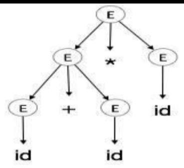
UNIT IV SYNTAX DIRECTED TRANSLATION & RUN TIME ENVIRONMENT

Syntax directed Definitions – Construction of Syntax Tree-Bottom-up Evaluation of S -Attribute Definitions-Design of predictive translator – Type Systems-Specification of a simple type checker- Equivalence of Type Expressions-Type Conversions.

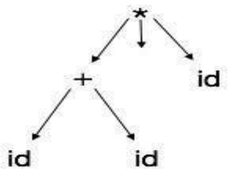
RUN-TIME ENVIRONMENT: Source Language Issues-Storage Organization-Storage Allocation- Parameter Passing-Symbol Tables-Dynamic Storage Allocation-Storage Allocation in FORTAN.

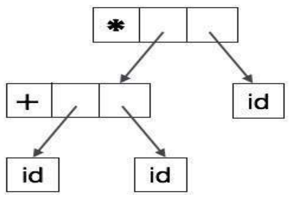
Parse Tree

Production rule: **E → E + E * E**



Abstract Syntax Tree





Issues in Syntax Tree

- A parser constructs parse trees in the syntax analysis phase.
- The plain parse-tree constructed in that phase is generally of no use for a compiler
- It does not carry any information of how to evaluate the tree.
- The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them.
- **Example**

  int a = "value";

It is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs.

Need for Semantics Analysis

- The rules are set by the grammar of the language and evaluated in semantic analysis.
- The following tasks should be performed in semantic analysis:
  - Scope resolution
  - Type checking
  - Array-bound checking

Semantic Errors

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Attribute Grammar

- Attribute grammar is a special form of context-free grammar where some additional information (attributes) are appended to one or more of its non-terminals in order to provide context-sensitive information.
- Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions.
- Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language.
- Example:

  E → E + T { E.value = E.value + T.value }

Syntax-Directed Definitions

- A syntax-directed definition (SDD) is a context-free grammar with attributes attached to grammar symbols and semantic rules attached to the productions.
- The semantic rules define values for attributes associated with the symbols of the productions.
- These values can be computed by creating a parse tree for the input and then making a sequence of passes over the parse tree, evaluating some or all of the rules on each pass.
- SDDs are useful for specifying translations.

CFG + semantic rules = Syntax Directed Definitions

Types of Attributes

- Synthesized attributes
- Inherited attributes
- S-attributed SDT
- L-attributed SDT

Synthesized attributes

These attributes get values from the attribute values of their child nodes.

- Example

1. 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.

2. E → E + T

The parent node E gets its value from its child node E and T.

Example

```
 E → E1 + T    { E.val = E1.val + T.val; }
E → T         { E.val = T.val; }
T → ( E )     { T.val = E.val; }
T → digit     { T.val = digit.lexval; }
```

Inherited attributes

- inherited attributes can take values from parent and/or siblings.
- Example

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.

Example

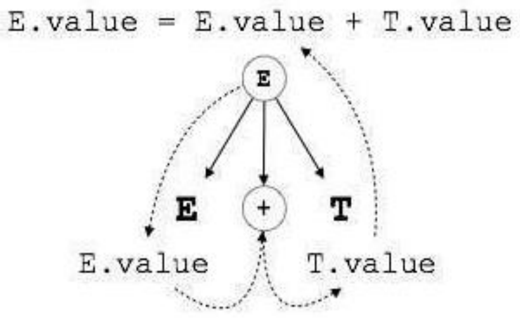
```
 E → T A       { E.node = A.s;
                 A.i = T.node; }
A → + T A1     { A1.i = Node('+', A.i, T.node);
                  A.s = A1.s; }
A → e         { A.s = A.i; }

T → ( E )     { T.node = E.node; }

T → id        { T.node = Leaf(id, id.entry); }
```

S-attributed SDT

- If an SDT uses only synthesized attributes, it is called as S-attributed SDT.
- These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side).

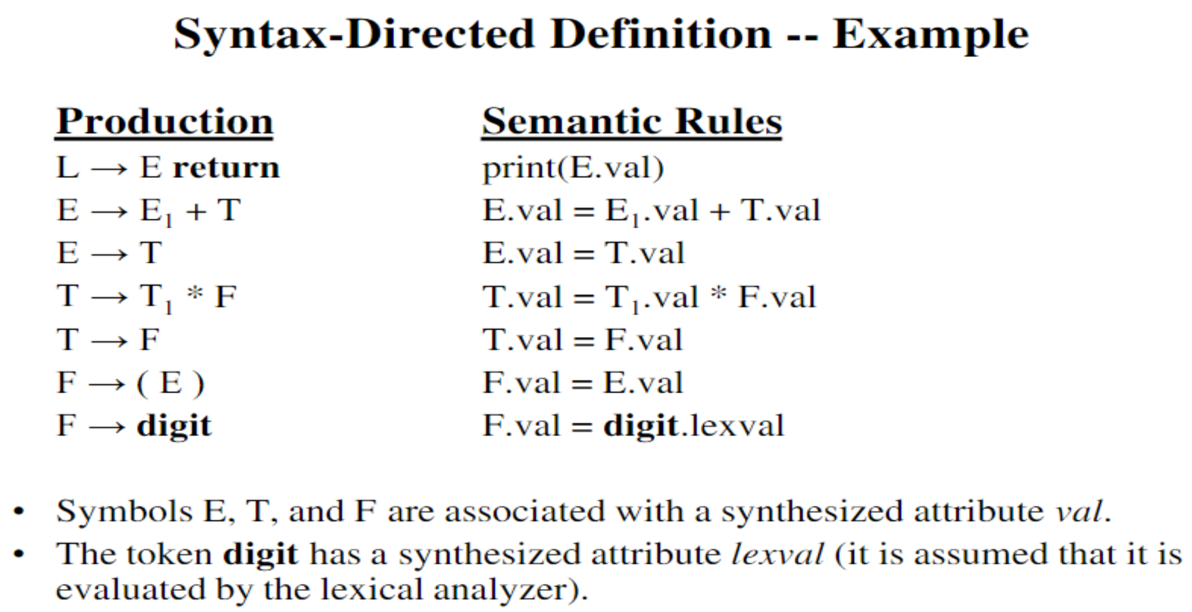


L-attributed SDT

- This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings.
- Example

$$S \rightarrow ABC$$

S can take values from A, B, and C (synthesized). A can take values from S only. B can take values from S and A. C can get values from S, A, and B. No non-terminal can get values from the sibling to its right.



Syntax-Directed Definition -- Example

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

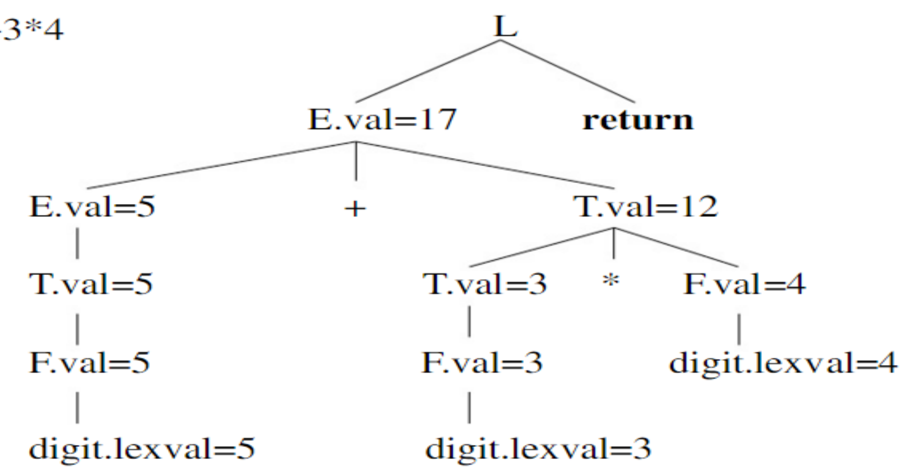
- Symbols E, T, and F are associated with a synthesized attribute *val*.
- The token **digit** has a synthesized attribute *lexval* (it is assumed that it is evaluated by the lexical analyzer).

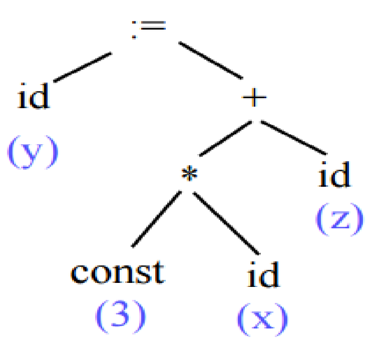
Order of evaluation

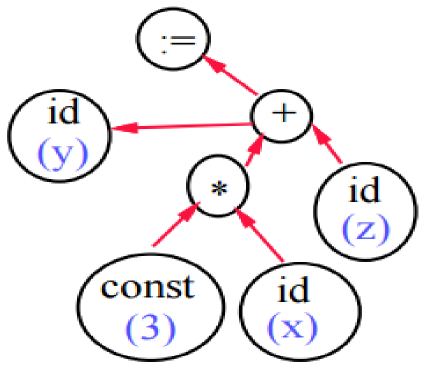
- General rule for ordering is based on Dependency graph
  - If attribute b needs attributes a and c, then a and c must be evaluated before b.
  - Represented as a directed graph without cycles.
  - Topologically order nodes in the dependency graph as n1, n2, . . . , nk such that there is no path from ni to nj with i > j.

Annotated Parse Tree -- Example

Input: 5+3*4

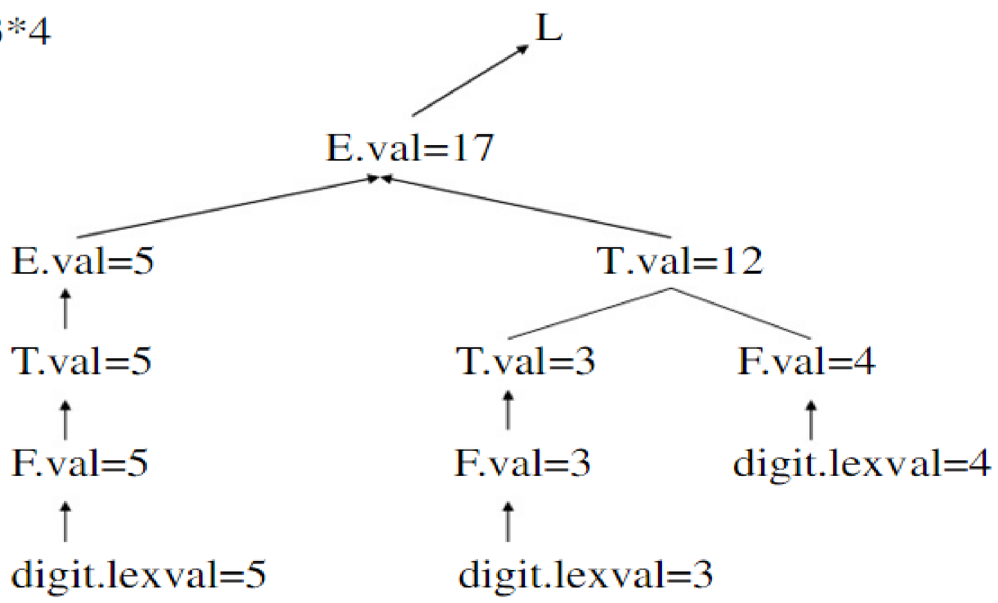






Dependency Graph

Input: 5+3*4



Construction of Syntax Trees

- Syntax trees are useful for representing programming language constructs like expressions and statements.
- They help compiler design by decoupling parsing from translation.
- Each node of a syntax tree represents a construct; the children of the node represent the meaningful components of the construct.

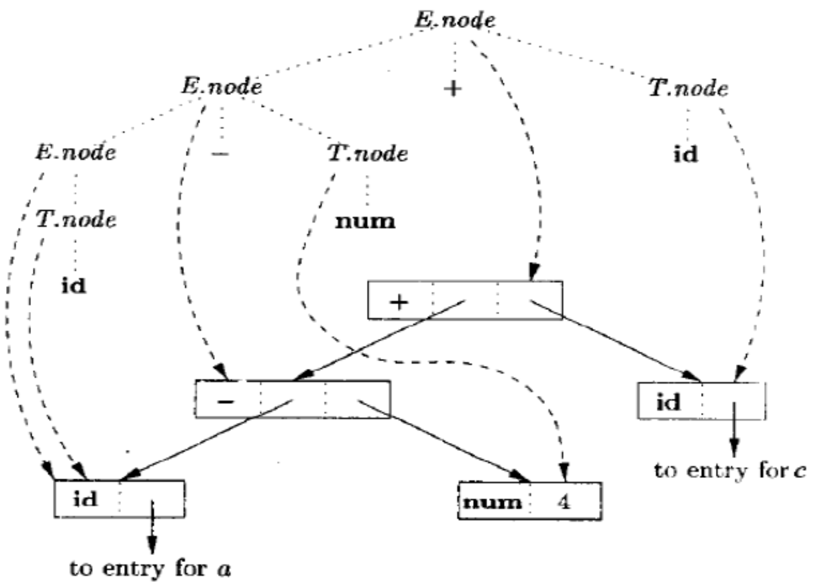
e.g. a syntax-tree node representing an expression E1 + E2 has label + and two children representing the sub expressions E1 and E2

- Each node is implemented by objects with suitable number of fields; each object will have an op field that is the label of the node with additional fields as follows:
    - If the node is a leaf, an additional field holds the lexical value for the leaf . This is created by function Leaf(op, val).
    - If the node is an interior node, there are as many fields as the node has children in the syntax tree. This is created by function Node(op, c1, c2,...,ck) .

Example:

Production	Semantic Rules
1) $E \rightarrow E_1 + T$	$E.node$ = **new** $Node$ ('+', $E_1.node$, $T.node$)
2) $E \rightarrow E_1 - T$	$E.node$ = **new** $Node$ ('-', $E_1.node$, $T.node$)
3) $E \rightarrow T$	$E.node = T.node$
4) $T \rightarrow (E)$	$E.node = T.node$
5) $T \rightarrow$ **id**	$T.node$ = **new** $Leaf$ (**id**, **id**.$entry$)
6) $T \rightarrow$ **num**	$T.node$ = **new** $Leaf$ (**num**, **num**.val)

Syntax tree for a-4+c using the above SDD



- If the rules are evaluated during a post order traversal of the parse tree, or with reductions during a bottom-up parse, then the sequence of steps shown below ends with p5 pointing to the root of the constructed syntax tree.

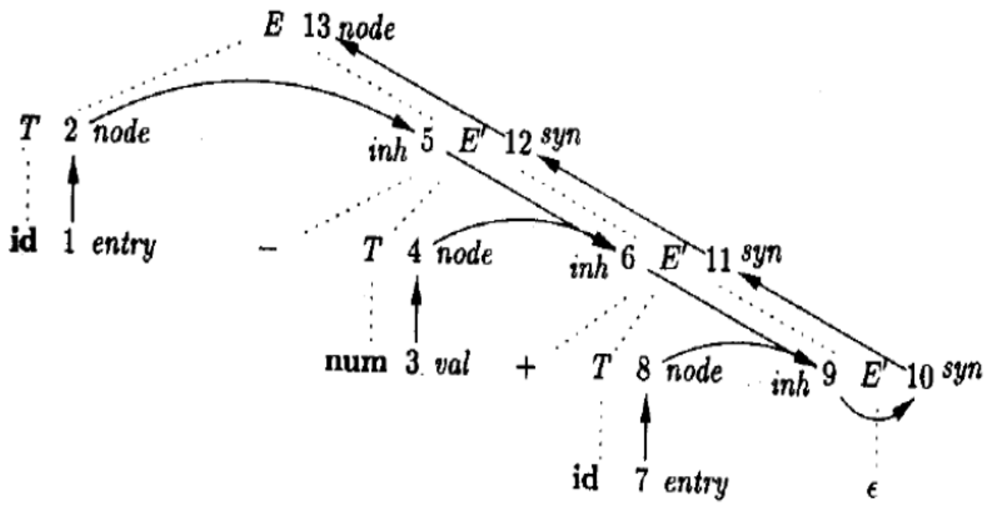
1) p_1 = **new** $Leaf$ (**id**, $entry\text{-}a$);
2) p_2 = **new** $Leaf$ (**num**, 4);
3) p_3 = **new** $Node$ ('–', p_1, p_2);
4) p_4 = **new** $Leaf$ (**id**, $entry\text{-}c$);
5) p_5 = **new** $Node$ ('+', p_3, p_4);

Constructing Syntax Trees during Top-Down Parsing

- With a grammar designed for top-down parsing, the same syntax trees are constructed, using the same sequence of steps, even though the structure of the parse trees differs significantly from that of syntax trees.

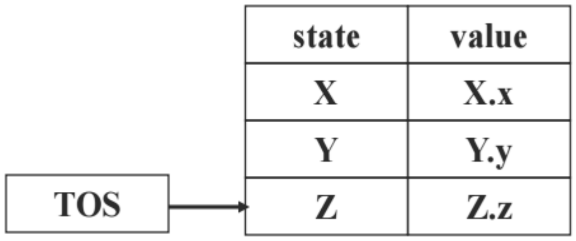
Production	Semantic Rules
1) $E \rightarrow T E'$	$E.node = E'.syn$ $E'.inh = T.node$
2) $E' \rightarrow + T E_1'$	$E_1'.inh =$ **new** $Node$ ('+', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
3) $E' \rightarrow - T E_1'$	$E_1'.inh =$ **new** $Node$ ('−', $E'.inh$, $T.node$) $E'.syn = E_1'.syn$
4) $E' \rightarrow \varepsilon$	$E'.syn = E'.inh$
5) $T \rightarrow (E)$	$T.node = E.node$
6) $T \rightarrow$ **id**	$T.node =$ **new** $Leaf$ (**id**, **id**.$entry$)
7) $T \rightarrow$ **num**	$T.node =$ **new** $Leaf$ (**num**, **num**.val)

Dependency Graph for a-4+c with L-Attributed SDD



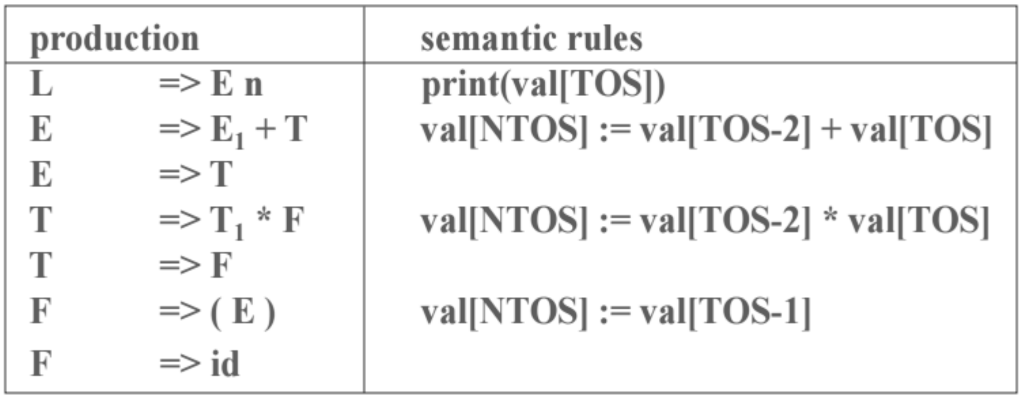
Bottom Up Evaluation of S-attributed Definitions

- Syntax-directed definitions with only synthesised attributes
- can be evaluated by a bottom up parser (BUP) as input is parsed
- values associated with the attributes can be kept on the stack as extra fields
- implementation using an LR parser (e.g. YACC)
- e.g. A => XYZ and A.a := f (X.x, Y.y, Z.z)

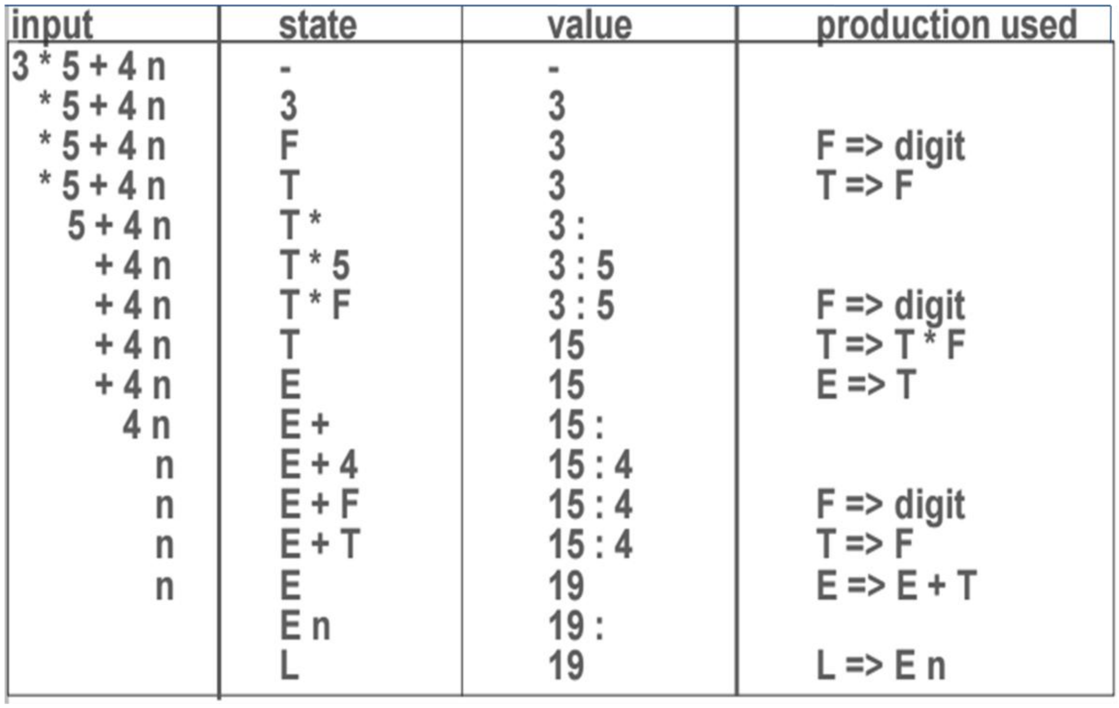


state	value
X	X.x
Y	Y.y
Z	Z.z

S-attributed Definitions: Parser Example



production		semantic rules
L	=> E n	print(val[TOS])
E	=> E_1 + T	val[NTOS] := val[TOS-2] + val[TOS]
E	=> T	
T	=> T_1 * F	val[NTOS] := val[TOS-2] * val[TOS]
T	=> F	
F	=> (E)	val[NTOS] := val[TOS-1]
F	=> id	

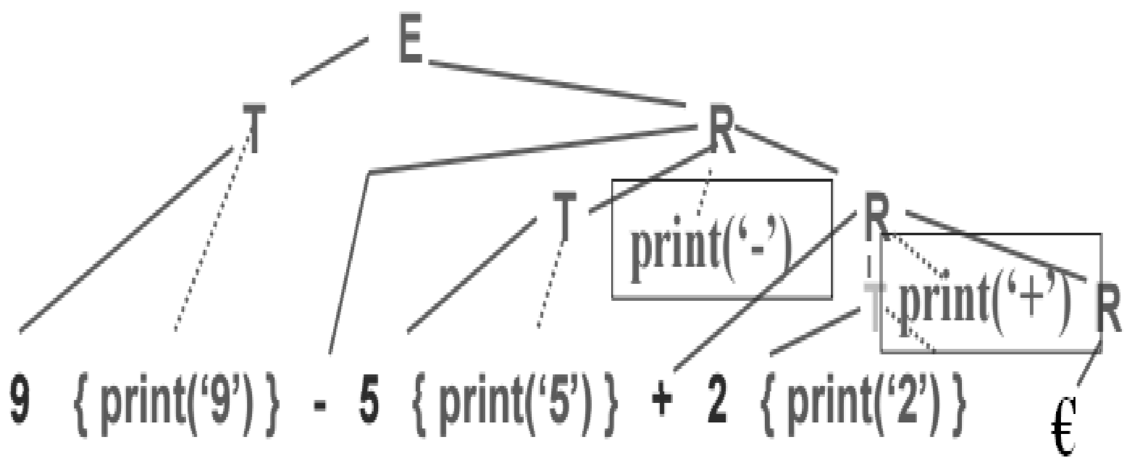


input	state	value	production used
3 * 5 + 4 n	-	-	
* 5 + 4 n	3	3	
* 5 + 4 n	F	3	F => digit
* 5 + 4 n	T	3	T => F
5 + 4 n	T *	3 :	
+ 4 n	T * 5	3 : 5	
+ 4 n	T * F	3 : 5	F => digit
+ 4 n	T	15	T => T * F
+ 4 n	E	15	E => T
4 n	E +	15 :	
n	E + 4	15 : 4	
n	E + F	15 : 4	F => digit
n	E + T	15 : 4	T => F
n	E	19	E => E + T
	E n	19 :	
	L	19	L => E n

Translation Scheme

- A context free grammar in which the attributes are associated with grammar symbols and semantic actions enclosed within { } are inserted within the RHS of productions
- example of a translation scheme + parse tree for 9-5+2 => 95-2+

E => T R
R => addop T { print(addop.lexeme) } R1 | €
T => num { print(num.val) }



Prdective Parser Translation

- Implementation of L-attribute definitions during predictive parsing using left recursion grammars and left recursion elimination algorithms
- e.g. translation schema with left recursive grammar

E	=>	E_1 + T	{ E.val := E_1.val + T.val }
E	=>	E_1 - T	{ E.val := E_1.val - T.val }
E	=>	T	{ E.val := T.val }
T	=>	(E)	{ T.val := E.val }
T	=>	num	{ T.val := num.val }

- transformed translation scheme with right recursive grammar (i.e. grammar + interleaved semantic actions)

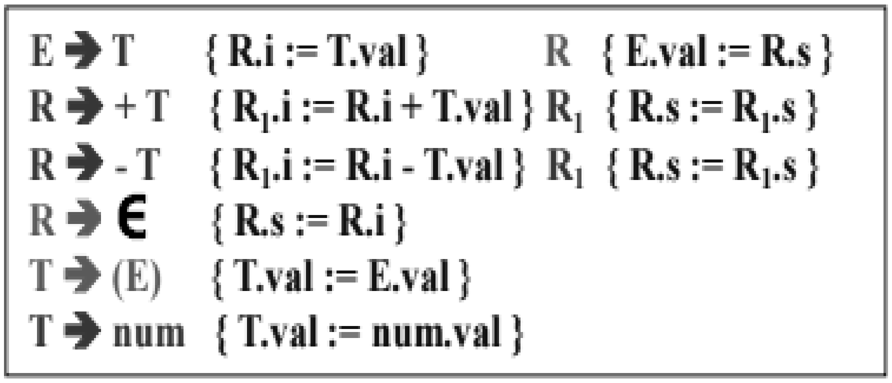
1. E ➔ T R
2. R ➔ + T R_1
3. R ➔ - T R_1
4. R ➔ ϵ
5. T ➔ (E)
6. T ➔ num

1. E ➔	T	{ R.i := T.val }	R	{ E.val := R.s }
2. R ➔	+ T	{ R_1.i := R.i + T.val }	R_1	{ R.s := R_1.s }
3. R ➔	- T	{ R_1.i := R.i - T.val }	R_1	{ R.s := R_1.s }
4. R ➔	ϵ	{ R.s := R.i }		
5. T ➔	(E)	{ T.val := E.val }		
6. T ➔	num	{ T.val := num.val }		

X.i - inherited attribute
X.s - synthesised attribute

Top-Down Translation

- Evaluation of the expression 9-5+2



$E \rightarrow T \quad \{ R.i := T.val \} \quad R \quad \{ E.val := R.s \}$

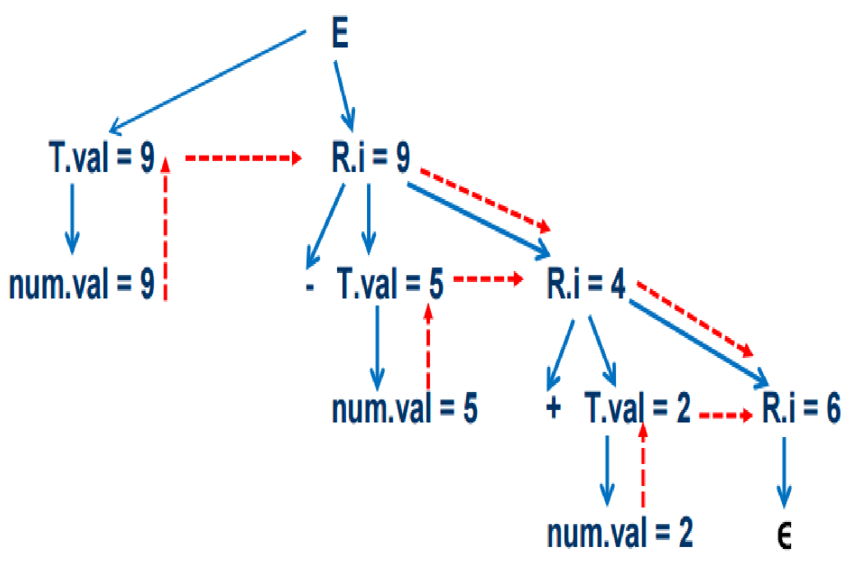
$R \rightarrow + T \quad \{ R_1.i := R.i + T.val \} \quad R_1 \quad \{ R.s := R_1.s \}$

$R \rightarrow - T \quad \{ R_1.i := R.i - T.val \} \quad R_1 \quad \{ R.s := R_1.s \}$

$R \rightarrow \epsilon \quad \{ R.s := R.i \}$

$T \rightarrow (E) \quad \{ T.val := E.val \}$

$T \rightarrow num \quad \{ T.val := num.val \}$



Summary

- syntax-directed translations - grammar + semantic rules
  - – synthesised attributes
  - – inherited attributes
  - – b := f(c1, c2, c3, …, ck)
- S-attributed definition
  - – only synthesised attributes
- dependency graphs
- constructing syntax trees
  - – syntax-directed definition
- bottom up evaluation of S-attributed definitions
- L-attributed definitions
  - – inherited attributes depend on A and left siblings A => α
- translation schemes
  - – CFG + attributes + semantic actions in { } in RHS of P
- top down translation
  - – left recursive grammar
  - – right recursive grammar
  - – X.i & X.s attributes

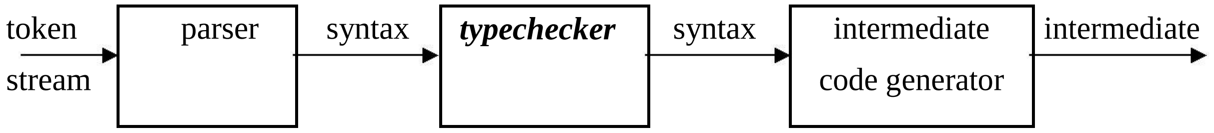
TYPE CHECKING

A compiler must check that the source program follows both syntactic and semantic conventions of the source language.
This checking, called *static checking,* detects and reports programming errors.

Some examples of static checks:

1. **Type checks** – A compiler should report an error if an operator is applied to an incompatible operand. Example: If an array variable and function variable are added together.
2. **Flow-of-control checks** – Statements that cause flow of control to leave a construct must have some place to which to transfer the flow of control. Example: An error occurs when an enclosing statement, such as break, does not exist in switch statement.

Position of type checker



- A ***type checker*** verifies that the type of a construct matches that expected by its context. For example : arithmetic operator *mod* in Pascal requires integer operands, so a type checker verifies that the operands of *mod* have type integer.
- Type information gathered by a type checker may be needed when code is generated.

TYPE SYSTEMS

The design of a type checker for a language is based on information about the syntactic constructs in the language, the notion of types, and the rules for assigning types to language constructs.

For example : " if both operands of the arithmetic operators of +,- and * are of type integer, then the result is of type integer "

Type Expressions

- The type of a language construct will be denoted by a "type expression."
- A type expression is either a basic type or is formed by applying an operator called a ***type constructor*** to other type expressions.
- The sets of basic types and constructors depend on the language to be checked.

The following are the definitions of type expressions:

1. Basic types such as *boolean, char, integer, real* are type expressions.

   A special basic type, *type_error* , will signal an error during type checking; *void* denoting "the absence of a value" allows statements to be checked.

2. Since type expressions may be named, a type name is a type expression.

3. A type constructor applied to type expressions is a type expression.
Constructors include:
Arrays : If T is a type expression then *array* (I,T) is a type expression denoting the type of an array with elements of type T and index set I.

Products : If T_1 and T_2 are type expressions, then their Cartesian product T_1 X T_2 is a type expression.

Records : The difference between a record and a product is that the fields of a record have names. The *record* type constructor will be applied to a tuple formed from field names and field types.
For example:

```
    type row = record
                    address: integer;
                    lexeme: array[1..15] of
                char end;
    var table: array[1...101] of row;
```

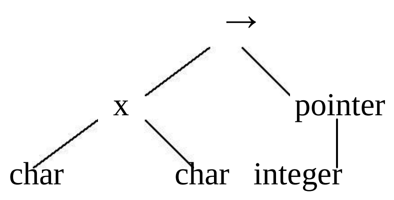
declares the type name *row* representing the type expression ***record((address X integer) X (lexeme X array(1..15,char)))*** and the variable *table* to be an array of records of this type.

Pointers : If T is a type expression, then *pointer*(T) is a type expression denoting the type "pointer to an object of type T".
For example, ***var p: ↑ row*** declares variable p to have type *pointer*(row).

Functions : A function in programming languages maps a *domain type D* to a *range type R*. The type of such function is denoted by the type expression D → R

4. Type expressions may contain variables whose values are type expressions.

Tree representation for char x char → *pointer* (integer)



Type systems

- A *type system* is a collection of rules for assigning type expressions to the various parts of a program.
- A type checker implements a type system. It is specified in a syntax-directed manner.
- Different type systems may be used by different compilers or processors of the same language.

Static and Dynamic Checking of Types

- Checkingdone by a compiler is said to be static, while checking done when the target program runs is termed dynamic.

- Any check can be done dynamically, if the target code carries the type of an element along with the value of that element.

- **Sound type system**

A *sound* type system eliminates the need for dynamic checking for type errors because it allows us to determine statically that these errors cannot occur when the target program runs. That is, if a sound type system assigns a type other than *type_error* to a program part, then type errors cannot occur when the target code for the program part is run.

Strongly typed language

A language is strongly typed if its compiler can guarantee that the programs it accepts will execute without type errors.

Error Recovery

➢ Since type checking has the potential for catching errors in program, it is desirable for type checker to recover from errors, so it can check the rest of the input.

➢ Error handling has to be designed into the type system right from the start; the type checking rules must be prepared to cope with errors.

SPECIFICATION OF A SIMPLE TYPE CHECKER

Here, we specify a type checker for a simple language in which 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 types of its subexpressions. The type checker can handle arrays, pointers, statements and functions.

<u>A Simple Language</u>

Consider the following grammar:

P → D ; E
D → D ; D | id : T
T → char | integer | array [num] of T | ↑ T
E → literal | num | id | E mod E | E [E] | E ↑

Translation scheme:

P → D ; E	
D → D ; D	
D → id : T	{ *addtype* (id.*entry* , T.*type*)}
T → char	{ T.*type* : = char }
T → integer	{ T.*type* : = integer }
T → ↑ T1	{ T.*type* : = pointer(T_1.*type*) }
T → array [num] of T1	{ T.*type* : = array (1… num.val , T_1.*type*) }

In the above language,
→ There are two basic types : char and integer ;
→ *type_error* is used to signal errors;

→ the prefix operator ↑ builds a pointer type. Example , ↑ **integer** leads to the type expression **pointer (integer)**.

Type checking of expressions

In the following rules, the attribute *type* forE gives the type expression assigned to the expression generated by E.

1. E → **literal** { E.*type* : = *char* }
   E → **num** { E.*type* : = *integer* }
   Here, constants represented by the tokens **literal** and **num** have type *char* and *integer*.

2. E → **id** { E.*type* : = *lookup* (**id**.*entry*) }
   lookup (e) is used to fetch the type saved in the symbol table entry pointed to by e.

3. E → E_1 **mod** E_2 { *E.type* : = **if** *E_1. type* = *integer* **and**
   E_2. type = *integer* **then**
   integer **else** *type_error* }
   The expression formed by applying the mod operator to two subexpressions of type integer has type integer; otherwise, its type is *type_error*.

4. E → E_1 [E_2] { *E.type* : = if *E_2.type* = *integer* **and**
   E_1.type = *array(s,t)* **then** *t*
   else *type_error* }
   In an array reference E_1 [E_2] , the index expression E 2 must have type integer. The result is the element type *t* obtained from the type *array(s,t)* of E_1.

5. E → E_1 ↑ { *E.type* : = **if** *E_1.type* = *pointer (t)* **then** *t*
   else *type_error* }

   The postfix operator ↑ yields the object pointed to by its operand. The type of E ↑ is the type *t* of the object pointed to by the pointer E.

Type checking of statements

Statements do not have values; hence the basic type *void* can be assigned to them. If an error is detected within a statement, then *type_error* is assigned.

Translation scheme for checking the type of statements:

1. Assignment statement:

S → **id** : = E { S.*type* : = **if id**.*type* = E.*type* **then** *void* **else**
type_error }

2. Conditional statement:

S → **if** E **then** S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

3. While statement:

S → while E do S_1 { S.*type* : = **if** E.*type* = *boolean* **then** S_1.*type*
else *type_error* }

4. Sequence of statements:

$S \rightarrow S_1 ; S_2$ { $S.\mathit{type}$: = **if** $S_1.\mathit{type} = \mathit{void}$ and $S_1.\mathit{type} = \mathit{void}$
then *void*
else *type_error* }

<u>Type checking of functions</u>

The rule for checking the type of a function application is :

$E \rightarrow E_1 (E_2)$ { $E.\mathit{type}$: = **if** $E_2.\mathit{type} = s$ **and**
$E_1.\mathit{type} = s \rightarrow t$ **then**
t **else** *type_error* }

SOURCE LANGUAGE ISSUES

Procedures:

A *procedure definition is a declaration that associates an identifier with a* statement. The identifier is the *procedure name*, and the statement is the *procedure body*.
For example, the following is the definition of procedure named *readarray* :

```
 procedure readarray;
var i : integer;
 begin
      for i : = 1 to 9 do
read(a[i])  end;
```

When a procedure name appears within an executable statement, the procedure is said to be *called* at that point.

Activation trees:

An *activation tree* is used to depict the way control enters and leaves activations. In an activation tree,

1. Each node represents an activation of a procedure.
2. The root represents the activation of the main program.
3. The node for *a* is the parent of the node for *b* if and only if control flows from activation *a* to *b*.
4. The node for *a* is to the left of the node for *b* if and only if the lifetime of *a* occurs before the lifetime of *b*.

Control stack:

- A *control stack* is used 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 to pop the node when the activation ends.
- The contents of the control stack are related to paths to the root of the activation tree. When node *n* is at the top of control stack, the stack contains the nodes along the path from *n* to the root.

The Scope of a Declaration:
A declaration is a syntactic construct that associates information with a name. Declarations may be explicit, such as:

    var i : integer ;

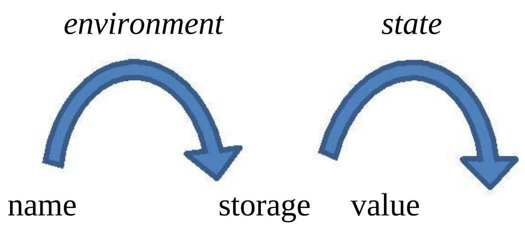
or they may be implicit. Example, any variable name starting with I is assumed to denote an integer.

The portion of the program to which a declaration applies is called the ***scope*** of that declaration.

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 holds 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.

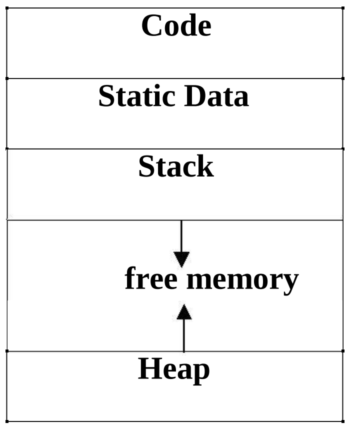


When an *environment* associates storage location *s* with a name *x*, we say that *x* is *bound* to *s*. This association is referred to as a *binding* of *x*.

STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

Typical subdivision of run-time memory:



- Run-time storage comes in blocks, where a byte is the smallest unit of addressable memory. Four bytes form a machine word. Multibyte objects are stored in consecutive bytes and given the address of first byte.
- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.

Activation record
Actual Parameters
Returned Values
Control Link
Access Links
Saved Machine states
Local Data
Temporaries

- Temporary values such as those arising from the evaluation of expressions.
- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

- Space for the return value of the called functions, if any. Again, not all called procedures return a value, and if one does, we may prefer to place that value in a register for efficiency.
- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation** – lays out storage for all data objects at compile time
2. **Stack allocation** – manages the run-time storage as a stack.
3. **Heap allocation** – allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

- In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.
- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

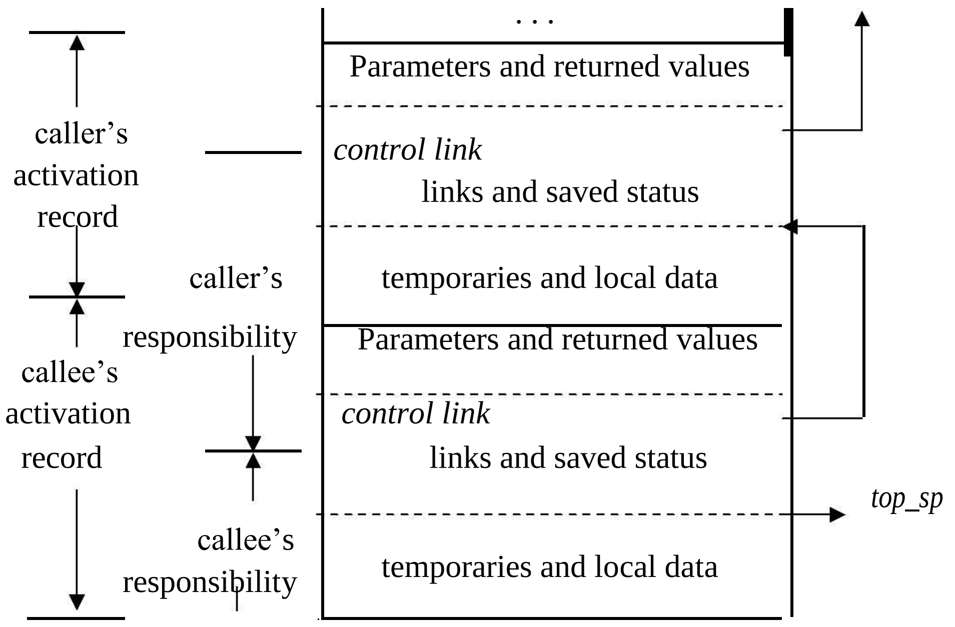
STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consists of code that allocates an activation record on the stack and enters information into its fields.
- A return sequence is similar to code to restore the state of machine so the calling procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:
  - ➢ Values communicated between caller and callee are generally placed at the beginning of the callee’s activation record, so they are as close as possible to the caller’s activation record.
- ➢ Fixed length items are generally placed in the middle. Such items typically include the control link, the access link, and the machine status fields.
- ➢ Items whose size may not be known early enough are placed at the end of the activation record. The most common example is dynamically sized array, where the value of one of the callee’s parameters determines the length of the array.
- ➢ We must locate the top-of-stack pointer judiciously. A common approach is to have it point to the end of fixed-length fields in the activation record. Fixed-length data can then be

accessed by fixed offsets, known to the intermediate-code generator, relative to the top-of-stack pointer.



. . .
Parameters and returned values
control link
links and saved status
temporaries and local data
Parameters and returned values
control link
links and saved status
temporaries and local data
caller's activation record
caller's responsibility
callee's activation record
callee's responsibility
top_sp

Division of tasks between caller and callee

- The calling sequence and its division between caller and callee are as follows.

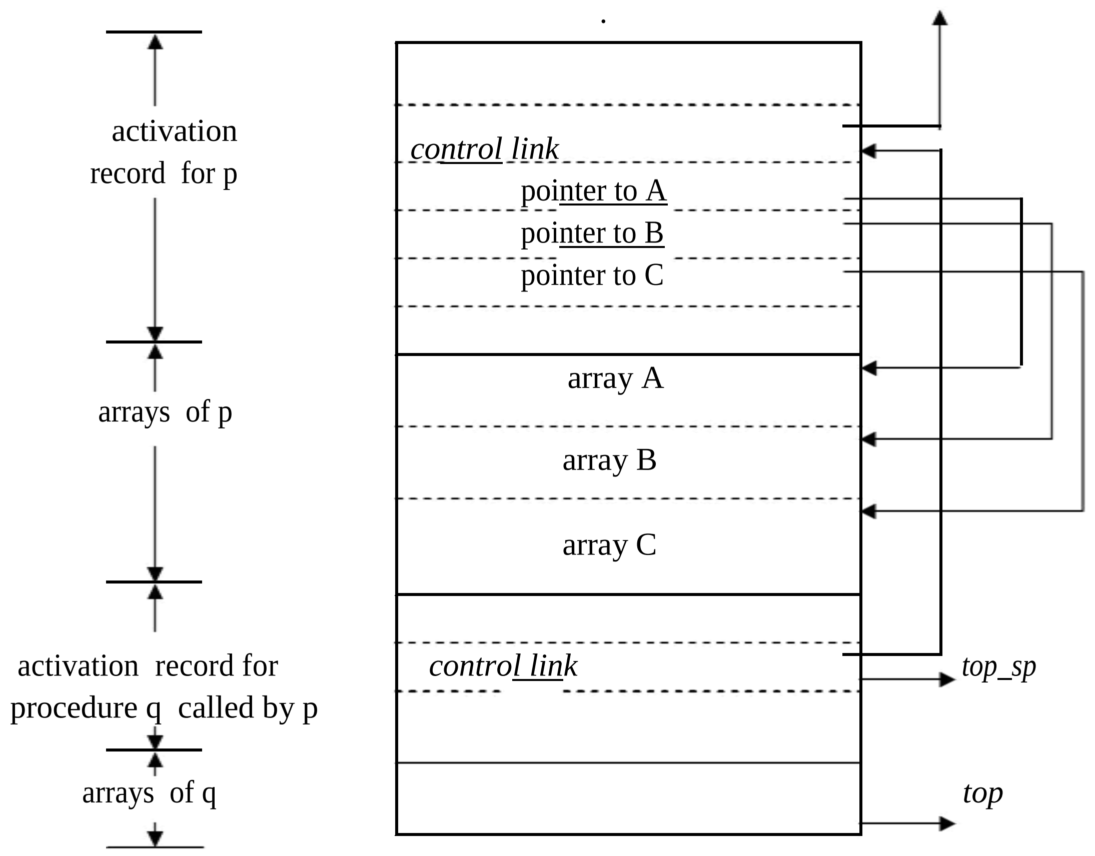
  - The caller evaluates the actual parameters.
  - The caller stores a return address and the old value of *top_sp* into the callee's activation record. The caller then increments the *top_sp* to the respective positions.
  - The callee saves the register values and other status information.
  - The callee initializes its local data and begins execution.

- A suitable, corresponding return sequence is:

  - The callee places the return value next to the parameters.
  - Using the information in the machine-status field, the callee restores *top_sp* and other registers, and then branches to the return address that the caller placed in the status field.
  - Although *top_sp* has been decremented, the caller knows where the return value is, relative to the current value of *top_sp*; the caller therefore may use that value.

Variable length data on stack:

- The run-time memory management system must deal frequently with the allocation of space for objects, the sizes of which are not known at the compile time, but which are local to a procedure and thus may be allocated on the stack.
- The reason to prefer placing objects on the stack is that we avoid the expense of garbage collecting their space.
- The same scheme works for objects of any type if they are local to the procedure called and have a size that depends on the parameters of the call.



Access to dynamically allocated arrays

- Procedure p has three local arrays, whose sizes cannot be determined at compile time. The storage for these arrays is not part of the activation record for p.
- Access to the data is through two pointers, *top* and *top-sp*. Here the *top* marks the actual top of stack; it points the position at which the next activation record will begin.
- The second *top-sp* is used to find local, fixed-length fields of the top activation record.
- The code to reposition *top* and *top-sp* can be generated at compile time, in terms of sizes that will become known at run time.

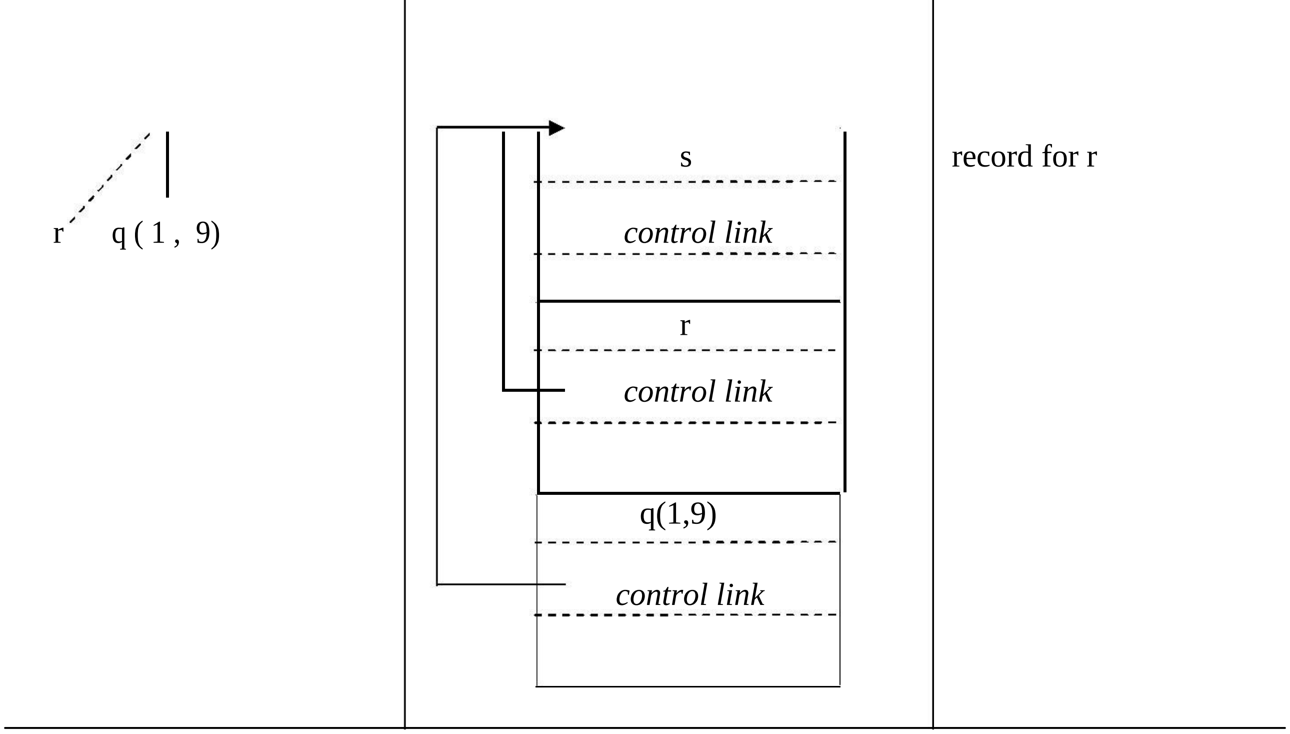
<u>HEAP ALLOCATION</u>

Stack allocation strategy cannot be used if either of the following is possible :

1. The values of local names must be retained when an activation ends.
2. A called activation outlives the caller.

- Heap allocation parcels out pieces of contiguous storage, as needed for activation records or other objects.
- Pieces may be deallocated in any order, so over the time the heap will consist of alternate areas that are free and in use.

Position in the activation tree	Activation records in the heap	Remarks
s		Retained activation



- The record for an activation of procedure r is retained when the activation ends.
- Therefore, the record for the new activation q(1 , 9) cannot follow that for s physically.
- If the retained activation record for r is deallocated, there will be free space in the heap between the activation records for s and q.