P_5 = mknode(‘=’, P_3 , P_4)

Creation of DAG - Contd

Solve e := a + b * c + d

P_1 = mkleaf(id, symbol table entry for a)
P_2 = mkleaf(id, symbol table entry for b)
P_3 = mknode(‘+’, P_1 , P_2)
P_4 = mkleaf(id, symbol table entry for c)
P_5 = mkleaf(id, symbol table entry for d)
P_6 = mknode(‘+’, P_4 , P_5)
P_7 = mknode(‘*’, P_3 , P_6)
P_8 = mkleaf(id, symbol table entry for 2)
P_9 = mknode(‘=’, P_8 , P_7)

- Representations of the syntax tree :



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

0	id	b	
1	id	c	
2	uminus	1	
3	*	0	2
4	id	b	
5	id	c	
6	uminus	5	
7	*	4	6
8	+	3	7
9	id	a	
10	assign	9	8
11	...		

root

index to node

(a) records (b) array

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

Postfix notation

- It is the linearized representation of Syntax tree in which the root node appears immediately after the children.

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

- POSTFIX notation:

```
a b c uminus * b c uminus  *  +  :=
```

Three address code

- Statements of the form x := y op z → Three address code.
- x,y,z → temporary values | constants | names
- op → arithmetic | logical | binary operator
- Rules:
  - Three address code should have
    - atmost 3 operands + atmost 1 operator + 1 assignment operator

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

t → temporary names

Three address code for Syntax Tree

t_1 := uminus c

t_2 := b * t_1

t_3 := uminus c

t_4 := b * t_3

t_5 := t_2 + t_4

a := t_5

Three address code DAG

t_1 := uminus c

t_2 := b * t_1

t_3 := t_2 + t_2

a := t_3

Types of three-address statements

- **Assignment statement**
  - Binary
    - x := y op z
  - Unary
    - x := op y
- **Copy statement**
  - Simple
    - x := y
  - Indexed:
    - x := y[i] or x[i] := y
  - Address and pointer manipulation
    - x := &y

- **Jump statement**
  - Unconditional
    - goto L
  - Conditional
    - if x relop y goto L1
- **Procedure call**
  - Call procedure P(X1,X2, . . . ,Xn)

Example:

do i = i+1; while (a[i] < v);

```
L:      t1 = i + 1                  100:    t1 = i + 1
        i = t1                      101:    i = t1
        t2 = i * 8                  102:    t2 = i * 8
        t3 = a[t2]                  103:    t3 = a[t2]
        if t3 < v goto L            104:    if t3 < v goto 100
```

Symbolic labels Position numbers

Implementation of three address code

- TAC is implemented by 3 forms
  - **Quadruples**
  - **Triples**
  - **Indirect triples**

Quadruples

- It consists of 4 fields
  - operator
  - arg1
  - arg2
  - Result
- Same as 3 address code general form x := y op z

Quadruple representation

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

- Three address code

t_1 = uminus c
t_2 = b * t_1
t_3 = uminus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

Quadruple representation

	op	arg1	arg2	result
[0]	uminus	c		t_1
[1]	*	b	t_1	t_2
[2]	uminus	c		t_3
[3]	*	b	t_3	t_4
[4]	+	t_2	t_4	t_5
[5]	:=	t_5		a

Triples

- Avoid using more temporary names
- References to instructions are used
  - Has only 3 fields → op, arg1,arg2

t_1 = uminus c
t_2 = b * t_1
t_3 = uminus c
t_4 = b * t_3
t_5 = t_2 + t_4
a = t_5

Triple representation

	op	arg1	arg2
[0]	uminus	c	
[1]	*	b	[0]
[2]	uminus	c	
[3]	*	b	[2]
[4]	+	[1]	[3]
[5]	:=	a	[4]

Indirect Triples

- Use list of pointers to 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

We Need Some Tools

- To help in semantic analysis
- To help in intermediate code generation
- Two such tools
  - Semantic rules (Syntax-Directed Definitions)

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

  - Semantic actions (Syntax Directed Translations)

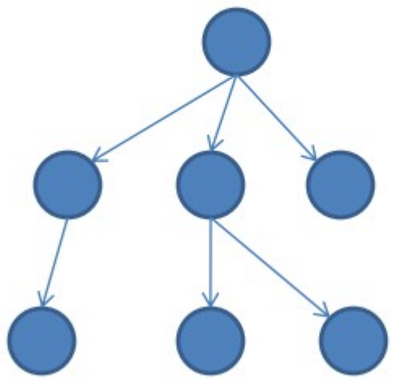
$$E \rightarrow E_1 + T \quad \{ \text{print } '+' \}$$

Syntax-Directed Definitions

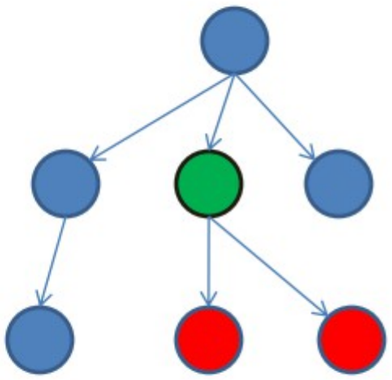
- Context-free grammar
- With attributes and rules to calculate the attributes

PRODUCTION	SEMANTIC RULE
$E \rightarrow E_1 + T$	$E.code = E_1.code \parallel T.code \parallel '+'$

Two Types of Attributes



Two Types of Attributes

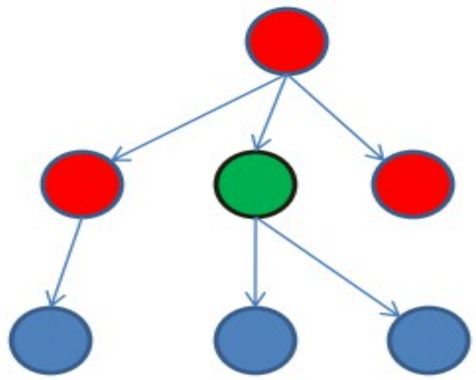


SDD involving only synthesized attributes is called ***S-attributed***

Synthesized Attributes

Attribute of the node is defined in terms of:
- Attribute values at children of the node
- Attribute value at node itself

Two Types of Attributes



Inherited Attributes

Attribute of the node is defined in terms of:
- Attribute values at parent of the node
- Attribute values at siblings
- Attribute value at node itself

SYNTAX-DIRECTED DEFINITION

- When we associate semantic rules with productions then we use the term SYNTAX-DIRECTED DEFINITION.

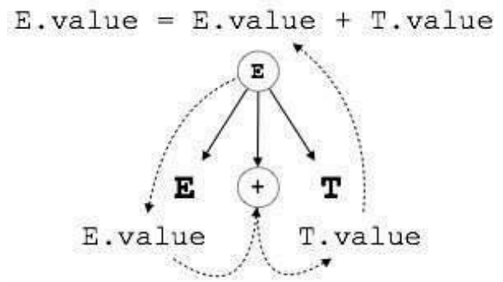
Definition

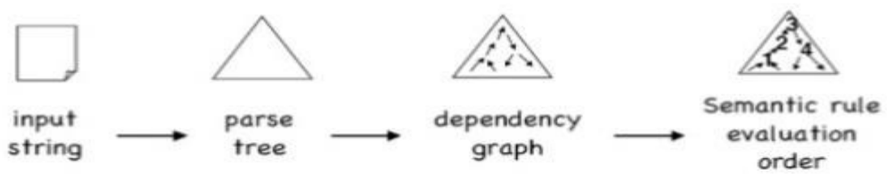
- A SYNTAX-DIRECTED DEFINITION is a context-free grammar in which
  - each grammar symbol X is associated with two finite sets of values: the **synthesized attributes of X** and the **inherited attributes of X**.
- Attributes are
  - Numbers
  - Strings
  - Memory locations,
  - A data type etc..

SDD Contd..

- Each production rule is associated with semantic rules.
- Semantic rules set up dependencies between attributes which can be represented by a **Dependency Graph**.

- **Dependency graph** → specifies the evaluation order of the semantic rules.





Synthesized attribute Vs Inherited attribute

- **Synthesized attribute:** get values from the attribute values of their child nodes
  - S → ABC
  - If S is taking values from its child nodes (A,B,C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.
  - (E → E + T), the parent node E gets its value from its child node.

- **Inherited attribute:** take values from parent and/or siblings.
  - S → ABC
  - A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Synthesized attribute Vs Inherited attribute

- The attribute which takes data values from its child nodes is called a **synthesized attribute.**
- The attribute which takes values from parents or sibling nodes is called **inherited attributes.**

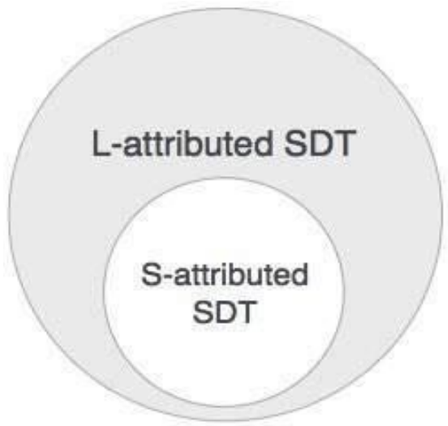
Synthesized Versus Inherited Attributes (cont'd)

Production	Semantic Rule
$D \rightarrow T\ L$	$L.\text{in} := T.\text{type}$
$T \rightarrow \textbf{int}$	$T.\text{type} :=$ 'integer'
…	…
$L \rightarrow \textbf{id}$	… $:= L.\text{in}$

inherited

synthesized

S-attributed SDT Vs L-attributed SDT



- We may conclude that if a definition is S-attributed, then it is also L-attributed as L-attributed definition encloses S-attributed definitions.

S-attributed SDT Vs L-attributed SDT

S - attributed SDT	L - attributed SDT
Uses only synthesized attributes	Uses both inherited and synthesized attributes. Each inherited attributes is restricted to inherit either from parents or left siblings only A -> XYZ {Y.S = A.S, Y.S = X.S, **Y.S ≠ Z.S**}
Semantic actions are placed on the right end of the production A → BCC{ } Hence, postfix SDT	Semantic actions are placed anywhere A → { }BC D{ }E FG{ }
Attributes are evaluated using bottom up parsing	Attributes are evaluated by traversing depth-first, left-to-right

SDD of simple Desk Calculator

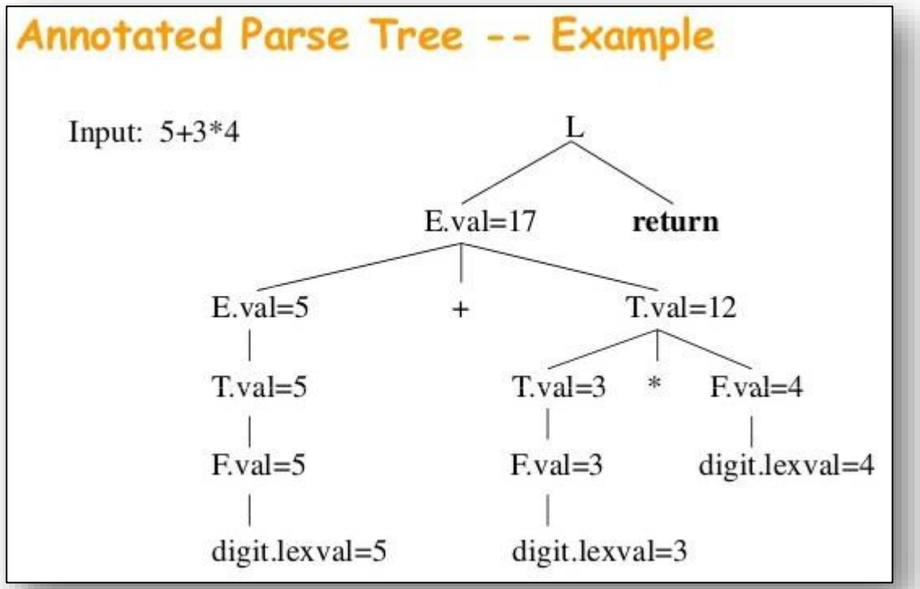
Production	Semantic Rules
1) $L \rightarrow E$ **n**	*L.val* = *E.val*
2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
3) $E \rightarrow T$	*E.val* = *T.val*
4) $T \rightarrow T_1 * F$	$T.val = T_1.val \times F.val$
5) $T \rightarrow F$	*T.val* = *F.val*
6) $F \rightarrow (E)$	*F.val* = *E.val*
7) $F \rightarrow$ **digit**	*F.val* = **digit**.*lexval*

Fig 5.1: Syntax Directed Definition of simple desk calculator

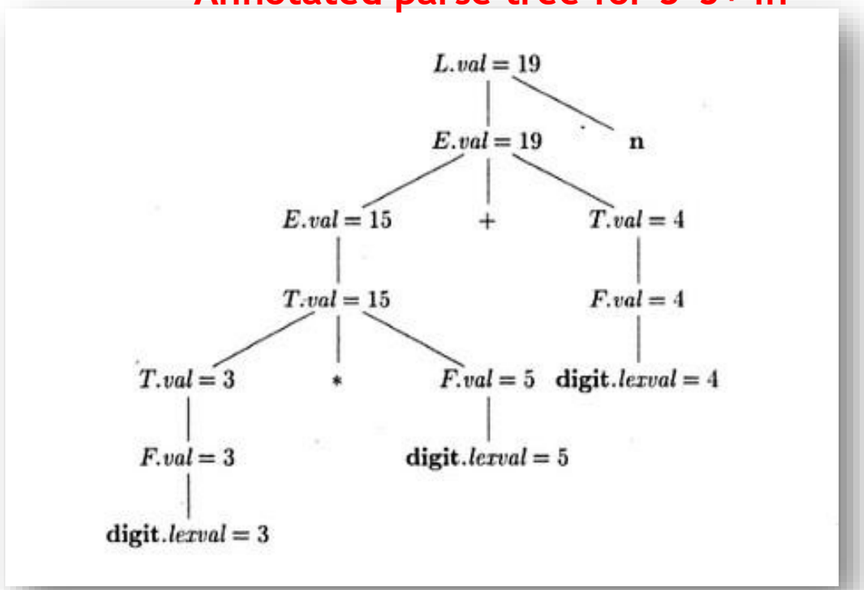
- E,T,F → Synthesized attribute val.
- The token **digit** has a synthesized attribute *lexval*.

ANNOTATED PARSE TREE

- An ANNOTATED PARSE TREE is a parse tree showing the values of the attributes at each node. The process of computing the attribute values at the nodes is called **annotating or decorating** the parse tree.



Annotated parse tree for 3*5+4n



Syntax-directed generation of 3AC

- Idea: expressions get two attributes:
  - **E.place:** a name to hold the value of E
    - id.place (like id =19)
    - x + y → x.place + y.place
  - **E.code**: the expressions such as $E_1 = E_2 + E_3$

- The function **newtemp()** returns a fresh temporary name

- A function **gen(…)** to produce sequence of three address statements
  - The statements themselves are kept in some data structure, e.g. list
  - SDD operations described using pseudo code

Syntax-directed translation of three adddress code for ASSIGNMENT statements and expressions

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

gen(x ':=' y '+' z) represents x := y + z

Example problem : x := (y + z) * (-w + v)

- Three address Code

t1 = y +z
t2 = -w
t3 = t2 +v
t4 = t1 * t3
x = t4

Syntax directed translation for the above expression

1. E→id | E.place := y
2. E→id | E.place := z
3. E→E + E | E.place := t1 E.Code := gen(t1 = y + z)
4. E→(E) | E.place := t1 E.Code := gen(t1 = y + z)
5. E→id | E.place := w
6. E→ -E | E.place := t2 E.Code := gen(t2 = -w)
7. E→id | E.place := v
8. E→E + E | E.place := t3 E.Code := gen(t2 = -w , t3 = t2 +v)
9. E →(E) | E.place := t3 E.Code := gen(t2 = -w , t3 = t2 + v)
10. E→E * E | E.place = t4 E.Code := gen(t1= y + z , t2 = -w , t3 = t2 + v , t4 = t1 * t3)
11. S→id:=E | S.Code := gen(t1= y + z , t2 = -w , t3 = t2 + v , t4 = t1 * t3 , x=t4)

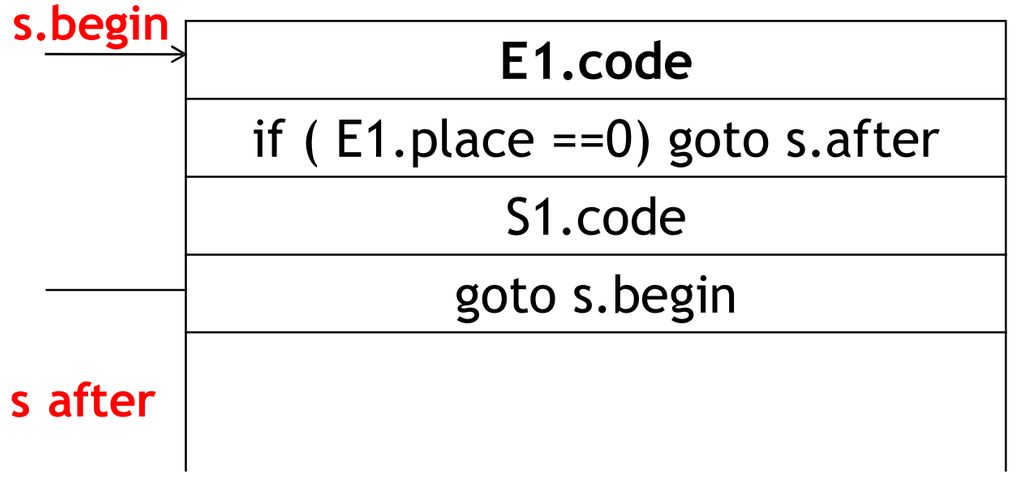
Basic Concept Invloved

- The basic idea of converting any flow of control statement to a three address code is to simulate the “branching” of the flow of control using the goto statement.
- This is done by skipping to different parts of the code (label) to mimic the different flow of control branches.

Syntax-directed translation of three adddress code for CONTROL statements

S →While E_1 do S_1

s. begin → ptr to the beginning of the statement
s.after → statement which follows the code s



SDD for Control statements

- With respect to the previous syntax-directed translation, we have added two new synthesized attributes S.begin and S.after
- When the value of E becomes zero, control leaves the while statement

Production	Semantic Rules
S -> while E do S1	S.begin := newlabel(); S.after := newlabel(); code1 := generate(S.begin ':') | | E.code | | code2 := generate('if ' E.place '= 0 ' 'goto' S.after) | | S1.code code3 := generate('goto' S.begin) | | generate(S.after) S.code := code1 | | code2 | | code3 (or) S.code := gen(S.begin ':') || E.code || gen('if' E.place '=' '0' 'goto' S.after) || S1.code || gen('goto' S.begin) || gen(S.after ':')

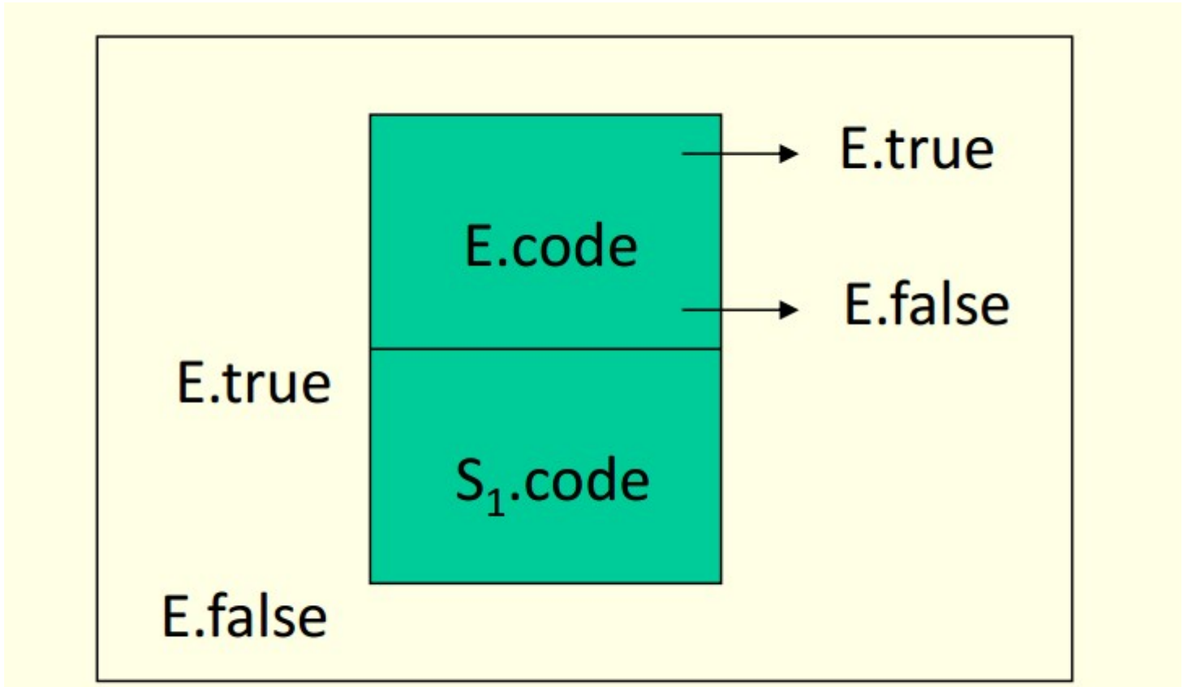
Flow-of-Control Statements

- Consider grammar

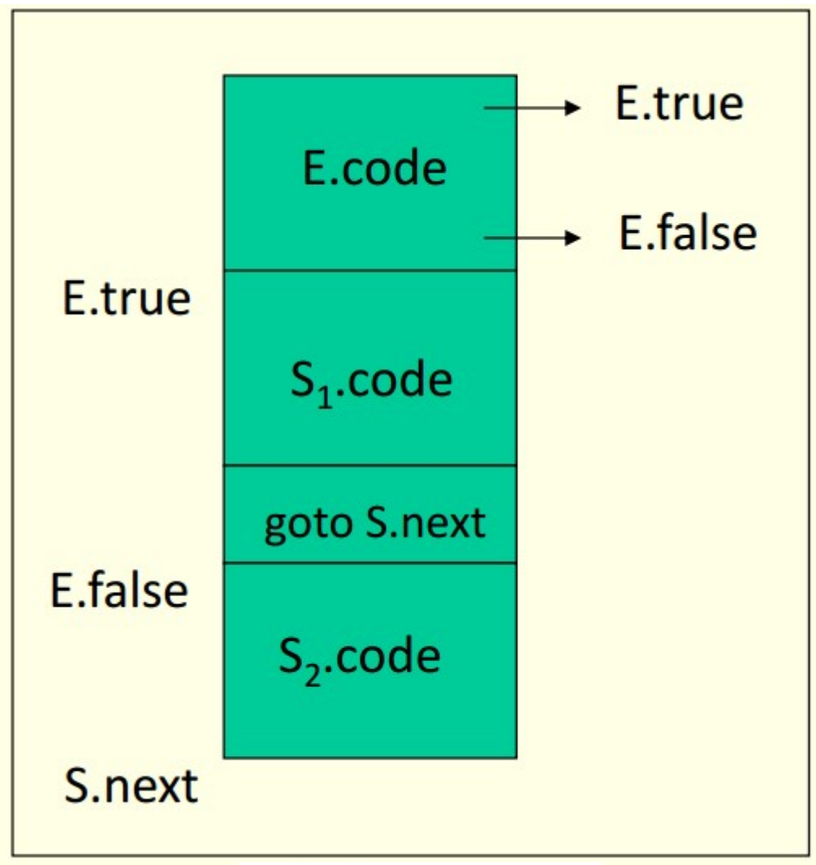
$$S \rightarrow \textbf{if}\ E\ \textbf{then}\ S_1 \mid \textbf{if}\ E\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \mid \textbf{while}\ E\ \textbf{do}\ S_1$$

- Each boolean expression E gets two label attributes
  - *true* : where to go when expression evaluates to **true**
  - *false* : where to go when expression evaluates to **false**
- Inherited attribute *S.next* is address of next instruction after S
- Inherited attribute *S.begin* is address of first instruction of S

If and If - else structure

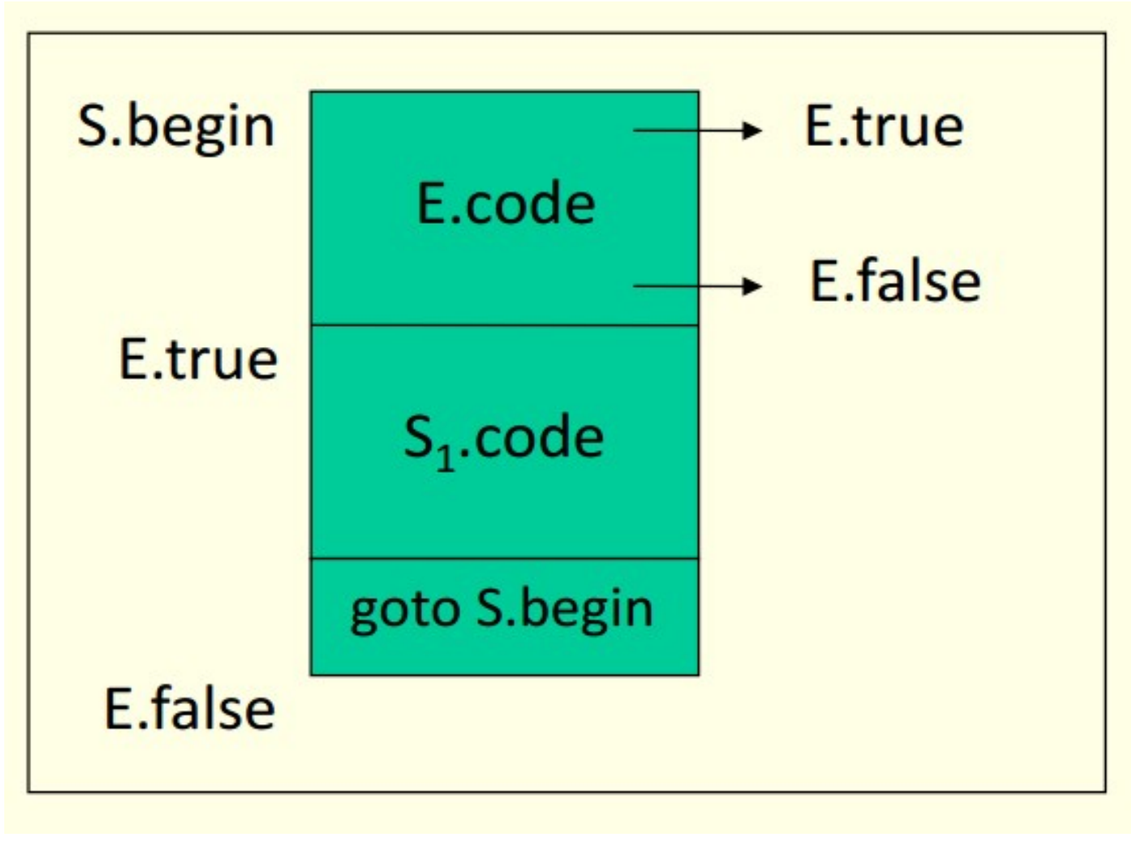


if-then



if-then-else

While - do



Flow-of-Control Statements

$S \rightarrow$ **if** E **then** S_1 { $E.true := newlabel$;
$E.false := S.next$;
$S_1.next := S.next$;
$S.code := E.code \,||\, gen(E.true\ ':')\ S_1.code$ }

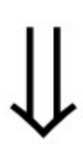
$S \rightarrow$ **if** E **then** S_1 **else** S_2 { $E.true := newlabel$;
$E.false := newlabel$;
$S_1.next := S.next$;
$S_2.next := S.next$;
$S.code := E.code \,||$
$gen(E.true\ ':')\ S_1.code \,||$
$gen(\texttt{'goto'}\ S.next) \,||$
$gen(E.false\ ':')\ S_2.code$ }

Flow-of-Control Statements

$S \rightarrow$ **while** E **do** S_1 { $S.begin := newlabel$;
$E.true := newlabel$;
$E.false := S.next$;
$S_1.next := S.begin$;
$S.code :=$ $gen(S.begin\ ':')$ || $E.code$ ||
$gen(E.true\ ':')$ || $S_1.code$ ||
$gen(\text{'goto'}\ S.begin)$ }

Control-Flow Translation of Boolean Expressions

- Now we can see how to implement the flow-of-control version of boolean expression translation.
- *E.g.,* `a < b or c < d and e < f`



```
    if a < b goto Ltrue // whole thing true
    goto L1
L1: if c < d goto L2
    goto Lfalse         // whole thing false
L2: if e < f goto Ltrue
    goto Lfalse
```

Code for a < b or c < d and e < f

100: if a < b goto 103
101: $t_1 = 0$
102: goto 104
103: $t_1 = 1$
104:
if c < d goto 107
105: $t_2 = 0$
106: goto 108
107: $t_2 = 1$
108:
if e < f goto 111
109: $t_3 = 0$
110: goto 112
111: $t_3 = 1$
112:
$t_4 = t_2$ and t_3
113: $t_5 = t_1$ or t_4

Control-Flow Translation

$E \rightarrow E_1$ **or** E_2 { $E_1.true := E.true$;
$E_1.false := newlabel$
$E_2.true := E.true$;
$E_2.false := E.false$;
$E.code := E_1.code \parallel gen(E_1.false\ ':') \parallel E_2.code$ }

$E \rightarrow E_1$ **and** E_2 { $E_1.true := newlabel$;
$E_1.false := E.false$
$E_2.true := E.true$;
$E_2.false := E.false$;
$E.code := E_1.code \parallel gen(E_1.true\ ':') \parallel E_2.code$ }

Control-Flow Translation

$E \rightarrow$ **not** E_1 $\quad$ { $E_1.true := E.false$;
$E_1.false := E.true$
$E.code := E_1.code$; }

$E \rightarrow$ **(** E_1 **)** $\quad$ {$E_1.true := E.true$;
$E_1.false := E.false$
$E.code := E_1.code$;}

$E \rightarrow$ **true** $\quad$ { $E.code :=$ *gen* ('**goto**') || $E.true$ }

$E \rightarrow$ **false** $\quad$ { $E.code :=$ *gen* ('**goto**') || $E.false$}

Control-Flow Translation

$E \rightarrow$ **id**$_1$ **relop** **id**$_2$ { *gen* (**'if'** **id**$_1$.*place* **relop**.*op* **id**$_2$.*place* **'goto'** *E.true*) ||
gen (**'goto'** *E.false*) }

Switch statement

```
switch ( E ) {
        case V1: S1
        case V2: S2
        ...
        case Vn−1: Sn−1
        default: Sn
}
```

Figure 6.48: Switch-statement syntax

```
        code to evaluate E into t
        goto test
L1:     code for S1
        goto next
L2:     code for S2
        goto next
        ...
Ln-1:   code for Sn-1
        goto next
Ln:     code for Sn
        goto next
test:   if t = V1 goto L1
        if t = V2 goto L2
        ...
        if t = Vn-1 goto Ln-1
        goto Ln
next:
```

Figure 6.49: Translation of a switch-statement

Example 2:

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

Contd…

```
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:     t1 = Y - Z
        X= t1
        goto Ll
Lnext:
```

if (x < 5 || x > 10 && x == y) x = 3 ;

```
        if x < 5 goto L2
        goto L3
L3:     if x > 10 goto L4
        goto L1
L4:     if x == y goto L2
        goto L1
L2:     x = 3
L1:
```

Example 2

```
int a[10], b[10], dot_prod, i;
dot_prod = 0;
for (i=0; i<10; i++) dot_prod += a[i]*b[i];
```

Intermediate code

```
    dot_prod = 0;          |     T6 = T4[T5]
    i = 0;                 |     T7 = T3*T6
L1: if(i >= 10)goto L2     |     T8 = dot_prod+T7
    T1 = addr(a)           |     dot_prod = T8
    T2 = i*4               |     T9 = i+1
    T3 = T1[T2]            |     i = T9
    T4 = addr(b)           |     goto L1
    T5 = i*4               |L2:
```

USC Viterbi School of Engineering | CSCI 565 - Compiler Design | Spring 2010

Loop Example

Source Code	Three Address Instructions
do	L: t1 = i + 1
i = i + 1;	i = t1
while (a[i] < v);	t2 = i * 8
	t3 = a[t2]
	if t3 < v goto L

Pedro Diniz pedro@isi.edu

USC Viterbi School of Engineering | CSCI 565 - Compiler Design | Spring 2010

Loop Example

Source Code	Three Address Instructions
do	L: t1 = i + 1
i = i + 1;	i = t1
while (a[i] < v);	**t2 = i * 8**
	t3 = a[t2]
	if t3 < v goto L

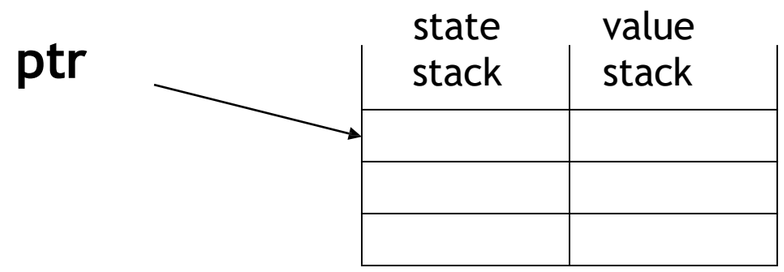
Where did this come from ?

Pedro Diniz pedro@isi.edu

Bottom-up Evaluation of *S*-attributed definitions

Bottom-up evaluation of S-attributed definitions

- Evaluated by a bottom-up parser as the input is being parsed.
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also & the current top of stack is indicated by ptr top



Contd…

- Suppose semantic rule A.a = f(X.x, Y.y, Z.z) is associated with production

  A → XYZ

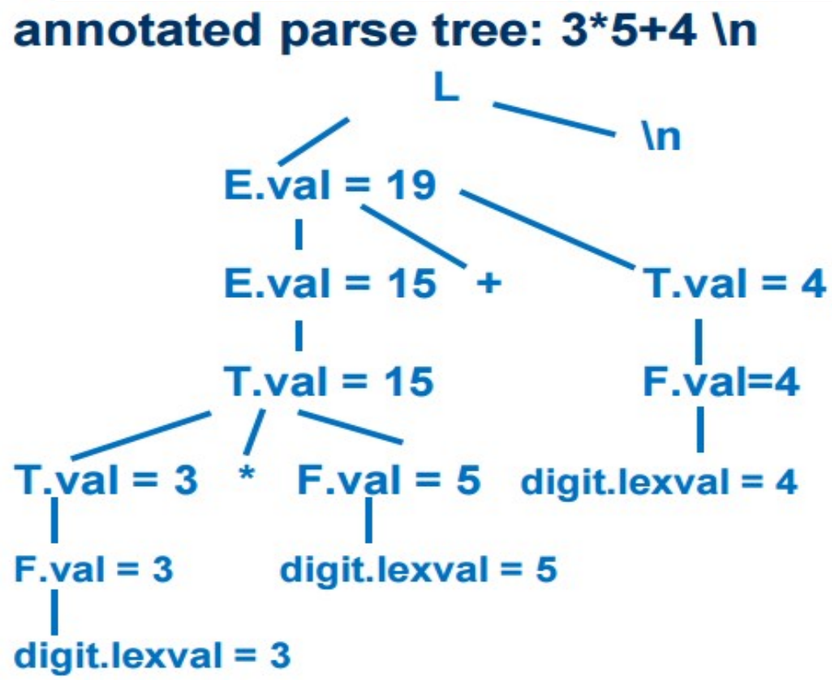
- Before reducing XYZ to A, value of Z is in val(top), value of Y is in val(top-1) and value of X is in val(top-2)
- If symbol has no attribute then the entry is undefined
- After the reduction, top is decremented by 2 and state covering A is put in val(top)

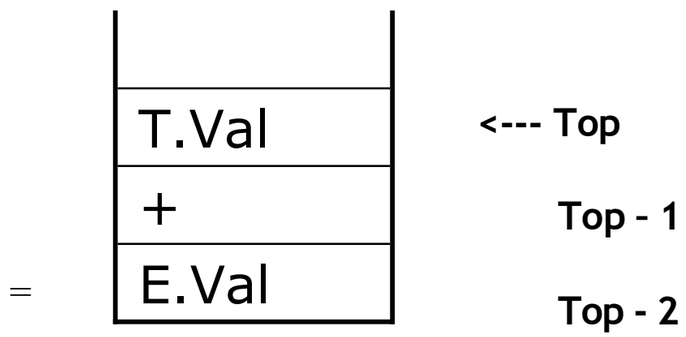
Example: desk calculator

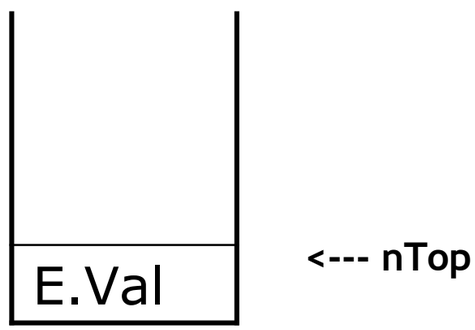
L → En	print(val(top))
E → E + T	val(ntop) = val(top-2) + val(top)
E → T	
T → T * F	val(ntop) = val(top-2) * val(top)
T → F	
F → (E)	val(ntop) = val(top-1)
F → digit	

Before reduction , **ntop = top - r +1** (r is # symbols reduced on RHS)
After code reduction**, top = ntop**

- Using LR Parser: Parse 3 * 5 + 4 $







Input	Stack	Attribute	Production Used
3 * 5 + 4 $	-	-	
* 5 + 4 $	3	3	
* 5 + 4 $	F	3	F ---> digit
* 5 + 4 $	T	3	T ---> F
5 + 4 $	T *	3	
+ 4 $	T * 5	3 * 5	
+ 4 $	T * F	3 * 5	F -→ digit
+ 4 $	T	15	T ---> T * F
+ 4 $	E	15	E ---> T
4 $	E +	15	
$	E + 4	15 + 4	
$	E + F	15 + 4	F ---> digit
$	E + T	15 4	T ---> F
$	E	19	E ---> E + T
	E	19	
	L	19	L ---> E $

DESIGN OF THE PREDICTIVE TRANSLATOR

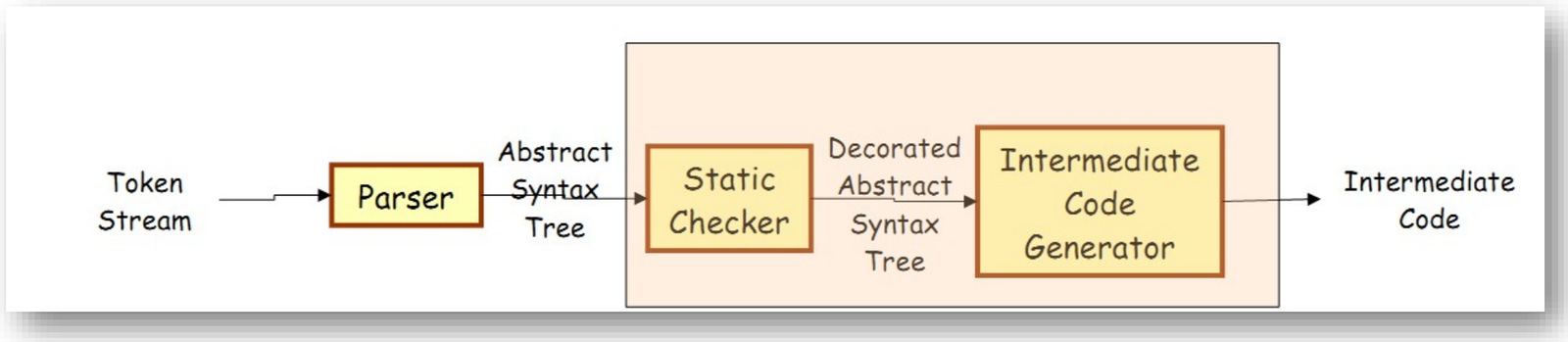
Input: syntax-directed translation scheme; **Outp.:** Syntax-directed translator

1. For each nonterminal A, construct a function that has a formal parameter for each inherited attribute of A and that returns the values of the synthesized attributes of A. This function has a local variable for each attribute of each grammar symbol that appears in a production for A.
2. As previously described (see predictive parsing), the code for nonterminal A decides what production to use based on the current input symbol.
3. The code for each production does the following (evaluation from left to right):
   1. Token X with synthesized attribute x: Save the value of x in a variable $X.x$. Generate a call to match token X.
   2. Nonterminal B: Generate $c := B(b_1, \ldots, b_k)$; $b_1, \ldots, b_k$ variables for inherited attributes of B; c variable for synthesized attribute of B.
   3. For an action, copy the code into the parser, replacing each reference to an attribute by the variable for that attribute.

Type Checking

Type Checking?

- Type checking is the process of verifying that each operation executed in a program respects the system language.
  - Checking all operands are of appropriate data types and values
- **Semantic analysis phase == Type checking.**



Type Check

- Type checks: A compiler should report an error if an operator is applied to an incompatible operand.
- *Type Checking maybe be either static or dynamic*.
  - Compile time checking is static.
  - Pascal and C type checking is static
  - Static type checking is also useful to determine the amount of memory needed to store variables

Examples of static check

- Type checks (incompatible operand)

```
int a, b[10], c;
...
a = b + c;
```

- Flow-of-control checks (break statement)

```
main {
    int I
    ....
    I++;
    break;
}
```

- Uniqueness checks

```
main() {
    int i, j;
    double i, j;
....
}
```

- Defined before use

```
main() {
    int i;
    i1 = 0;
....
}
```

- Name-related checks

```
LOOPA:
   LOOP
       EXIT WHEN I=N
       I=I+1;
END LOOP LOOPB;
```

Type Checking Contd..

- **Dynamic Type Checking**
  - Performed at run time
  - More flexible, rapid prototyping
  - Overhead to check run time errors.

- Array-bound checking in java:

```
a[i] = 0;
```

Rules for Type Checking

- **Type synthesis**
  - Builds the type of the expression from the types of its sub expressions
  - Requires name to be declared before usage
  - Examples

string	string		int	string
"drive"	+ "drink"		42 +	"the answer"
string			ERROR	

- **Type inference**
  - Determines the type of the construct from the way its used.

Type systems

- The design of a type checker for a language is based on the information about the syntactic constructs in the language, the notations and language for the language constructs.

Specifications of a simple type checker

- The type of each identifier must be declared before the identifier is used.
  - The type checker is a translation scheme that synthesizes the type of each expression from the type of its sub-expressions.

– $P \rightarrow D;E$
– $D \rightarrow D;D$
– $D \rightarrow$ id:T {addtype(id.entry, T.type)}
– $T \rightarrow$char {T.type=char}
– $T \rightarrow$ integer {T.type=integer}
– $T \rightarrow$ array [num] of T {T.type=array(1..num,T_1.type)}
– $T \rightarrow$ ^T_1 {T.type=pointer(T_1.type)}

Contd…

- Procedure P consists of declarations D where declaration may be nested or declaration of variable with its type T.
- The type of the variable may be int, char, real, pointer or array of records.
- Whenever an identifier is declared its added to symbol table entry with its attributes & types.

Type checking of Expressions

$E \rightarrow$ **id**	{ E.type:=lookup(id.entry) }
$E \rightarrow$ **literal**	{ E.type:=char }
$E \rightarrow$ **int**	{ E.type:=int }
$E \rightarrow$ **real**	{ E.type:=real }
$E \rightarrow E_1$ **mod** E_2	{ if (E_1.type=int and E_2.type=int) then E.type:=int else E.type:=type-error }
$E \rightarrow E_1 [E_2]$	{ if (E_2.type=int and E_1.type=array(s,t)) then E.type:=t else E.type:=type-error }
$E \rightarrow E_1 \uparrow$	{ if (E_1.type=pointer(t)) then E.type:=t else E.type:=type-error }

- Rule (1) signifies that the identifier saved in the symbol table is fetched and assigned to E.type
- Rule (2) and (3) signifies that the tokens literal & int have the data type ‘char’ and ‘integer’ respectively.
- Rule (5) say that the two sub expressions E1 & E2 should be integers else return error.
- Rule (7) says that the postfix operator yields the object pointed by the operand.

Type checking of Statements

S -> id := E

{if id.type == E.type then S.type = void;

else S.type = error;}

S -> if E then S1

{if E.type == boolean then S.type = S1.type;

else S.type = error;}

S ->while E do S1

{ if E.type == boolean then S.type = S1.type;

else S.type = error;}

S->S1;S2

{if S1.type == void and S2.type == void then

S.type = void;

else S.type = type_error;

}

Type checking of Functions

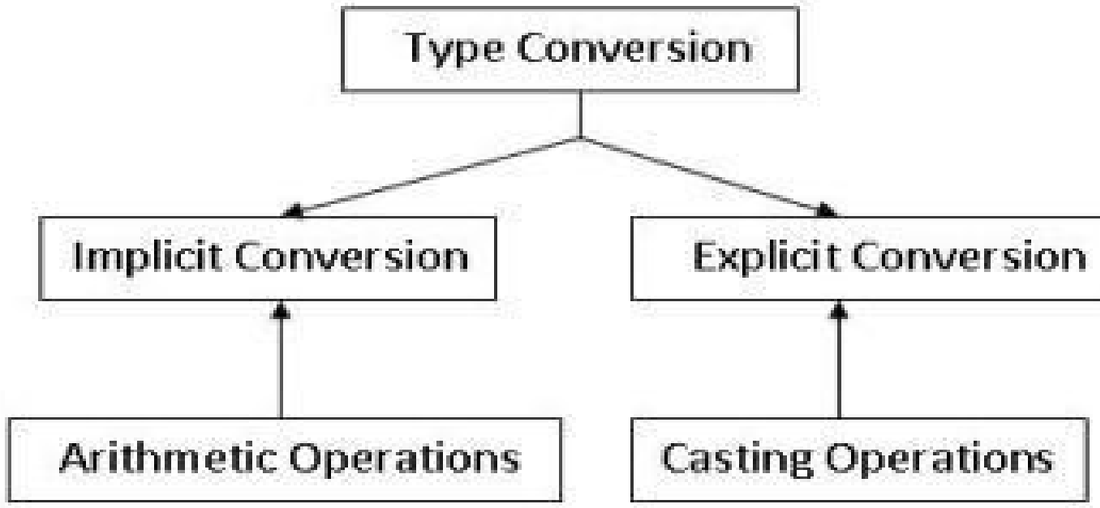
```
T->T1->T2 /* function declaration */
{T.type = T1.type ->T2.type}
```

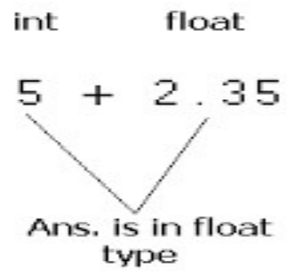
```
E->E1(E2)  /* function call */
{if E1.type == t1.type->t2.type && E2.type == t1.type then
    T.type = t2.type;
 else T.type - error;
}
```

Algorithm for structural Equivalence

- Its necessary to check whether two type expressions are equivalent or not. the type expressions are built from basic types and constructors. If two type expressions are integer, then the resultant operand's type is also an integer.

```
Algorithm 7.1 Structural Equivalence
    function sequiv (Type : s, t) : boolean
    begin
        if s and t are the same basic type
            return (TRUE);
        else if (s = array(s1, s2)) and (t = array(t1, t2))
            return (sequiv(s1, t1) and sequiv(s2, t2))
        else if (s = s1 × s2) and (t = t1 × t2)
            return (sequiv(s1, t1) and sequiv(s2, t2))
        else if (s = pointer(s1)) and (t = pointer(t1))
            return (sequiv(s1, t1))
        else if (s = s1 → s2) and (t = t1 → t2)
            return (sequiv(s1, t1) and sequiv(s2, t2))
        else
            return (FALSE);
        end if
    end                                              { sequiv }
```






```
double d   = 10;
int i;
i = (int) d
```

Type Cast Operator

Type Conversions

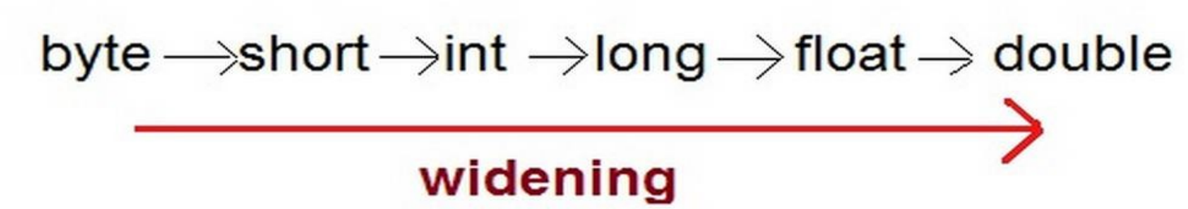
- **Implicit type conversions**
  - In an expression such as f + i where f is a float and i is an integer, a compiler must first convert the integer to a float before the floating point addition operation is performed.

- **Explicit type conversions**
  - **In C**, explicit type conversions can be forced ("coerced") in an expression using a unary operator called a **cast**.
    - E.g., sqrt((double) n) converts the value of the integer n to a double before passing it on to the square root routine sqrt.

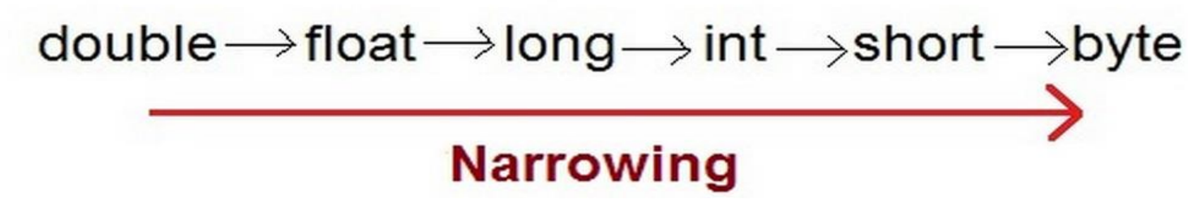
Rules for Type conversions

In Java, type casting is classified into two types,

- Widening Casting(Implicit)



- Narrowing Casting(Explicitly done)



Runtime Environments

Run-Time Environments

- At run time, we need a system to map NAMES (in the source program) to STORAGE on the machine.

- Allocation and de-allocation of memory is handled by a **RUN-TIME SUPPORT SYSTEM.**

- One of the primary responsibilities of the run-time system is to manage **ACTIVATIONS** of procedures.

Source language issues

- Procedures
- Activation trees
- Control stack
- Scope of the Definition

Procedures

- **PROCEDURE DEFINTION** is associated with the identifier and the statement (a statement could actually be a block of statements).
  - The identifier is called the **PROCEDURE NAME**.
  - The statement is called the **PROCEDURE BODY**.

- A **PROCEDURE CALL** is an invocation of a procedure within an executable statement.

Syntax

```
Procedure Procedure_name
{
        ....
    }
```

Example

```
Procedure add
{
        ....
    }
```

add → identifier

Parameter Passing

actual parameters
(arguments)

```
int x = plus(1,2);
```

function call

formal parameters

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

Function body

- How are parameters passed?

Types of Parameter Passing

- Call by value
- Call be reference
- Call by copy/restore
- Call by name

Call by value

The **call by value method** copies the actual value of an argument into the formal parameters. In this case, changes made to the parameters inside the ***function have no effect*** on the argument.

```
 Void main ()
{
   int a = 100;
   int b = 200;

   cout << "Before swap, value of a :" << a ;
   cout << "Before swap, value of b :" << b ;

    swap(a, b);
   // calling a function to swap the values.

   cout << "After swap, value of a :" << a ;
   cout << "After swap, value of b :" << b ;
}
```

Function Definition

```
void swap(int x, int y)
{
   int temp;

   temp = x;
    x = y;
   y = temp;
   return;
}
```

OUTPUT

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :100
After swap, value of b :200
```

Call by Reference

In **call by reference method**, the address of the variable is passed to the function as parameter. Hence, the value of the actual parameter is modified by formal parameter.

```
 Void main ()
{
  int a = 100;
  int b = 200;

  cout << "Before swap, value of a :" << a ;
  cout << "Before swap, value of b :" << b ;

   swap(a, b);
  // calling a function to swap the values.

  cout << "After swap, value of a :" << a ;
  cout << "After swap, value of b :" << b ;
}
```

Function Definition

```
void swap(int  &x, int  &y)
{
/* &x and &y are reference variables */

  int  temp;
  temp = x;
  x = y;
  y = temp;
  return;
}
```

OUTPUT

```
Before swap, value of a :100
Before swap, value of b :200
After swap, value of a :200
After swap, value of b :100
```

Call by copy/restore

- This method is the hybrid value between call by value and call by reference. This method is also copy –in-copy-out method.

- The calling procedure calculates the value of the actual parameter and it is copied to activation record of the called procedure. During execution of the called procedure, the actual parameters is not affected.

- If there are any changes by the formal parameters, then it is copied to the actual parameters.

Difference between call by reference and copy/restore:
Changes made to the function variable → Not shown until after the end of the function
But, in call by reference changes will be seen immediately.

Call by name/macro expansion

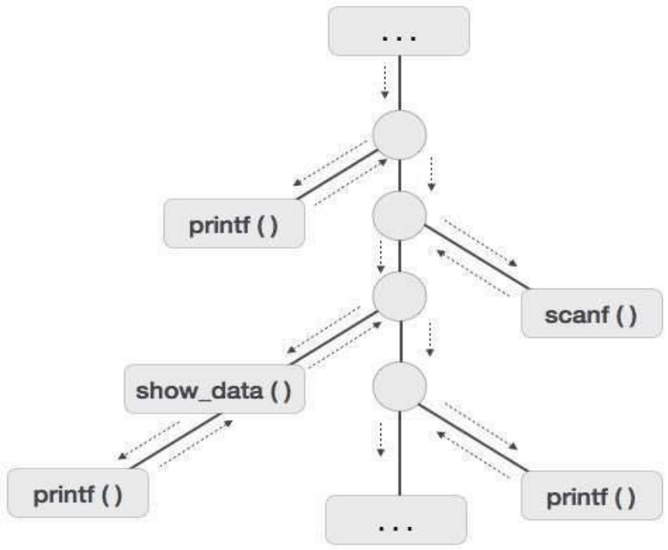
- This is the less popular method of parameter passing.
- Procedure treated as macro (macro-expansion, inline expansion): body substituted for call in caller, actual parameters literally substituted for the formals.
- Actual parameters are surrounded by parenthesis before substitution (if necessary to preserve their lexical boundaries)
- swap(x, y) cannot be written with call-by-name.

Activation Tree

- Every execution of a procedure is called an **ACTIVATION.**
- The LIFETIME of an activation of procedure P is the sequence of steps between the first and last steps of P's body, including any procedures called while P is running.
- When activations are nested, we can represent control flow with ACTIVATION TREES.

Activation Tree Definition.

- Represent the activations of procedures during the running of an entire program by a tree, called an Activation Tree.



How to build an activation tree?

Conditions

- Each node represents the activation of the procedure
- The root represents the activation of the main program
- The node for a is the parent of node b iff control flows from activation a to b
- The node for a is to the left of the node for b iff the lifetime(a) occurs before lifetime(b)

Activation Tree for a program

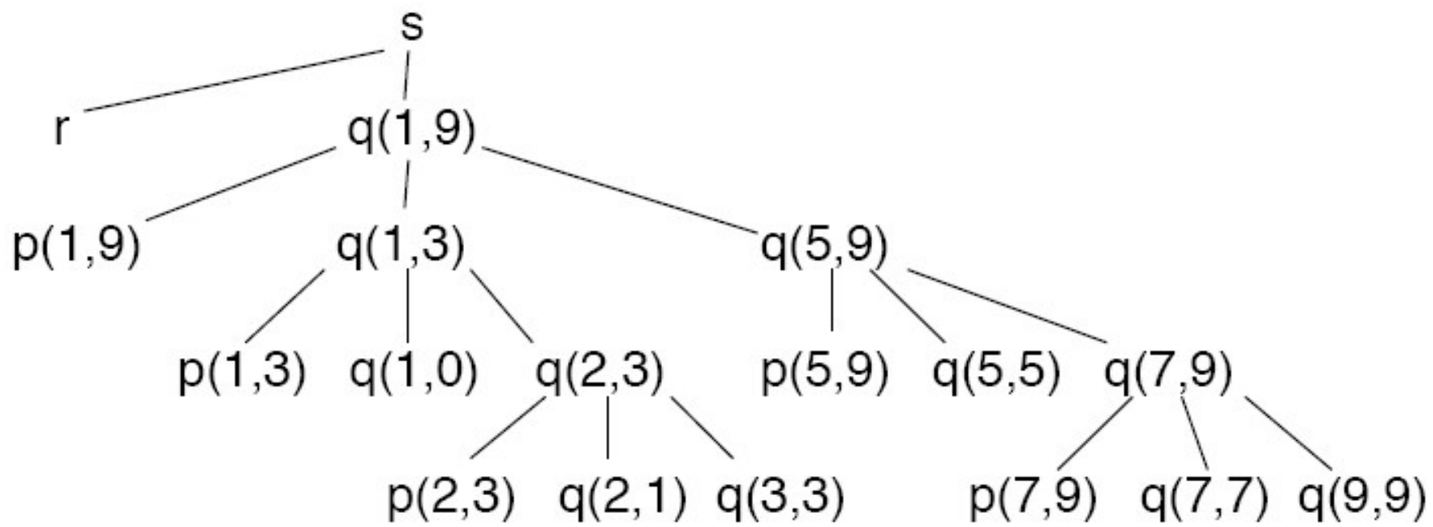
```
PROGRAM sort;
  VAR a : array[0..10] of Integer;
  PROCEDURE readarray;
    VAR i : Integer;
    ....

FUNCTION partition(y,z : Integer): Integer;
   VAR i,j,x,v : Integer;
   BEGIN
     ...
   END;
```

```
PROCEDURE quicksort(m,n : Integer);
  {
   VAR i : Integer;
   i := partition(m,n);
        quicksort(m, i-1);
        quicksort(i+1,n)
 }

main()
   {
      readarray;
      quicksort(1,9)
   }
```

Activation trees



Execution begins...

enter readarray
leave readarray

enter quicksort(1,9)

enter partition(1,9)
leave partition(1,9)

enter quicksort(1,3)
...
leave quicksort(1,3)

enter quicksort(5,9)
...
leave quicksort(5,9)

leave quicksort(1,9)

Execution terminated.

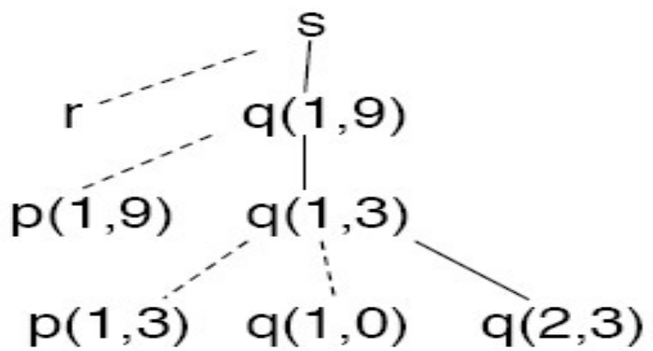
Activation Tree follows the "Depth first Traversal" (i.e) nodes should be visited before moving to the children.

Control stacks

- We can use a stack to keep track of currently-active activations.
  - when a procedure is called → push a record onto the stack
  - procedure returns → pop that record off the stack
- At any point in time, the control stack represents a path from the root of the activation tree to one of the nodes.

Example - Control Stack

This partial activation tree corresponds to control stack (growing downward)



Stack contains

q(2,3) ← Top of stack
q(1,3)
q(1,9)
s

Scope of the Declaration

Lifetime of Declarations

- **Implicit Declarations**
  - Some languages like Perl and Python have IMPLICIT declarations.

    var i;
- **Explicit Declarations**
  - In Pascal and C, declarations are EXPLICIT:
    var i : integer;

lvalue and rvalue

- An **lvalue (locator value)** represents an object that occupies some identifiable location in memory (i.e. has an address).
- **rvalues** are defined, by saying that every expression is either an *lvalue* or an *rvalue*.

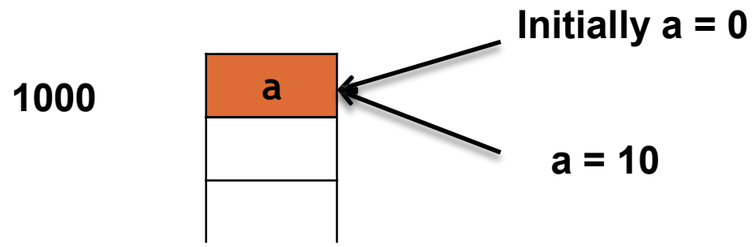
Environments and states

- The **ENVIRONMENT** is a function mapping from names to storage locations.
- The **STATE** is a function mapping storage locations to the values held in those locations.



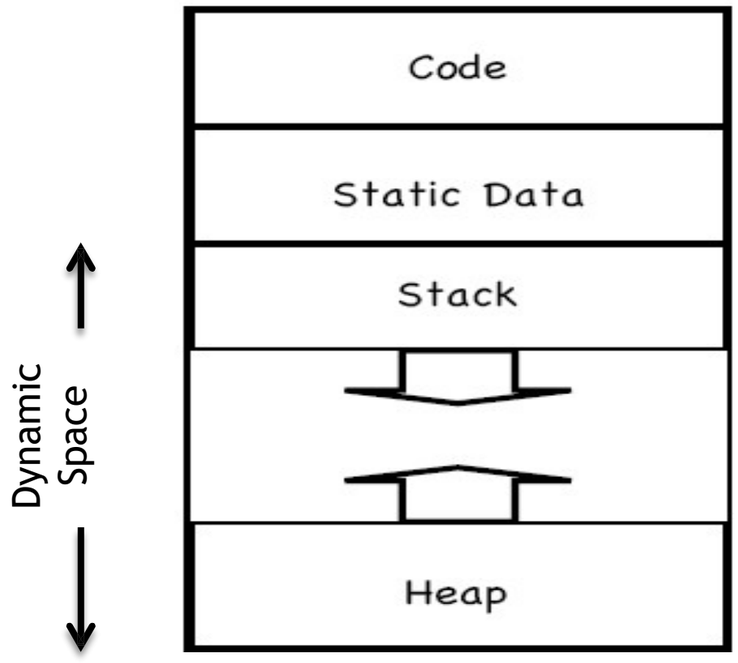
- Envi .
- States map l-values to r-values.

Example - Environments and states



- State may change, but environment doesn’t change

Storage Organization



This runtime storage must hold

- Generated target code
- Data objects
- Control stack to keep track of procedure activations
- Heap

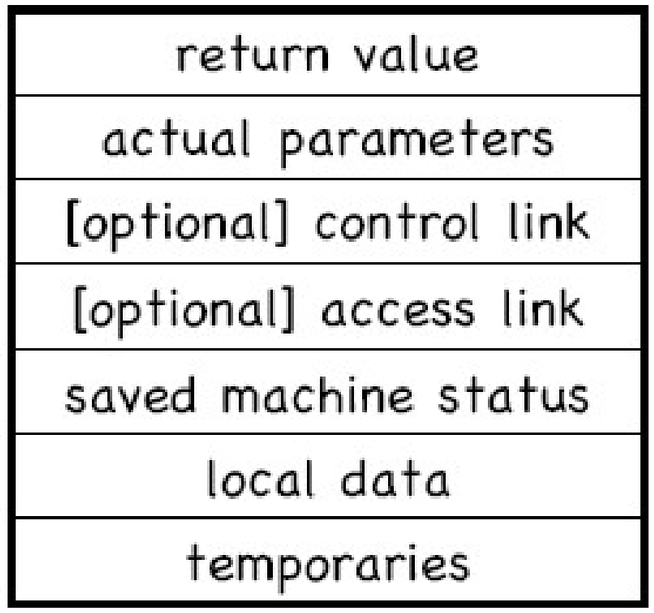
Control Stack

- The CONTROL STACK is used to store:
  - **Procedure activations.**
    - The status of the machine just before calling a procedure, so that the status can be restored when the called procedure returns.
- HEAP holds all other information.
  - For example, activations that cannot be represented as a tree.

Activation records

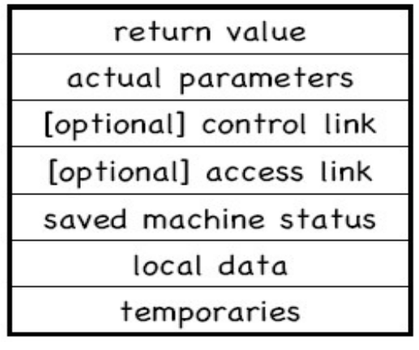
Important University Question

- Information needed by a single execution of a procedure is managed using an **ACTIVATION RECORD** or **STACK FRAME.**



- **Calling Procedure → Activation record Pushed.**
- **Returned Procedure → Activation record popped**

Activation Record - Details

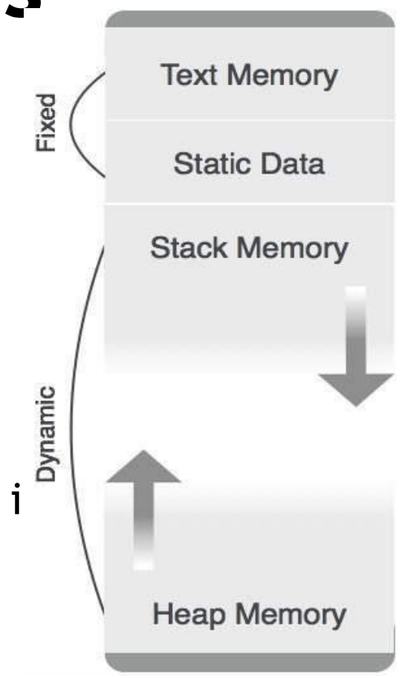


1) Temporary values
   - Used to store temporary values.
2) Local data
   - Local data is active only in that procedure
3) Saved machine status
   - State of the machine information before procedure is called.
   - PROGRAM COUNTER + MACHINE REGISTER values need to be restored when control returns from the procedure
4) Access Link
   - Refer to non-local data held in other activation records
5) Control link
   - Points to the activation record of the caller
6) Actual parameters
   - Used by the calling procedure to supply parameters to the called procedure
7) Returned value
   - Used by the called procedure to return a value to the calling procedure

Storage Allocation Strategies

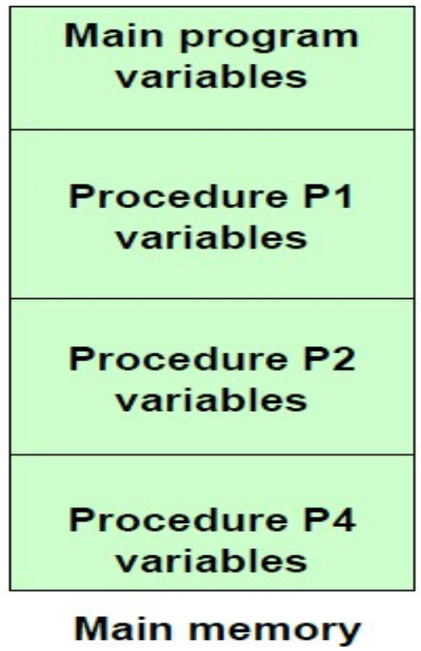
- **Static allocation**
  - allocates storage at compile time

- **Dynamic allocation**
- Storage allocation decisions are made only while the program i
  - **Stack allocation**
    - manages run time storage as stack
  - **Heap allocation**
    - allocates and de-allocates storage area as needed at runtime.



Static Storage Allocation

- Compiler allocates space for all variables (local and global) of all procedures at compile time
- Variable access is fast since addresses are known at compile time



Limitations:

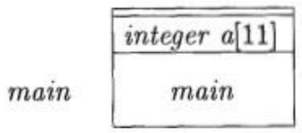
- The size required must be known at compile time.
- Recursive procedures cannot be implemented as all locals are statically allocated.
- No data structure can be created dynamically as all data is static.

Dynamic Storage Allocation

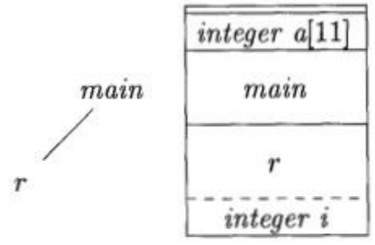
- Compiler allocates space only for global variables at compile time
- Space for variables of procedures will be allocated at run-time
- Stack/heap allocation
  - Variable access is slow (compared to static allocation) since addresses are accessed through the stack/heap pointer
  - Recursion can be implemented

Possible activations for the program

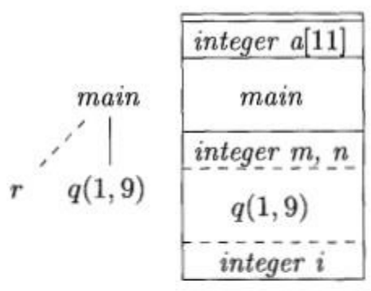
- If an activation of **procedure p calls procedure q**, then that activation of q must end before the activation of p can end. There are three common cases:
  - The activation of q terminates normally.
  - The activation of q is impossible to continue.
  - The activation of q terminates because of an exception, so ends.



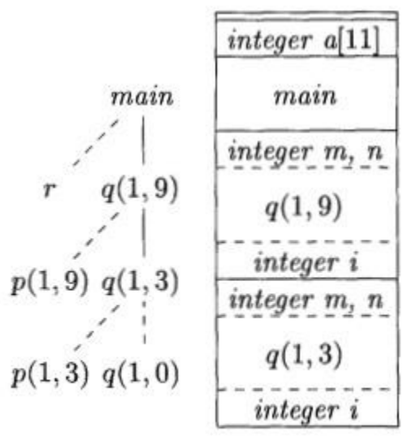
(a) Frame for *main*



(b) r is activated



(c) r has been popped and $q(1,9)$ pushed



(d) Control returns to $q(1,3)$

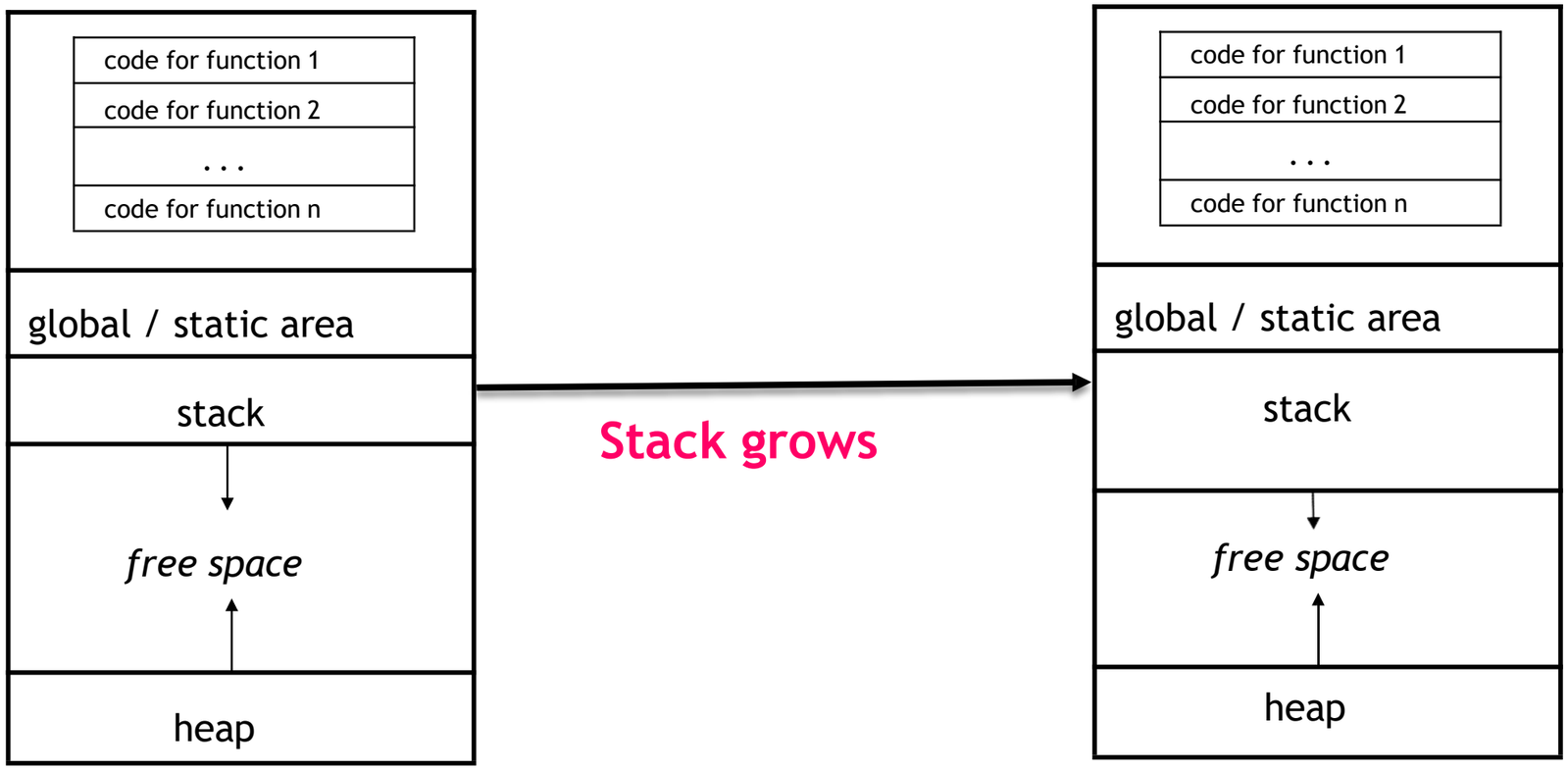
```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Working.

- When control reaches the first call in the body of main, procedure r is activated, and its activation record is pushed onto the stack as in (b). The activation record for r contains space for local variable i. When control returns from this activation, its record is popped, leaving just the record for main on the stack.

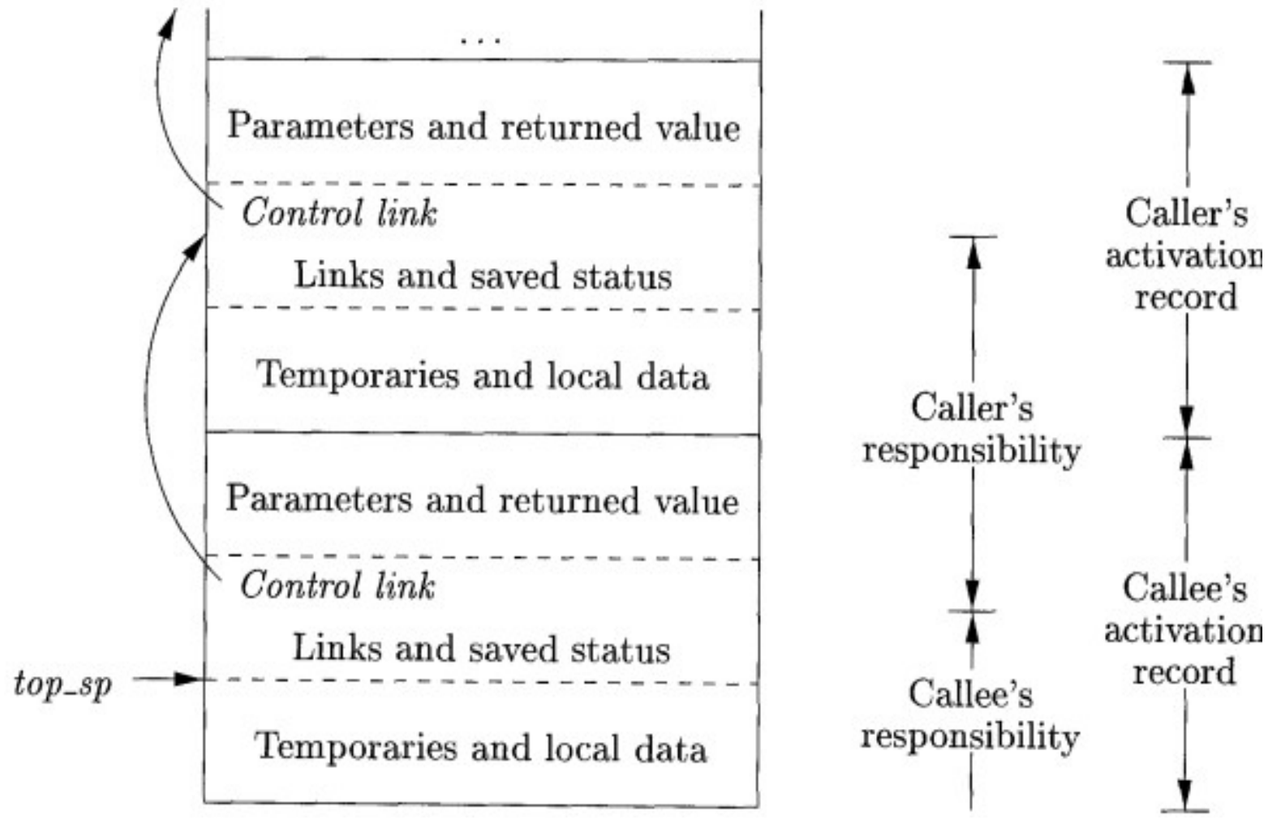
- Control then reaches the call to q (quicksort) with actual parameters 1 and 9, and an activation record for this call is placed on the top of the stack, as in (c). When q(l,9) returns, the stack again has only the activation record for main.

Actual Runtime Memory



Creation of An Activation Record (cont.)

Caller's Activation Record



Typical Actions During Call / Return

Calling sequence actions

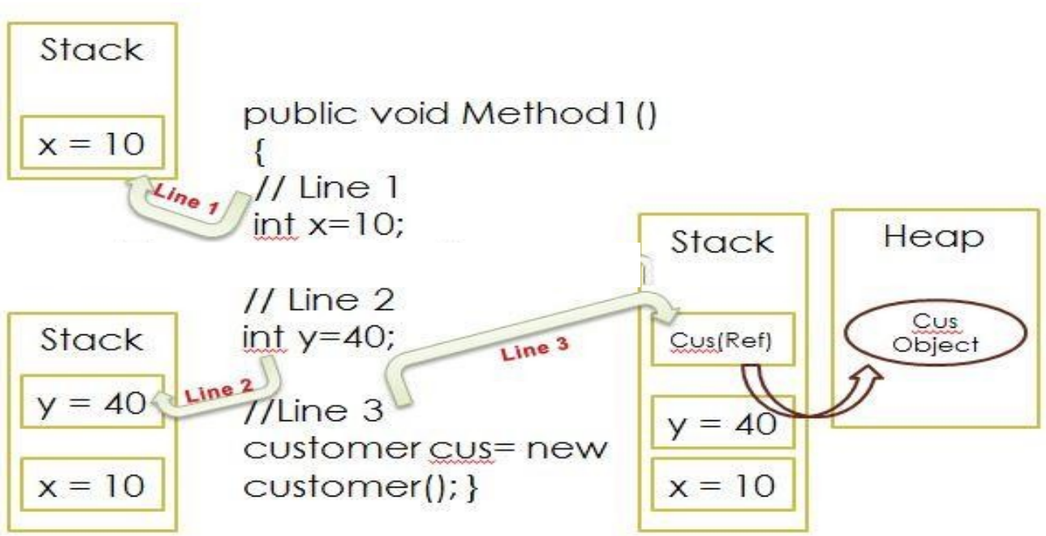
- caller assigns the actual arguments.
- caller stores return address.
- caller jumps to callee.
- callee adjusts stack pointer for the activation record.
- callee saves nonscratch register values it will use.

Return sequence actions

- callee assigns the return value.
- callee restores nonscratch register values and stack pointer address.
- callee jumps to the return address.
- caller accesses the return value.

Heap allocation

- Heap allocation -- allocates and deallocates, storage as needed at runtime from a data area known as heap.
  - Most flexible: no longer requires the activation of procedures to be LIFO.
  - Most inefficient: need true dynamic memory management.



Stack → contains variables and Function calls
Heap → contains objects

Introduction

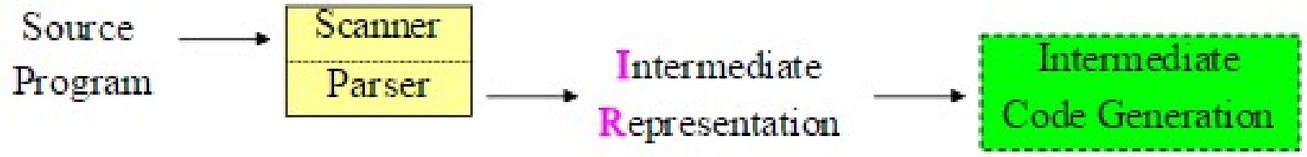
- **Formal Specification**
  - lexical structure : regular expression
  - syntactic structure : context-free grammar
  - the remaining phases of compilation : no such notations

    ⇒but, we use a syntax-directed translation scheme which is a method associating semantic rules(or actions) with production.

- **SDTS ::= cfg + semantic actions**

- **Intermediate code generation**
  - the **phase** that generates an explicit intermediate code of the source program.
  - after syntax and semantic analysis.

♣ **A Model for Intermediate code generation**



Completed