Syntax Directed Translation

Syntax-Directed Translation

- We associate information with the programming language constructs by attaching attributes to grammar symbols.

- Values of these attributes are evaluated by the **semantic rules** associated with the production rules.

- Evaluation of these semantic rules:
  - may generate intermediate codes
  - may save information into the symbol table
  - may perform type checking
  - may issue error messages
  - may perform some other activities

- An attribute may hold almost any thing
  - a string, a number, a memory location, a complex record.

Syntax-Directed Translation

- E→E1+E2 {E.val =E1.val + E2.val}
- A→XYZ {Y.val = 2 * A.val}

Syntax-Directed Translation

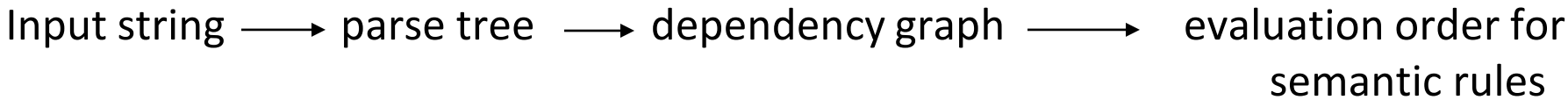
- Two notations to associate semantic rules with productions
  - Syntax-Directed Definitions
  - Translation Schemes

- Syntax-Directed Definitions
  - give high-level specifications for translations
  - hide many implementation details such as order of evaluation of semantic actions.
  - We associate a production rule with a set of semantic actions, and we do not say when they will be evaluated.

- Translation Schemes
  - indicate the order of evaluation of semantic actions associated with a production rule.
  - In other words, translation schemes give a little bit information about implementation details.

Syntax-Directed Translation

- Conceptually with both the syntax directed definition and translation scheme we
  - Parse the input token stream
  - Build the parse tree
  - Traverse the tree to evaluate the semantic rules at the parse tree nodes.



Conceptual view of syntax directed translation

Syntax-Directed Definitions

- A **syntax-directed definition** is a generalization of a context-free grammar in which
  - Each grammar symbol has an associated set of attributes partitioned into two subsets called **synthesized** and **inherited attributes** of that grammar symbol
- The value of an attribute at a parse tree node is defined by the semantic rule associated with a production at that node
- The value of a **synthesized attribute** at a node is computed from the values of attributes at the children in that node of the parse tree
- The value of an **inherited attribute** at a node is computed from the values of attributes at the siblings and parent of that node of the parse tree

Syntax-Directed Definitions

Synthesized attribute : $E \rightarrow E1+E2$ {E.val =E1.val + E2.val}

Inherited attribute : $A \rightarrow XYZ$ {Y.val = 2 * A.val}

1. *Semantic rules* set up dependencies between attributes which can be represented by a *dependency graph*.
2. This *dependency graph* determines the evaluation order of these semantic rules.
3. Evaluation of a semantic rule defines the value of an attribute.

Annotated Parse Tree

- A parse tree showing the values of attributes at each node is called an **annotated parse tree**.
- Values of attributes in nodes of annotated parse tree are either
  - initialized to constant values by the lexical analyzer.
  - determined by the semantic-rules.
- The process of computing the attributes values at the nodes is called **annotating** (or **decorating**) the parse tree.
- The order of these computations depends on the dependency graph induced by the semantic rules.

Syntax-Directed Definition

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

$$b=f(c_1,c_2,\ldots,c_n)$$

where *f* is a function and either *b*

- *b* is a synthesized attribute of A and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols of the production

OR

- *b* is an inherited attribute one of the grammar symbols on the right side of the production and $c_1,c_2,\ldots,c_n$ are attributes of the grammar symbols in the production

Attribute Grammar

- So, a semantic rule $b=f(c_1,c_2,\ldots,c_n)$ indicates that the attribute b *depends* o*n* attributes $c_1,c_2,\ldots,c_n$.

- In a **syntax-directed definition**, a semantic rule may just evaluate a value of an attribute or it may have some side effects such as printing values.

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

Syntax-Directed Definition - Example

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

1. Symbols E, T, and F are associated with a synthesized attribute *val*.
2. The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).
3. Terminals are assumed to have synthesized attributes only. Values for attributes of terminals are usually supplied by the lexical analyzer.
4. The start symbol does not have any inherited attribute unless otherwise stated.

S-attributed definition

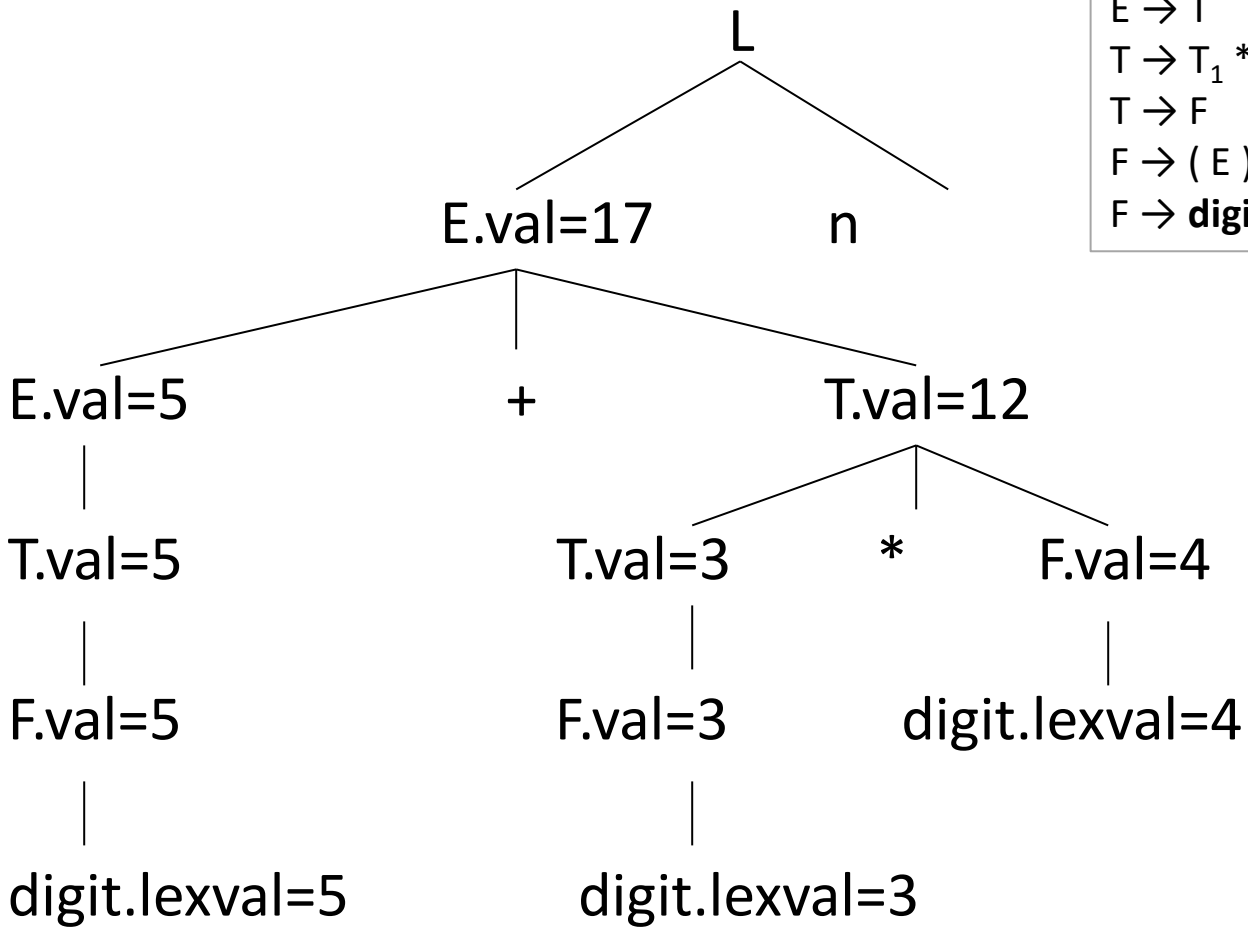
- A syntax directed definition that uses synthesized attributes exclusively is said to be a **S-attributed definition**.

- A parse tree for a S-attributed definition can be annotated by evaluating the semantic rules for the attributes at each node, bottom up from leaves to the root.

Annotated Parse Tree -Example

L → E **n**	print(E.val)
E → E_1 + T	E.val = E_1.val + T.val
E → T	E.val = T.val
T → T_1 * F	T.val = T_1.val * F.val
T → F	T.val = F.val
F → (E)	F.val = E.val
F → **digit**	F.val = **digit**.lexval

Input: 5+3*4



Inherited attributes

- An inherited value at a node in a parse tree is defined in terms of attributes at the parent and/or siblings of the node.
- Convenient way for expressing the dependency of a programming language construct on the context in which it appears.
- We can use inherited attributes to keep track of whether an identifier appears on the left or right side of an assignment to decide whether the address or value of the assignment is needed.
- Example: The inherited attribute distributes type information to the various identifiers in a declaration.

Syntax-Directed Definition – Inherited Attributes

Production	**Semantic Rules**
D → T L	L.in = T.type
T → **int**	T.type = integer
T → **real**	T.type = real
L → L_1, **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
L → **id**	addtype(**id**.entry,L.in)

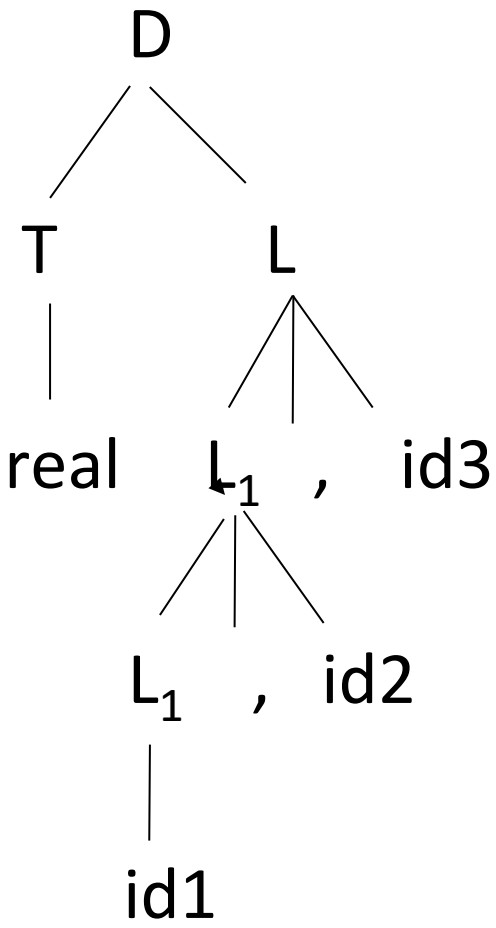
1. Symbol T is associated with a synthesized attribute *type*.

2. Symbol L is associated with an inherited attribute *in.*

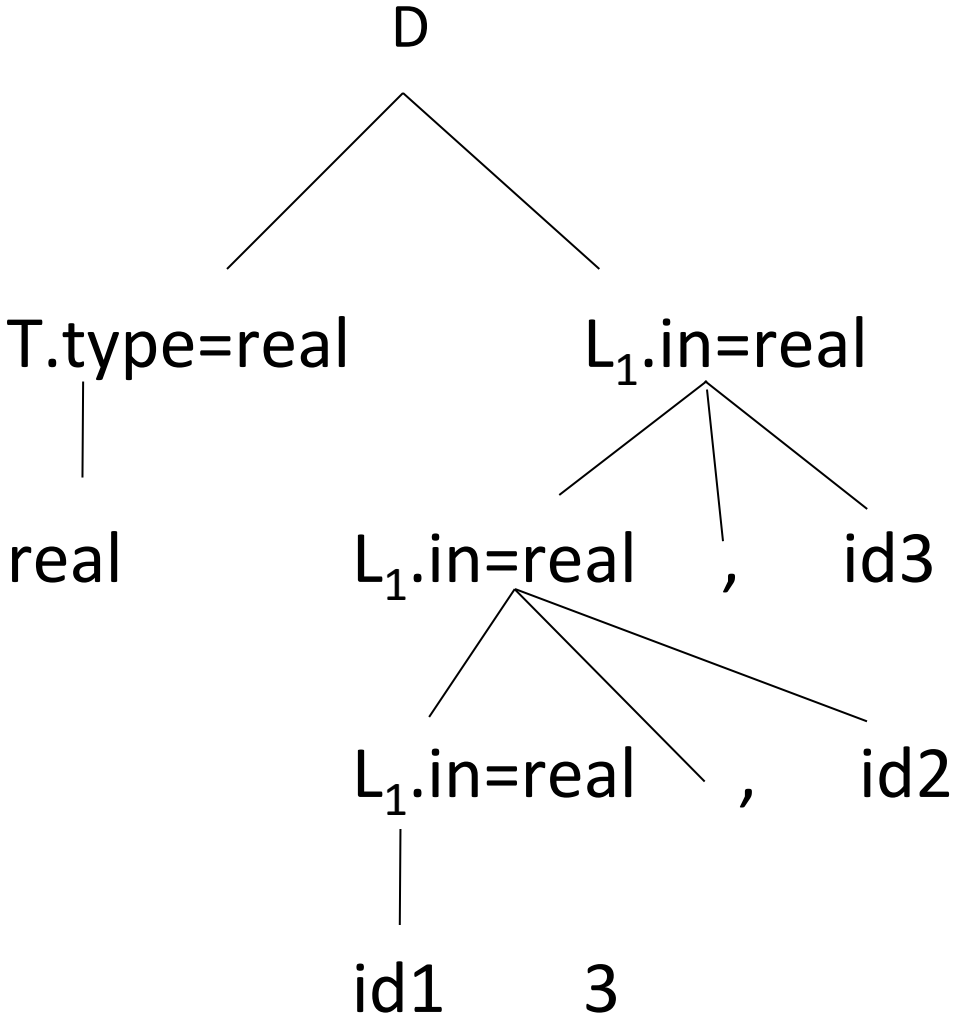
Annotated parse tree

Input: `real p,q,r`

parse tree



annotated parse tree



Dependency Graph

- Directed Graph
- Shows interdependencies between attributes.
- If an attribute b at a node depends on an attribute c, then the semantic rule for b at that node must be evaluated after the semantic rule that defines c.
  - Put each semantic rule into the form $b=f(c_1,\ldots,c_k)$ by introducing dummy synthesized attribute b for every semantic rule that consists of a procedure call.
  - The graph has a node for each attribute and an edge to the node for b from the node for c if attribute b depends on attribute c.

Dependency Graph Construction

for each node *n* in the parse tree do

  for each attribute ***a*** of the grammar symbol at ***n*** do

    construct a node in the dependency graph for ***a***

for each node *n* in the parse tree do

  for each semantic rule $b = f(c_1,\ldots,c_n)$

    associated with the production used at *n* do

      for i= 1 to n do

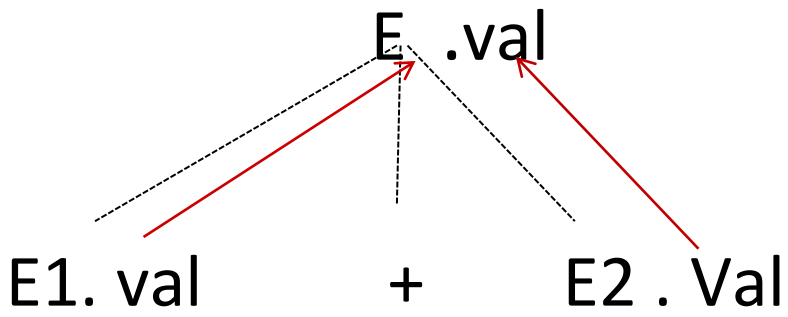
        construct an edge from

        the node for c_i to the node for *b*

Dependency Graph Construction

- Example
- Production — Semantic Rule

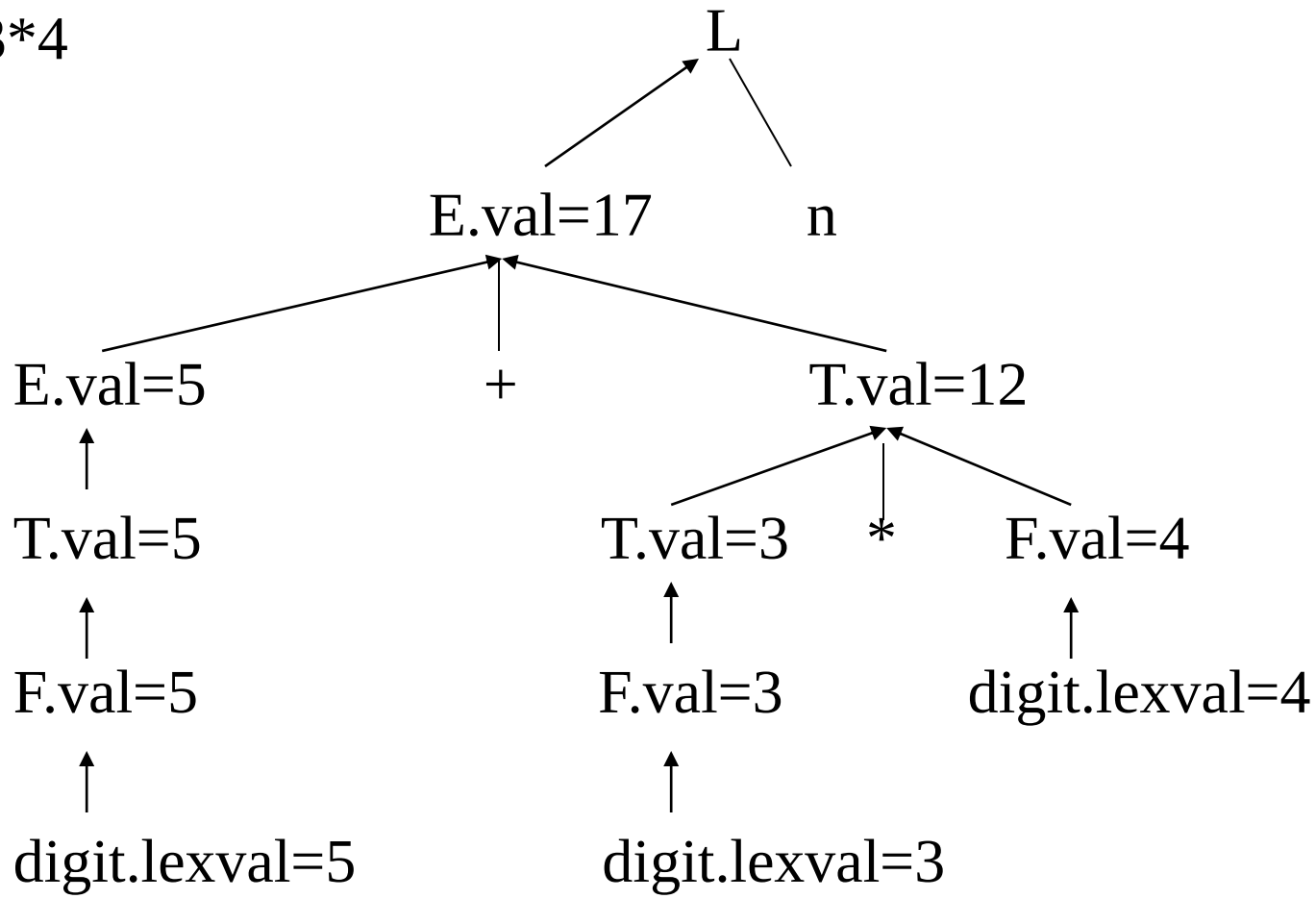
Production	Semantic Rule
E→E1 + E2	E.val = E1.val + E2.val



- E.val is synthesized from E1.val and E2.val
- The dotted lines represent the parse tree that is not part of the dependency graph.

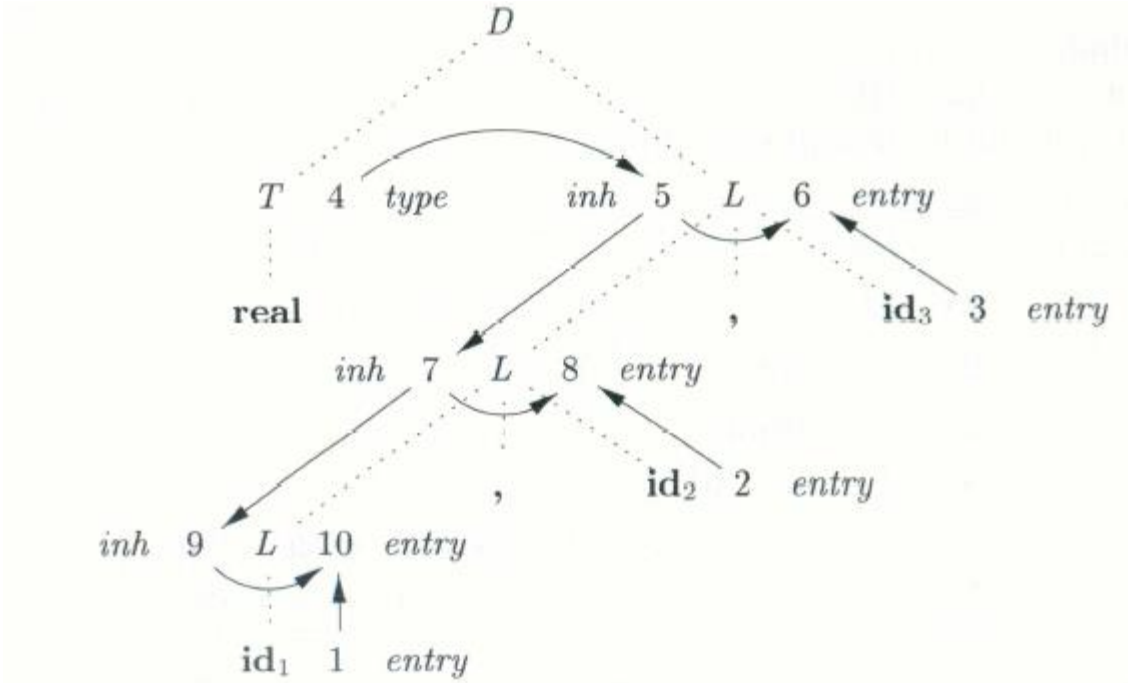
Dependency Graph

Input: 5+3*4



Dependency Graph

$D \rightarrow T\ L$	L.in = T.type
$T \rightarrow$ **int**	T.type = integer
$T \rightarrow$ **real**	T.type = real
$L \rightarrow L_1$, **id**	L_1.in = L.in, addtype(**id**.entry,L.in)
$L \rightarrow$ **id**	addtype(**id**.entry,L.in)



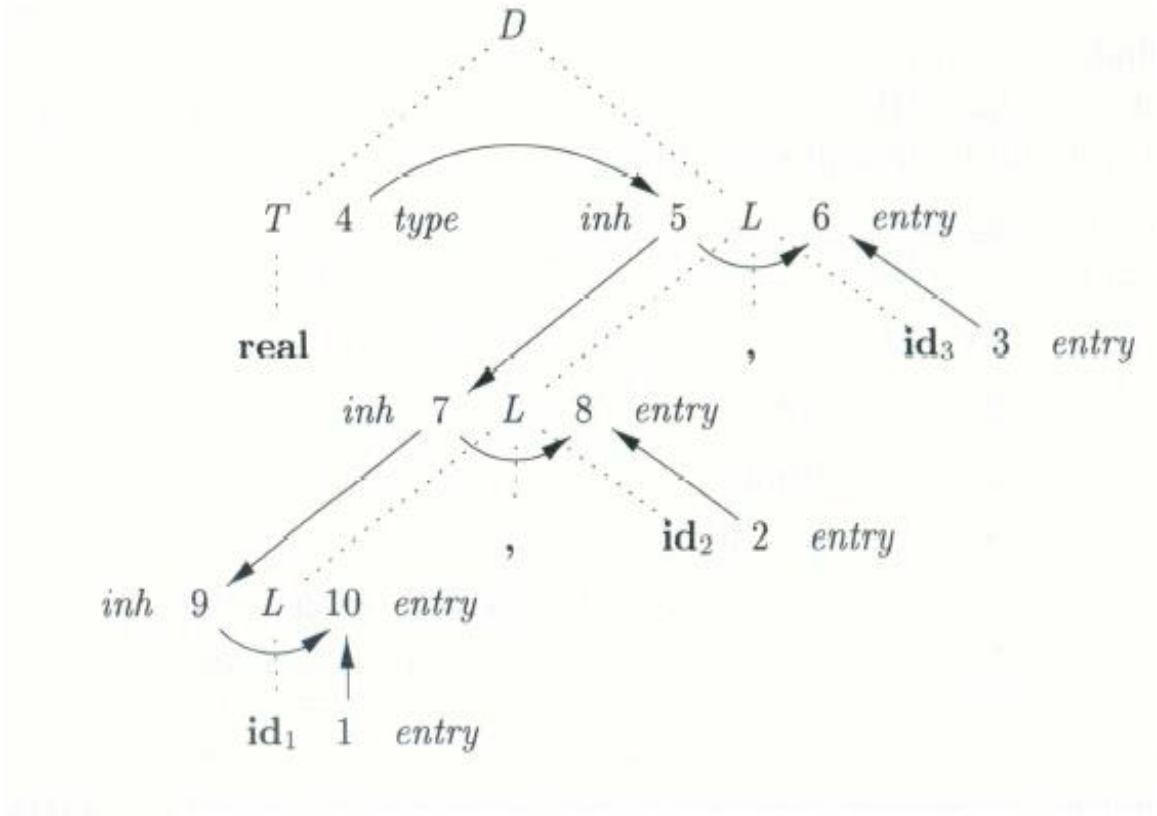
Dependency graph for a declaration **real** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

Evaluation Order

- A topological sort of a directed acyclic graph is any ordering m1,m2…mk of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes.
  - i.e if there is an edge from m_i to m_j them m_i appears before m_j in the ordering
- Any topological sort of dependency graph gives a valid order for evaluation of semantic rules associated with the nodes of the parse tree.
  - The dependent attributes c1,c2….ck in b=f(c1,c2….ck) must be available before f is evaluated.
- Translation specified by Syntax Directed Definition

- Input string $\longrightarrow$ parse tree $\longrightarrow$ dependency graph $\longrightarrow$ evaluation order for semantic rules

Evaluation Order



Dependency graph for a declaration **real** $\mathbf{id}_1$, $\mathbf{id}_2$, $\mathbf{id}_3$

- a4=real
- a5=a4;
- addtype(id3.entry,a5);
- a7=a5;
- addtype(id2.entry,a7);
- a9=a7;
- addtype(id1.entry,a9);

Translation of SDD

- In the context of a syntax directed definition, a semantic rule specifies how different attributes are related.

- Can an attribute of a grammar symbol defined in terms of the any other grammar symbols of the same production?

- The extent of restriction gives rise to two classes of translations
  - S-attributed
  - L-attributed

S-Attributed Definitions

- A translation method is S-attributed if
  - Every grammar symbol has synthesized attributes only and
  - All actions occur on right hand side of productions

- Synthesized attributes can be evaluated by a bottom up parser as the input is being parsed.
- The parser can keep values of the synthesized attributes associated with the grammar symbols on the stack
- Whenever a reduction is made, the values of the new synthesized attributes are computed from the attributes of the grammar symbols on the right hand side of the production.

L-Attributed Definitions

- A syntax-directed definition is **L-attributed** if each inherited attribute of X_j, where $1 \leq j \leq n$, on the right side of $A \rightarrow X_1X_2...X_n$ depends only on
  1. The attributes of the symbols $X_1,...,X_{j-1}$ to the left of X_j in the production
  2. The inherited attribute of A

- Every S-attributed definition is L-attributed, since the restrictions apply only to the inherited attributes (not to synthesized attributes).

A Definition which is *not* L-Attributed

Productions	**Semantic Rules**
$A \rightarrow L\ M$	L.i=l(A.i)
	M.i=m(L.s)
	A.s=f(M.s)
$A \rightarrow Q\ R$	R.i=r(A.i)
	Q.i=q(R.s)
	A.s=f(Q.s)

- This syntax-directed definition is not L-attributed because the semantic rule Q.i=q(R.s) violates the restrictions of L-attributed definitions.
- The value of Q.i depends on R.s of the grammar symbol to its right.

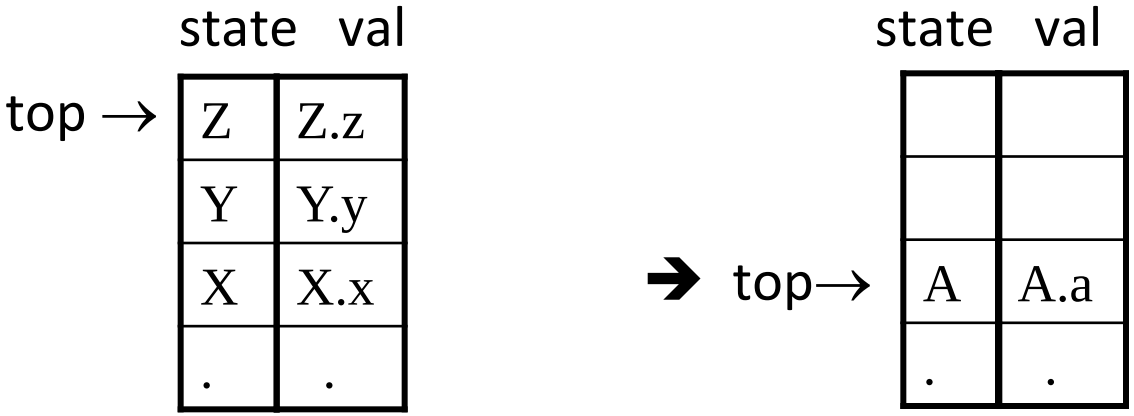
Bottom-Up Evaluation of S-Attributed Definitions

- A translator for an S-attributed definition can often be implemented with the help of an LR parser.
- From an S-attributed definition the parser generator can construct a translator that evaluates attributes as it parses the input.
- The stack has extra fields to hold the values of synthesized attributes
  - The stack is implemented by a pair of arrays *val & state*
  - If the i^{th} state symbol is A the val[i] will hold the value of the attribute associated with the parse tree node corresponding to this A.

Bottom-Up Evaluation of S-Attributed Definitions

- We evaluate the values of the attributes during reductions.

  $A \rightarrow XYZ$ A.a=f(X.x,Y.y,Z.z) where all attributes are synthesized.



- Synthesized attributes are evaluated before each reduction.
- Before XYZ is reduced to A, the value of Z.z is in val[top], that of Y.y in val[top-1] and that of X.x in val[top-2].
- After reduction top is decremented by 2.
- If a symbol has no attribute the corresponding entry in the array is undefined.

Bottom-Up Evaluation of S-Attributed Definitions

Production	**Semantic Rules**
L → E **n**	print(val[top-1])
$E \rightarrow E_1 + T$	val[ntop] = val[top-2] + val[top]
E → T	
$T \rightarrow T_1 * F$	val[ntop] = val[top-2] * val[top]
T → F	
F → (E)	val[ntop] = val[top-1]
F → **digit**	

1. At each shift of **digit**, we also push **digit.lexval** into *val-stack*.
2. At all other shifts, we do not put anything into *val-stack* because other terminals do not have attributes (but we increment the stack pointer for *val-stack*).

Bottom up Evaluation of S-Attributed definition

Input	State	Val	Production used
3*5+4n	-	-	
*5+4n	3	3	
*5+4n	F	3	F → digit
*5+4n	T	3	T → F
5+4n	T*	3-	
+4n	T*5	3-5	
+4n	T*F	3-5	F → digit
+4n	T	15	T → T*F
+4n	E	15	E → T
4n	E+	15-	
n	E+4	15-4	
n	E+F	15-4	F → digit
n	E+T	15-4	T → F
n	E	19	E → E+T
	En	19-	
	L	19	L → En

Runtime Environment

Runtime Environment

- Runtime Environment is the structure of the target computer's registers and memory that serves to manage memory and maintain the information needed to guide the execution process.
- Data objects (variables) come into existence during run time.
- Code refers to data objects using addresses that are resolved during compilation.
- Address depends on their organization in memory which to a great extent is decided by source language features.
- Allocation & deallocation is managed by run time support package consisting of routines loaded with the target code.
- Design of run time package is determined by the semantics of procedures

Runtime Environment

- A program is made up of procedures.
- Each execution of a procedure is referred to as activation of the procedure
- If a procedure is recursive, several of its activations may be alive at the same time
- Each execution or call of a procedure leads to an activation that may manipulate data objects allocated for its use.
- Representation of data object at run time is determined by its type.
- int,float etc.- Equivalent data objects in target machine.
- Array, string etc.- are represented by collection of primitive objects.

Source Language Issues

1. Procedure

- A procedure definition is a declaration that associates an identifier (procedure name) with a statement (procedure body)

- When a procedure name appears in an executable statement, it is *called* at that point

- A procedure call executes procedure body.

- *Formal parameters* are the one that appear in declaration.
- *Actual Parameters* are the one that appear in when a procedure is called, they are substituted for formals in the body.

2. Activation tree

- Assumptions about **flow of control** among procedures during execution of a program:
  - Control flows sequentially
  - Each execution of a procedure **starts** at the beginning of the **procedure body** and eventually **returns** to the point **immediately following** the place where the procedure was called.
- Each execution of a procedure body is referred as **an activation** of the procedure.
- Lifetime of an activation of a procedure ***P*** is the sequence of steps between the first and last steps in the execution of the procedure body, including time spent executing procedures called by ***P***, the procedures called by them and so on.
- If **a** and **b** are procedure activations, then their life times are either non-overlapping or are nested
  - i.e if **b** is entered before **a** is left then control must leave **b** before it leaves **a**.
- A procedure is recursive if a new activation can begin before an earlier activation of the same procedure has ended.

Activation tree

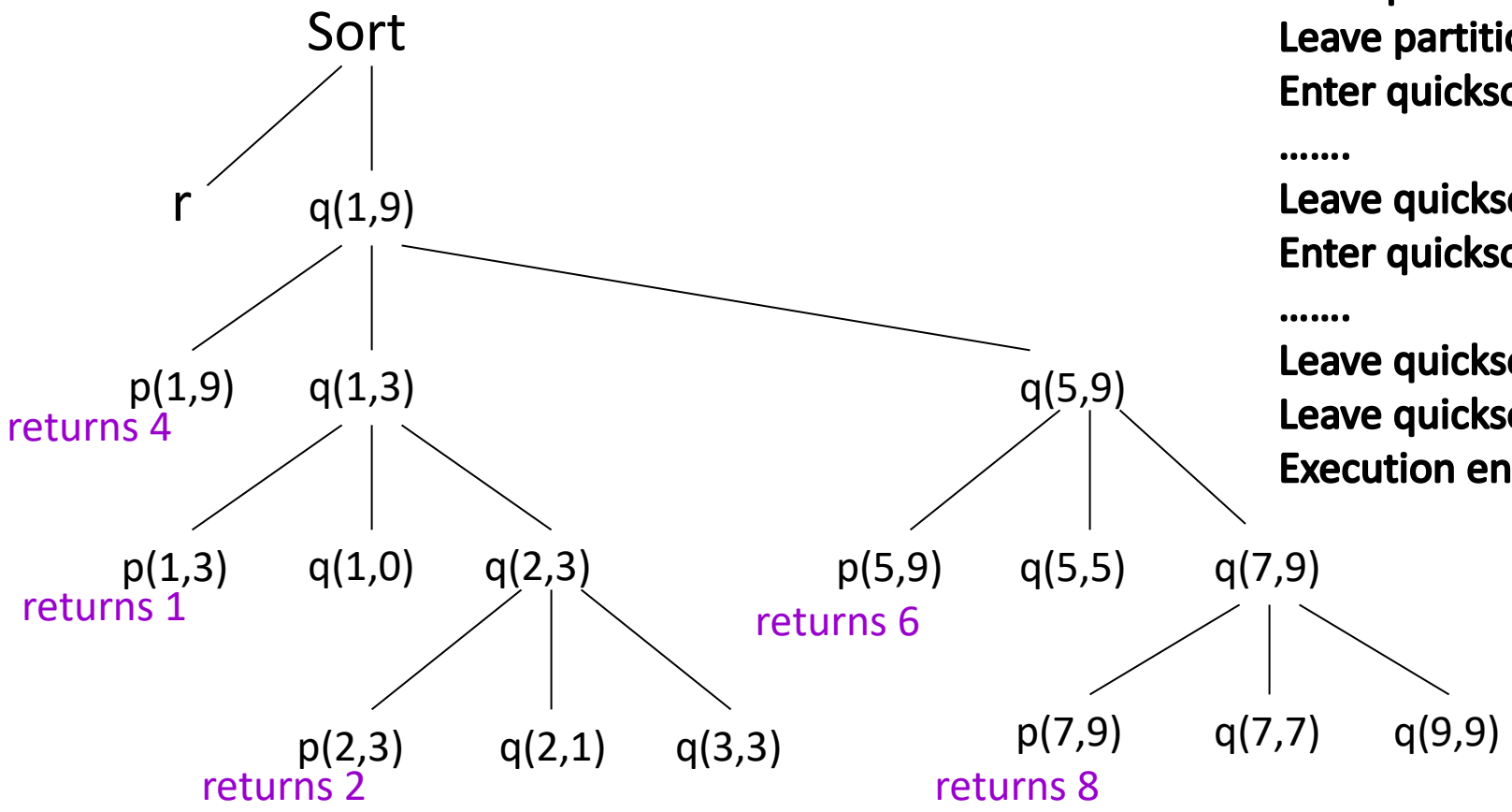
- A tree, called an activation tree, can be used to depict the way control enters and leaves activations
  - Each node represents an activation of procedure
  - The root represents the activation of main program
  - The node **a** is parent of **b** if control flows from **a** to **b**
  - The node **a** is to the left of node **b** if lifetime of **a** occurs before **b**

Example

```
program sort;
  var a : array[0..10] of integer;

procedure readarray;
begin
 var i :integer;
    :
end

function partition (y, z:integer) :integer;
begin
var i, j ,x, v :integer;
                :
end
```

```
procedure quicksort (m, n :integer);
begin
      var i :integer;
      if (n>m) then
      begin
      i:= partition (m,n);
      quicksort (m,i-1);
      quicksort(i+1, n);
   end;
end;


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

Activation Tree

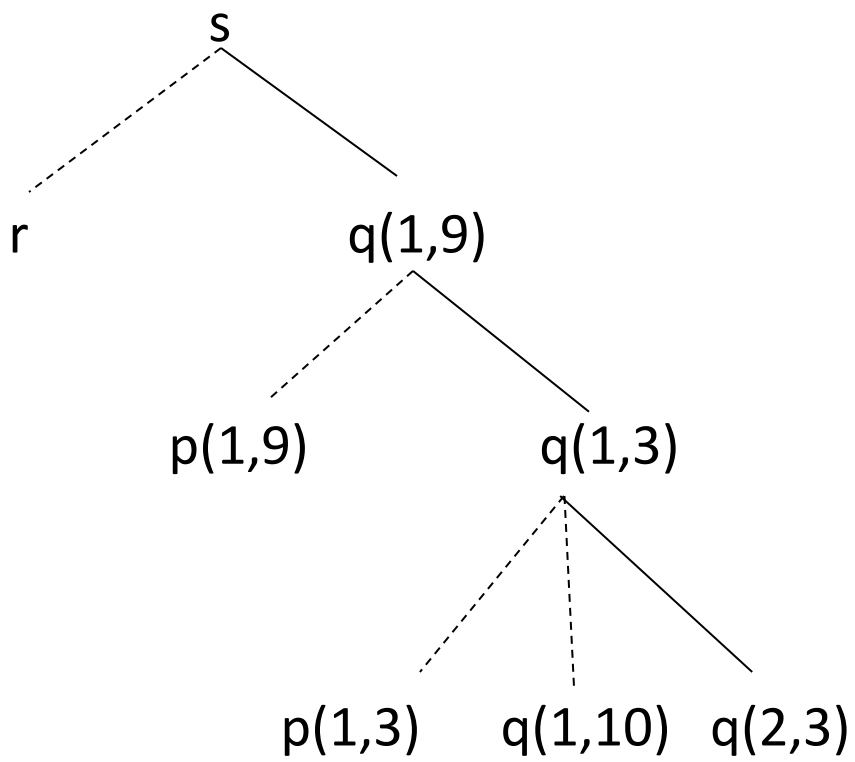


Execution begins
Enter readarray
Leave readarray
Enter quicksort(1,9)
Enter partition(1,9)
Leave partition(1,9)
Enter quicksort(1,3)
…….
Leave quicksort(1,3)
Enter quicksort(5,9)
…….
Leave quicksort(5,9)
Leave quicksort(1,9)
Execution ends

3. Control stack

- The flow of control in a program corresponds to a **depth-first-traversal** of the activation tree that starts at the root, visits a node before its children, and recursively visits children at each node in a **left-to-right** order.
- We can use a stack, called **control stack** to keep track of live procedure activations.
- The idea is to **push** the node for an activation onto the control stack as the activation begins and **pop** the node when the activation ends.
- Thus the contents of the stack are related to **paths** to the root of the activation tree.
- When node **n** is at the top of the control stack, the stack contains the nodes along the path from **n** to the root.

Control stack



At this point control stack contains the following nodes along the path to the root:

s ,q(1,9), q(1,3), q(2,3)

4. The Scope of a Declaration

- A declaration is a syntactic construct that associates the information with a name.
- Declaration may be
  - Explicit as in Pascal
    - **var i: integer**
  - Implicit as in Fortran
    - **Any variable name starting with I is assumed to denote an integer**
- The scope rules determine which declaration of a name applies when the name appears in the text of a program.
- The portion of the program to which a declaration applies is called **scope of that declaration**.
- An occurrence of a name in a procedure is said to be **local** to the procedure if it is in the scope of a declaration within the procedure, otherwise, the occurrence is said to be **non-local**.
- At compile time the symbol table can be used to find the declaration that applies to an occurrence of a name.

5.Binding of Names

- Even if each name is declared once in a program, the same name may denote different data objects at run time.
- Data object corresponds to a storage location that can hold values.
- The term environment refers to a function that maps a name to a storage location.
- The term state refers to a function that maps a storage location to the value held there.
- Environment: maps a name to an l-value
- State maps an l-value to an r-value

- name 

storage 

value

- Environments and states are different; an assignment changes the state but not the environment.

Binding of Names

- When an environment associates storage location **s** with a name **x**, we say that **x is *bound* to s**; the association itself is referred to as a ***binding*** **of x**.

- A binding is the dynamic counter part of a declaration.

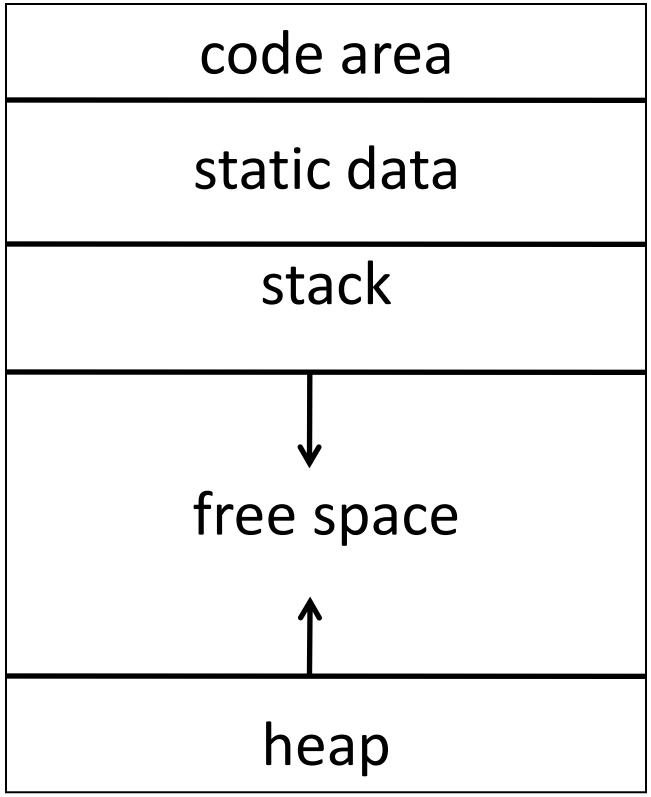
- In Pascal a local variable name in a procedure is bound to a different storage location in each activation of a procedure.

Static Notion	**Dynamic Counterpart**
Definition of a procedure	Activations of the procedure
Declaration of a name	Binding of the name
Scope of a declaration	Lifetime of a binding

Storage Organization

- The compiler obtains a block of storage from the operating system for the compiled program to run in.
- Run time storage might be subdivided to hold:
  - The generated target code
  - Data objects
  - A counterpart of the control stack to keep track of procedure activations.
- In most compiled languages, the size of the target code is fixed during compile time so the compiler can place it in a statically determined area.
- The size of the some data objects may also be known at compile time, and these too can be placed in a statically determined area.
- One reason for statically allocating as many data objects as possible is that addresses of these objects can be compiled into target code.

Storage Organization



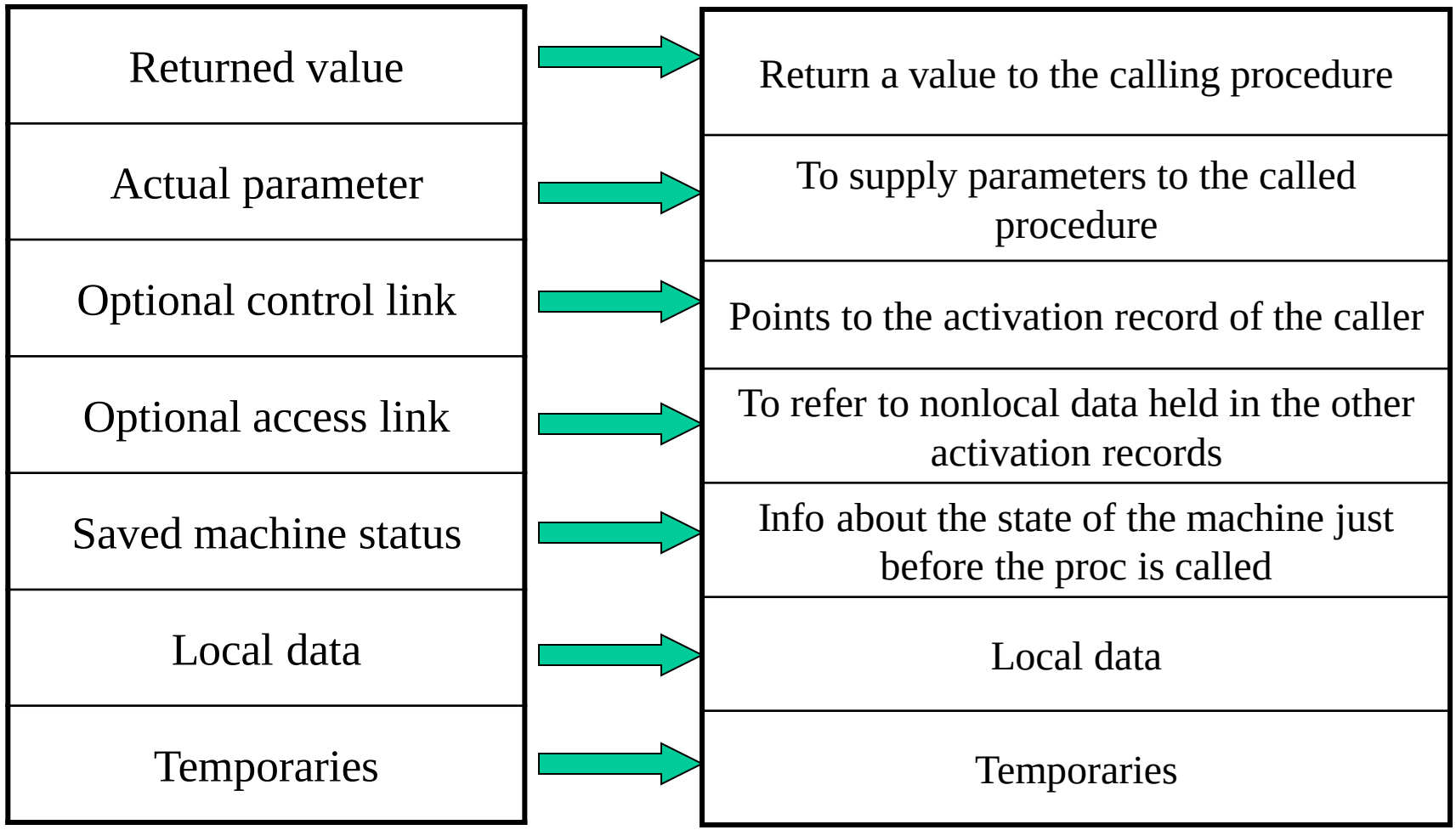
Why Stack and Heap?

- C/Pascal uses extensions of the control stack to manage activations of procedures.
- When a call occurs, execution of an activation is **interrupted** and information about the status of the machine, such as the value of the **program counter and machine registers**, is **saved** on the **stack**.
- When control returns from the call, this activation can be **restarted** after **restoring** the values of relevant registers and setting the **program counter** to the point **immediately after the call**.
- Heap, a separate area of run time memory holds all other information like the data under program control i.e. dynamic memory allocation.
- By convention, stacks grow down. Heaps grow up.

Activation Record

- An important unit of memory
- Information needed by a single execution of a procedure is managed using a contiguous block of storage called an *activation record or frame*.
- When procedure is called, the activation record of the procedure is pushed on the run time stack.
- When the control returns to caller, the activation record is popped off the stack
- When activation records are kept on stack, they are called *stack frames*

Activation Record



Layout of local data

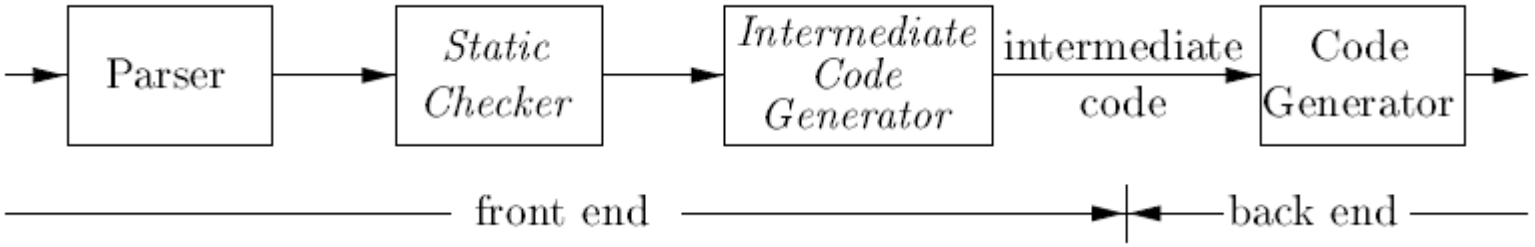
- Assume byte is the smallest unit and a group of bytes from a word
- Multi-byte objects are stored in consecutive bytes and given address of first byte
- The amount of storage needed is determined by its type
  - int, float, char etc. can be stored in an integral number of bytes
  - Storage for aggregates such as array or record is allocated in one contiguous block of bytes
- The field for local data is laid out as declarations in a procedure are examined at compile time.
- Variable length data is kept outside this field

Layout of local data

- A count of the memory locations allocated to the previous declarations is maintained.
- From the count, a relative address for the local with respect some position such as the beginning of activation record is determined
- Relative address or offset is difference between the address of the position and the data object
- Storage layout of local data is strongly influenced by addressing constraints of target machine.
  - Addition instruction: integers should be aligned at addresses divisible by 4
  - Array of ten characters requires only 10 bytes, compiler allocates 12 bytes, leaving 2 bytes unused
  - Space left unused due to alignment considerations is called padding
  - Complier may pack the data so no padding is left
  - Additional instructions may be required to execute packed data

Intermediate Code Generation

- In the analysis-synthesis model of a compiler
  - The front end translates the source program into an intermediate representation
  - The back end generates target code from intermediate representation
- Benefits of using machine independent intermediate form
  - Retargeting is facilitated
  - Machine independent code optimization can be applied.



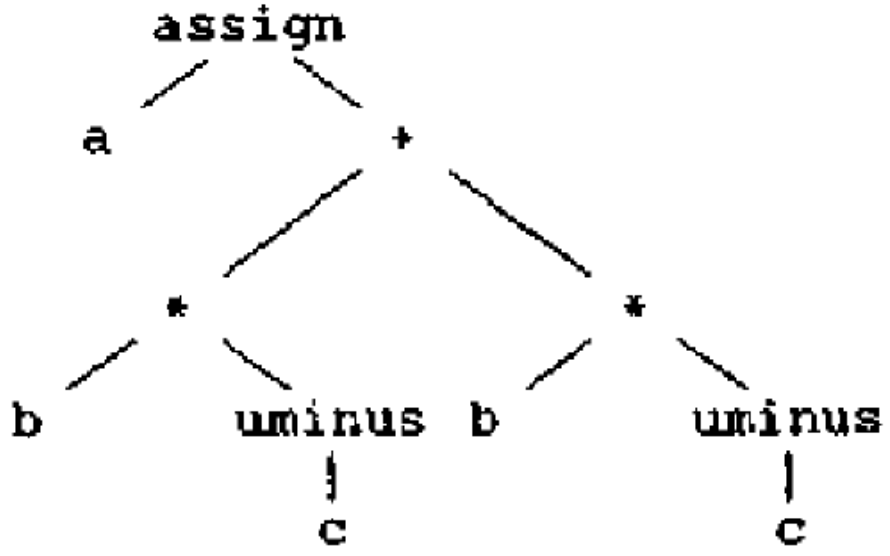
Position of intermediate code generator

Intermediate Languages

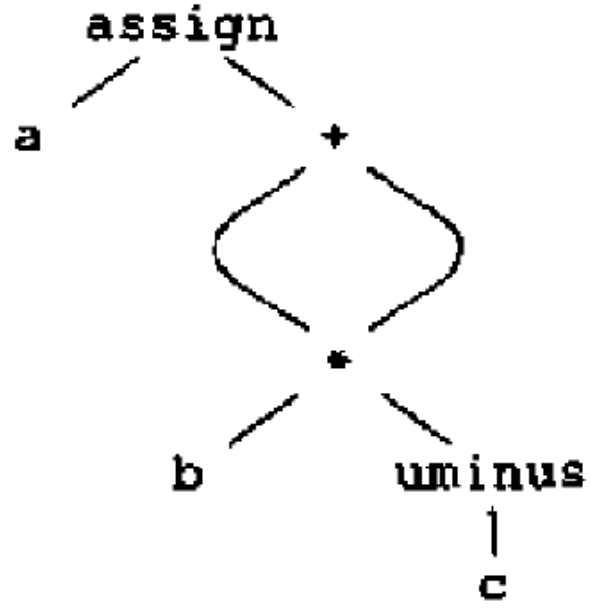
- **Three address code**
  - It is a general sequence of statements of the form x:=y op z
  - Each statement usually contains 3 addresses: two for operands and one for the result
- **Postfix notation**
  - Linearized representation of a syntax tree
  - It is a list of nodes in the tree in which a node appears immediately after its children
- **Graphical Representation**
  - **Syntax tree**
    - depicts the natural hierarchical structure of a source program
  - **DAG (Directed Acyclic Graph)**
    - gives the same information, but in a more compact way
    - common sub expressions are identified

Intermediate Languages

- Graphical Representation



(a) Syntax tree.



(b) Dag.

Graphical representations of a := b * - c + b * - c.

Intermediate Languages

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

- Postfix form: a b c unminus * b c unminus * + assign

Three Address Code

- Statements of general form x:=y op z
  where x, y, z are names, constants or compiler generated temporaries, op stands for any fixed or floating point operator or a logical operator on a Boolean valued data.
- Expression **x:=y + z * w** should be translated as

  t_1:=z * w

  t_2:=y + t_1

  x:=t_2
- Given, the syntax-tree or the dag of the graphical representation we can easily derive a three address code for assignments as above.
- Three-address code is a linearized representation of a syntax tree or DAG in which explicit names correspond to the interior nodes of the graph.
- There are no names corresponding to leaves

Three address code

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

- Three address code

t_1:= - c
t_2:= b * t_1
t_3:= - c
t_4:= b * t_3
t_5:= t_2 + t_4
a:= t_5

Code for DAG

t_1:= - c
t_2:= b * t_1
t_5:= t_2 + t_2
a:= t_5

Types of Three-Address Statements

1. **Assignment Statement:** x:=y op z
2. **Assignment Statement:** x:=op z
3. **Copy Statement:** x:=z
4. **Unconditional Jump:** goto L
5. **Conditional Jump:** if x relop y goto L
6. **Stack Operations:** Push/pop
7. **Procedure:**

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

Index Assignments: x:=y[i], x[i]:=y

Address and Pointer Assignments: x:=&y, x:=*y, *x:=y

Implementations of 3-address statements

- Three address statement is an abstract form of intermediate code
- In a compiler, these statements can be implemented as records with fields for operator and operands
- Three representations
  - Quadruples, triples and indirect triples
- Quadruples
  - Record structure with 4 fields- *op, arg1, arg2 and result*
  - ***op*** field contains an internal code for operator
  - x=y op z represented by placing y in arg1, z in arg2, and x in result
  - x=-y or x=y do not use arg2
  - Operators like param use neither arg2 nor result
  - Conditional and unconditional jumps put the target label in result
  - Contents of arg1, arg2 and result are usually pointers to symbol entries

Quadruples

- There are following exceptions-
- **Exception-01:**
- To represent the statement x = op y, we place-
  - op in the operator field
  - y in the arg1 field
  - x in the result field
  - arg2 field remains unused
- **Exception-02:**
- To represent the statement like param t1, we place-
  - param in the operator field
  - t1 in the arg1 field
  - Neither arg2 field nor result field is used
- **Exception-03:**
- To represent the unconditional and conditional jump statements, we place label of the target in the result field.

Implementations of 3-address statements

- **Quadruples**

t_1:= - c
t_2:= b * t_1
t_3:= - c
t_4:= b * t_3
t_5:= t_2 + t_4
a:= t_5

	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

Temporary names must be entered into the symbol table as they are created.

Implementations of 3-address statements

- **Triples**

t_1:=- c

t_2:=b * t_1

t_3:=- c

t_4:=b * t_3

t_5:=t_2 + t_4

a:=t_5

	op	*arg1*	*arg2*
(0)	uminus	c	
(1)	*	b	(0)
(2)	uminus	c	
(3)	*	b	(2)
(4)	+	(1)	(3)
(5)	assign	a	(4)

Temporary names are not entered into the symbol table.

Refer to the temporary value by the position of the statement that computes it.

Other types of 3-address statements

- x[i]:=y

	op	*arg1*	*arg2*
(0)	[ ]	x	i
(1)	assign	(0)	y

- x:=y[i]

	op	*arg1*	*arg2*
(0)	[ ]	y	i
(1)	assign	x	(0)

Implementations of 3-address statements

- Indirect Triples-lists pointers to triples

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

	op	*arg1*	*arg2*
(14)	uminus	c	
(15)	*	b	(14)
(16)	uminus	c	
(17)	*	b	(16)
(18)	+	(15)	(17)
(19)	assign	a	(18)

Indirect Triples

- This representation is an enhancement over triples representation.
- It uses an additional instruction array to list the pointers to the triples in the desired order.
- Requires less space than quadruples.
- Temporaries are implicit and easier to rearrange code
- Thus, instead of position, pointers are used to store the results.
- It allows the optimizers to easily re-position the sub-expression for producing the optimized code