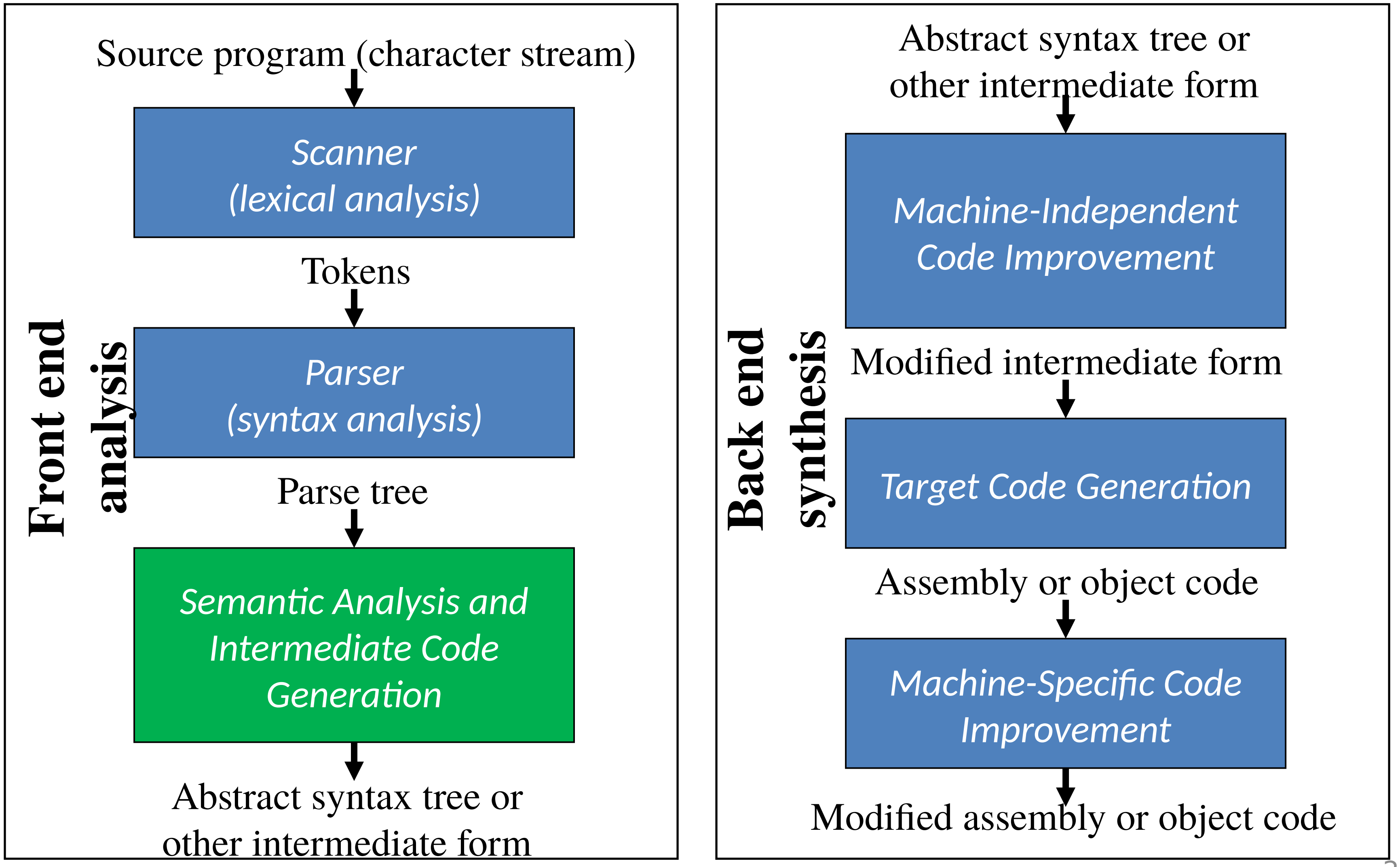
Compiler Phases

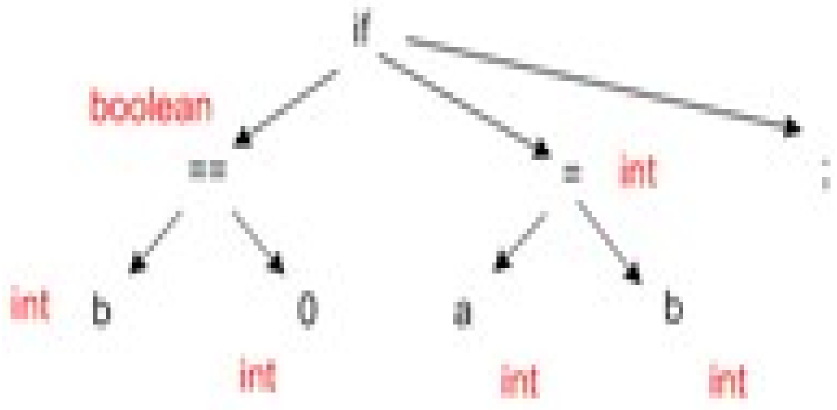
1. Lexical analysis
2. Syntax analysis
3. Semantic analysis
4. Intermediate (machine-independent) code generation
5. Intermediate code optimization
6. Target (machine-dependent) code generation
7. Target code optimization

Compiler Front- and Back-end


Source program (character stream)
Scanner (lexical analysis)
Tokens
Parser (syntax analysis)
Parse tree
Semantic Analysis and Intermediate Code Generation
Abstract syntax tree or other intermediate form
Front end analysis
Abstract syntax tree or other intermediate form
Machine-Independent Code Improvement
Modified intermediate form
Target Code Generation
Assembly or object code
Machine-Specific Code Improvement
Modified assembly or object code
Back end synthesis

Semantics analysis

1. Check semantics
2. Error reporting
3. Disambiguate overloaded operators
4. Type coercion
5. Static checking :
   a) Type checking
   b) Control flow checking
   c) Uniqueness checking
   d) Name checks



Semantics analysis

- Semantic analysis is a pass by a compiler that adds semantic information to the parse tree and performs certain checks based on this information.

- It logically follows the parsing phase, in which the parse tree is generated, and logically precedes the code generation phase, in which (intermediate / target) code is generated. (In a compiler implementation, it may be possible to fold different phases into one pass.)

- Typical examples of semantic information that is added and checked is
  - typing information (type checking) and
  - the binding of variables and function names to their definitions

    (object binding).

- Sometimes also some early code optimization is done in this phase. For this phase the compiler usually maintains *symbol tables* in which it stores what each symbol (variable names, function names, etc.) refers to.

Things are done in Semantic Analysis:

- **Disambiguate Overloaded operators** : If an operator is overloaded, one would like to specify the meaning of that particular operator because from one will go into code generation phase next.

- **Type checking** : The process of verifying and enforcing the constraints of types is called type checking. This may occur either at compile-time (a static check) or run-time (a dynamic check). Static type checking is a primary task of the semantic analysis carried out by a compiler.

- **Uniqueness checking** : Whether a variable name is unique or not, in the its scope.

- **Type coercion** : If some kind of mixing of types is allowed which is generally done in the languages which are not strongly typed. This can be done dynamically as well as statically.

- **Name Checks** : Check whether any variable has a name which is not allowed. Ex. Name is same as an identifier(Ex. int in java).

Beyond Parsing

- Parser cannot catch all the program errors
- There is a level of correctness that is deeper than syntax analysis
- Some language features cannot be modeled using context free grammar formalism
  - Whether an identifier has been declared before use
- A parser has its own limitations in catching program errors related to semantics, something that is deeper than syntax analysis.
- Typical features of semantic analysis cannot be modeled using context free grammar formalism.
- If one tries to incorporate those features in the definition of a language then that language doesn't remain context free anymore.

Beyond Parsing

- Example 1
  - string x; int y;
  - y = x + 3
  - the use of x is a type error

- Example 2
  - Int a, b;
  - a = b + c
  - c is not declared

- Example 3
  - An identifier may refer to different variables in different parts of the program

  An identifier x can be declared in two separate functions in the program, once of the type int and then of the type char. Hence the same identifier will have to be bound to these two different properties in the two different contexts.

- Example4:
  - An identifier may be usable in one part of the program but not another

  A variable declared within one function cannot be used within the scope of the definition of the other function unless declared there separately (in another scope).

What does a compiler need to know during semantic analysis?

A few examples:

- Whether a variable has been declared?
- Are there variables which have not been declared?
- What is the type of the variable? (**int a, b , c;)**
- Whether a variable is a scalar, an array, or a function?
- What declaration of the variable does each reference use?
- If an expression is type consistent?
- If an array use like A[i,j,k] is consistent with the declaration? Does it have three dimensions?

Then we see that syntax analyzer cannot alone handle this situation. We actually need to traverse the parse trees to find out the type of identifier and this is all done in semantic analysis phase.

What does a compiler need to know during semantic analysis?

Few more examples:

- How many arguments does a function take?
- Are all invocations of a function consistent with the declaration?
- If an operator/function is overloaded, which function is being invoked?
- Inheritance relationship
- Classes not multiply defined
- Methods in a class are not multiply defined
- The exact requirements depend upon the language

If the compiler has the answers to all these questions only then will it be able to successfully do a semantic analysis by using the generated parse tree. These questions give a feedback to what is to be done in the semantic analysis. These questions help in outlining the work of the semantic analyzer.

How to answer these questions?

- Use formal methods
  - Context sensitive grammars
  - Extended attribute grammars

- Use ad-hoc techniques
  - Symbol table
  - Ad-hoc code

- Something in between !!!
  - Use attributes
  - Do analysis along with parsing - Use code for attribute value computation
  - However, code is developed in a systematic way

How to answer these questions?

- In syntax analysis we used context free grammar. Here we put lot of attributes around it. So it consists of context sensitive grammars along with extended attribute grammars.

- Ad-hoc methods also good as there is no structure in it and the formal method is simply just too tough. So we would like to use something in between.

- Formalism may be so difficult that writing specifications itself may become tougher than writing compiler itself.

- **So we do use attributes but we do analysis along with parse tree itself instead of using context sensitive grammars.**

Why attributes ?

- For lexical analysis and syntax analysis formal techniques were used.

- However, we still had code in form of actions along with regular expressions and context free grammar

- The attribute grammar formalism is important
  - However, it is very difficult to implement
  - But makes many points clear
  - Makes "ad-hoc" code more organized
  - Helps in doing non local computations

Attribute grammar is nothing but it is a CFG and attributes put around all the terminal and non-terminal symbols are used. Despite the difficulty in the implementation of the attribute grammar formalism it has certain big advantages which makes it desirable.

Attribute Grammar Framework

- Generalization of CFG where each grammar symbol has an associated set of attributes
- Values of attributes are computed by semantic rules
- Two notations for associating semantic rules with productions
  - Syntax directed definition
    - high level specifications
    - hides implementation details
    - explicit order of evaluation is not specified
  - Syntax directed translation schemes
    - indicate order in which semantic rules are to be evaluated
    - allow some implementation details to be shown

Attribute Grammar Framework

- There are two ways for writing attributes:
  - Syntax Directed Definition : A SDD specifies the values of attributes by associating semantic rules with the grammar rules. For example, an infix-to-postfix translator might have a production and rule
    - Production = { $E \rightarrow E_1 + T$}
    - Semantic rule = { $E.code = E_1.code \,||\, T.code \,||$ '+'}
  - Syntax-directed Translation scheme : A SDT scheme embeds program fragments called semantic actions within production bodies
    - $E \rightarrow E_1 + T$ {print '+'}

Attribute Grammar Framework

- Conceptually both:
  - parse input token stream
  - build parse tree
  - traverse the parse tree to evaluate the semantic rules at the parse tree nodes

- Evaluation may:
  - generate code
  - save information in the symbol table
  - issue error messages
  - perform any other activity

- To avoid repeated traversal of the parse tree, actions are taken simultaneously when a token is found. So calculation of attributes goes along with the construction of the parse tree.

- Along with the evaluation of the semantic rules the compiler may simultaneously generate code, save the information in the symbol table, and / or issue error messages etc. at the same time while building the parse tree.

- This saves multiple passes of the parse tree.

Syntax Directed Definition

1. A syntax-directed definition is a generalization of a context-free grammar in which:
   - Each grammar symbol is associated with a set of attributes.
   - This set of attributes for a grammar symbol is partitioned into two subsets called synthesized attributes and inherited attributes of that grammar symbol.
   - Each production rule is associated with a set of semantic rules.

2. Semantic rules set up dependencies between attributes which can be represented by a dependency graph.

3. This dependency graph determines the evaluation order of these semantic rules.

4. Evaluation of a semantic rule defines the value of an attribute. But a semantic rule may also have some side effects such as printing a value.

Example:

- Number → sign list
- sign → + | -
- list → list bit | bit
- bit → 0 | 1

- Build attribute grammar that annotates Number with the value it represents

- Associate attributes with grammar symbols

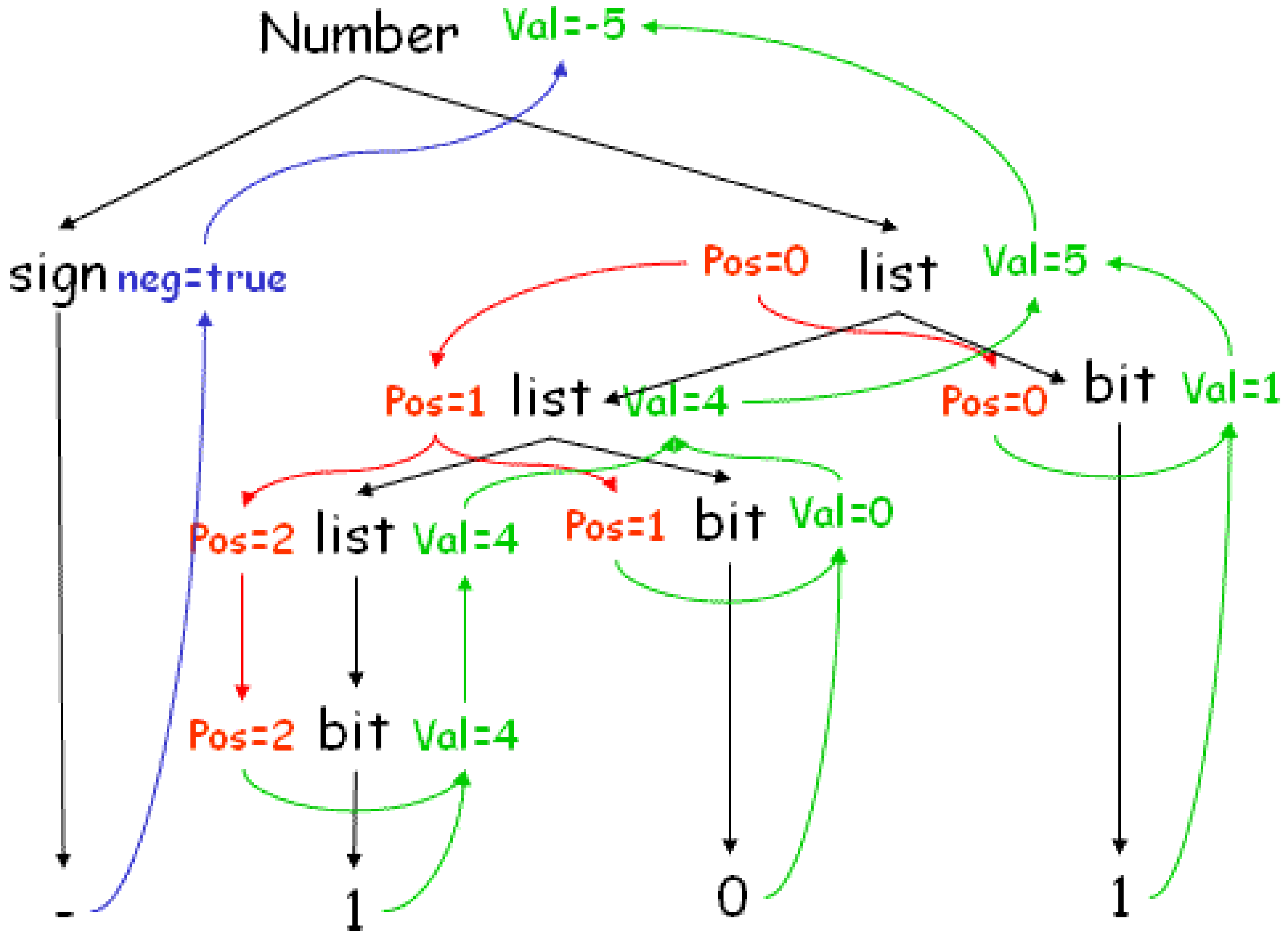
<u>symbol</u>	<u>attributes</u>
Number	value
sign	negative
list	position, value
bit	position, value

Example:

Production	Attribute rule	Explanations
number ▭ sign list	list.position ⩸ 0	/*since list is the rightmost so it is assigned position 0 */
	if sign.negative then number.value ⩸ - list.value else number.value ⩸ list.value	/*Sign determines whether the value of the number would be same or the negative of the value of list*/
sign ▭ +	sign.negative ⩸ false	/*Set the Boolean attribute (negative) for sign*/
sign ▭ -	sign.negative ⩸ true	
list ▭ bit	bit.position ⩸ list.position	/*bit position is the same as list position because this bit is the rightmost*/
	list.value ⩸ bit.value	/*value of the list is same as bit.*/
list$_0$ ▭ list $_1$ bit	list$_1$ .position ⩸ list $_0$ .position + 1	/*position and value calculations*/
	bit.position ⩸ list $_0$ .position	
	list$_0$ .value ⩸ list$_1$ .value + bit.value	
bit ▭ 0	bit.value ⩸ 0	/*set the corresponding value*/
bit ▭ 1	bit.value ⩸ $2^{\text{bit.position}}$	

Note: Attributes of RHS can be computed from attributes of LHS and vice versa.

Dependency Graph:

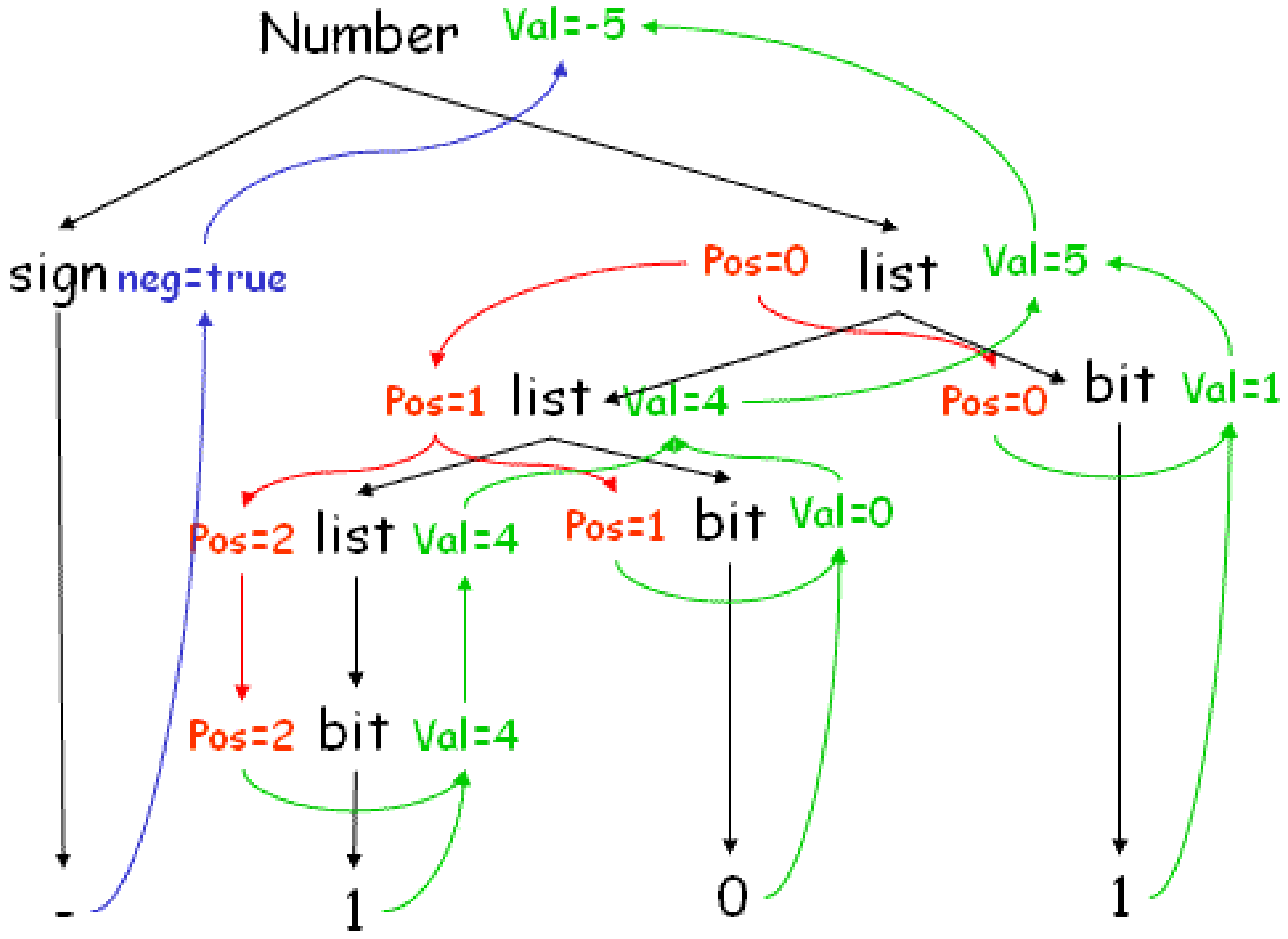


Dependence graph shows the dependence of attributes on other attributes, along with the syntax tree. Top down traversal is followed by a bottom up traversal to resolve the dependencies.

Attributes:

- attributes fall into two classes:
  - synthesized:
    - value of a synthesized attribute is computed from the values of its children nodes
  - inherited:
    - value of an inherited attribute is computed from the sibling and parent nodes
- The synthesized attributes are the result of the attribute evaluation rules also using the values of the inherited attributes.
- The values of the inherited attributes are inherited from parent nodes and siblings.

Dependency Graph:



Number, val and neg are synthesized attributes. Pos is an inherited attribute.

Attributes:

- Each grammar production $A \rightarrow \alpha$ has associated with it a set of semantic rules of the form

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

  where **f** is a function, and **b** can be one of the following:

- **b** is a synthesized attribute of **A** and $c_1, c_2, \ldots, c_k$ are attributes of the grammar symbols in the production $(A \rightarrow \alpha)$ **OR**

- **b** is an inherited attribute of one of the grammar symbols in α (on the right side of the production) and attribute **b** depends on attributes $c_1, c_2, \ldots, c_k$

- Dependence relation tells us what attributes we need to know before hand to calculate a particular attribute.

- Here the value of the attribute **b** depends on the values of the attributes c_1 to c_k. If c_1 to c_k belong to the children nodes and **b** to **A** then **b** will be called a synthesized attribute.

- And if **b** belongs to one among a (child nodes) then it is an inherited attribute of one of the grammar symbols on the right.

Synthesized Attributes

- A syntax directed definition that uses only synthesized attributes is said to be an S- attributed definition

- A parse tree for an S-attributed definition can be annotated by evaluating semantic rules for attributes

- S-attributed grammars are a class of attribute grammars, comparable with L-attributed grammars but characterized by having no inherited attributes at all.

- Attribute evaluation in S-attributed grammars can be incorporated conveniently in both top-down parsing and bottom-up parsing .

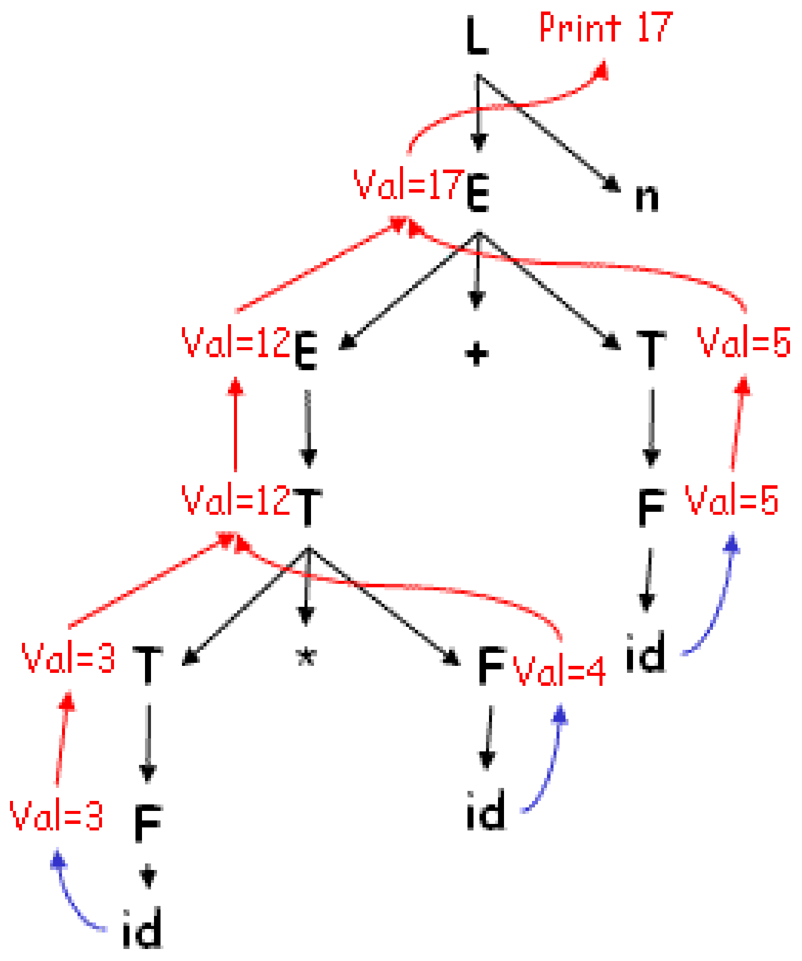
Syntax-Directed Definition -- Example

L E n	Print (E.val)
E E + T	E.val = E.val + T.val
E T	E.val = T.val
T T * F	T.val = T.val * F.val
T F	T.val = F.val
F (E)	F.val = E.val
F digit	F.val = digit.lexval

- Terminals are assumed to have only synthesized attribute values of which are supplied by lexical analyzer
- Start symbol does not have any inherited attribute

This is a grammar which uses only synthesized attributes. Start symbol has no parents, hence no inherited attributes.

Annotated Parse tree for 3 * 4 + 5 n (Bottom up parsing)



- Values of lexval is supplied by lexical analyzer
- Each node (nonT) has attribute val which is computed in bottom up order
- At the node with a child level *, after computing T.val =3 and f.val=5 at its first and third children, we apply the rule that says T.val is the product of these two values or 15.

Dependency graph is also shown in Blue and red lines.

Inherited Attributes

- An inherited attribute is one whose value is defined in terms of attributes at the parent and/or siblings

- Inherited attributes help to find the context (type, scope etc.) of a token e.g., the type of a token or scope when the same variable name is used multiple times in a program in different functions

- Let take an example.

Example: Inherited Attributes

Production ID	Production Rules	Semantic Rules associated with each production
p1	D ▭ T L	L.inh = T.type
p2	T ▭ float	T.type = float
p3	T ▭ int	T.type = integer
p4	L ▭ L_1 , id	L_1.inh = L.inh addtype (id.entry, L.inh)
p5	L ▭ id	addtype (id.entry,L.inh)

1. The above SDD takes a simple declaration D consisting of a basic type T followed by a list of L of identifiers. T can be *int* or *float*
2. Nonterminal D represents a declaration (p1) consists of a type T followed by a list of L identifiers.
3. T has one attributes, T.type, which is the type in the declaration D.
4. Nonterminal L has also one inherited attribute (inh ▭ inherited). The purpose of L.inh is to pass the declared type down the list of identifiers, so that it can be added to the appropriate symbol table entries.

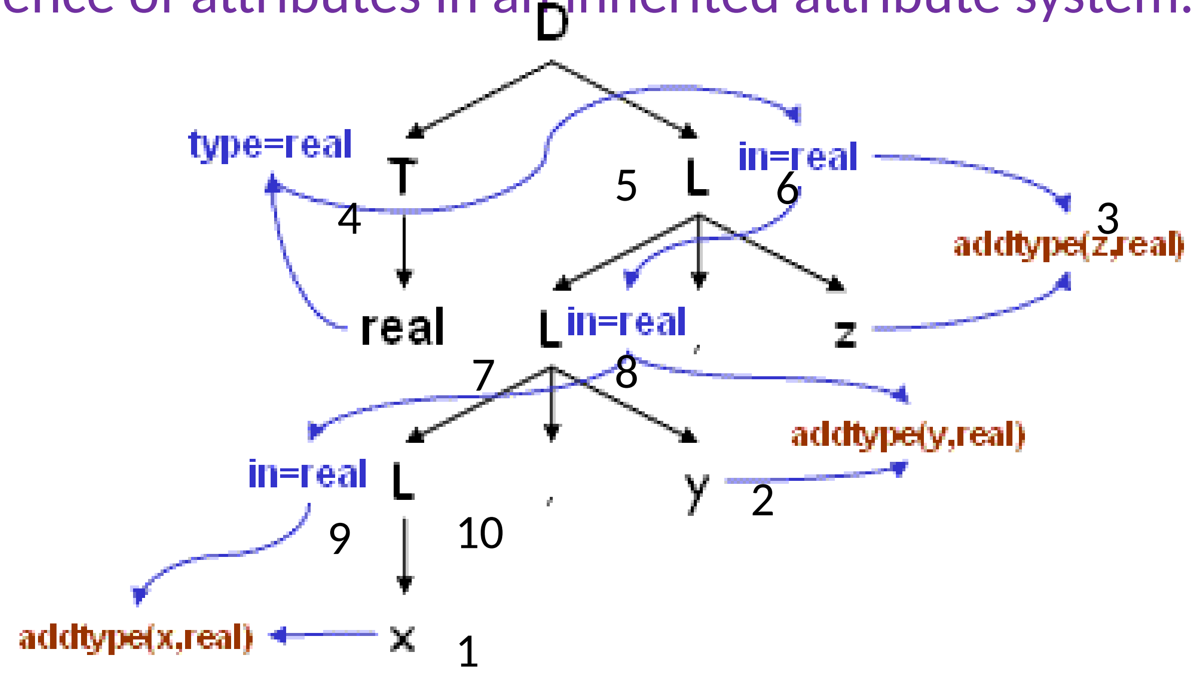
Example: Inherited Attributes

Production ID	Production Rules	Semantic Rules associated with each production
p1	D ▭ T L	L.inh = T.type
p2	T ▭ float	T.type = float
p3	T ▭ int	T.type = integer
p4	L ▭ L_1 , id	L_1.inh = L.inh addtype (id.entry, L.inh)
p5	L ▭ id	addtype (id.entry,L.inh)

5. Each of p2 and p3 evaluate the synthesized attribute T.type (int or float). This type is passed to the attribute L.inh semantic rule with p1.
6. p4 passes L.inh down the parse tree i.e L1.inh is computed at a parse tree node by copying the value of L.inh from the parent of that node; the parent corresponds to the head of the production.
7. P4 and p5 have a rule in which func. addType is called with two arguments
   - Id.entry ▭ a lexical value that points to a symbol table objects
   - L.inh, the type being assigned to every identifier on the list

Example: Parse tree for real x, y, z

Dependence of attributes in an inherited attribute system.



- 1 to 10 denotes nodes of the dependency graph
- Nodes 1,2,3 represents attributes entry with ids (x,y,z)
- 6,8,10 are the dummy attributes that represents the application of the function addType to a type and one of these entry values.
- Node 4 represents the attribute T.type where evaluation begins. This type is then passed to node 5, 7and 9 representing L.inh associated with each of the occurrences' of the non terminal L

Dependency Graph

- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c

- The dependencies among the nodes can be depicted by a directed graph called dependency graph

Dependency Graph : Directed graph indicating interdependencies among the synthesized and inherited attributes of various nodes in a parse tree.

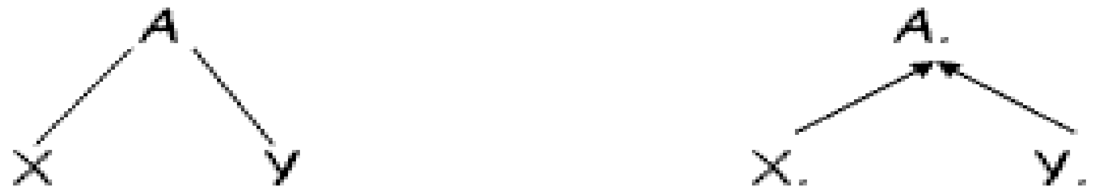
Algorithm to construct dependency graph

```
for each node **n** in the parse tree do
    for each attribute **a** of the grammar symbol 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 ($c_1$ , $c_2$ , ..., $c_k$ ) do
    { associated with production at n }
    for i = 1 to k do
    construct an edge from $c_i$ to b
```

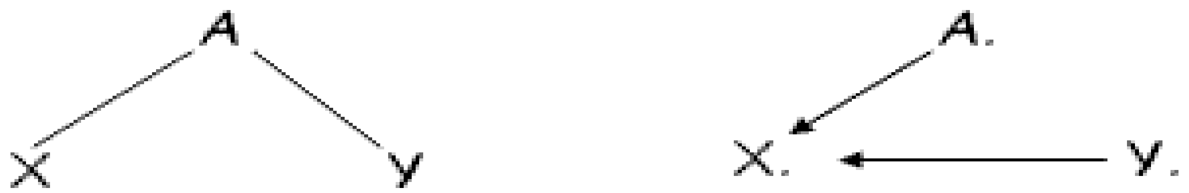
An algorithm to construct the dependency graph. After making one node for every attribute of all the nodes of the parse tree, make one edge from each of the other attributes on which it depends.

Example

- Suppose A.a = f(X.x , Y.y) is a semantic rule for A → X Y



- If production A → X Y has the semantic rule X.x = g(A.a, Y.y)



The semantic rule A.a = f(X.x , Y.y) for the production A -> XY defines the synthesized attribute a of A to be dependent on the attribute x of X and the attribute y of Y . Thus the dependency graph will contain an edge from X.x to A.a and Y.y to A.a accounting for the two dependencies. Similarly for the semantic rule X.x = g(A.a , Y.y) for the same production there will be an edge from A.a to X.x and an edge from Y.y to X.x.

Example

Whenever following production is used in a parse tree

E → E 1 + E 2   E.val = E 1 .val + E 2 .val

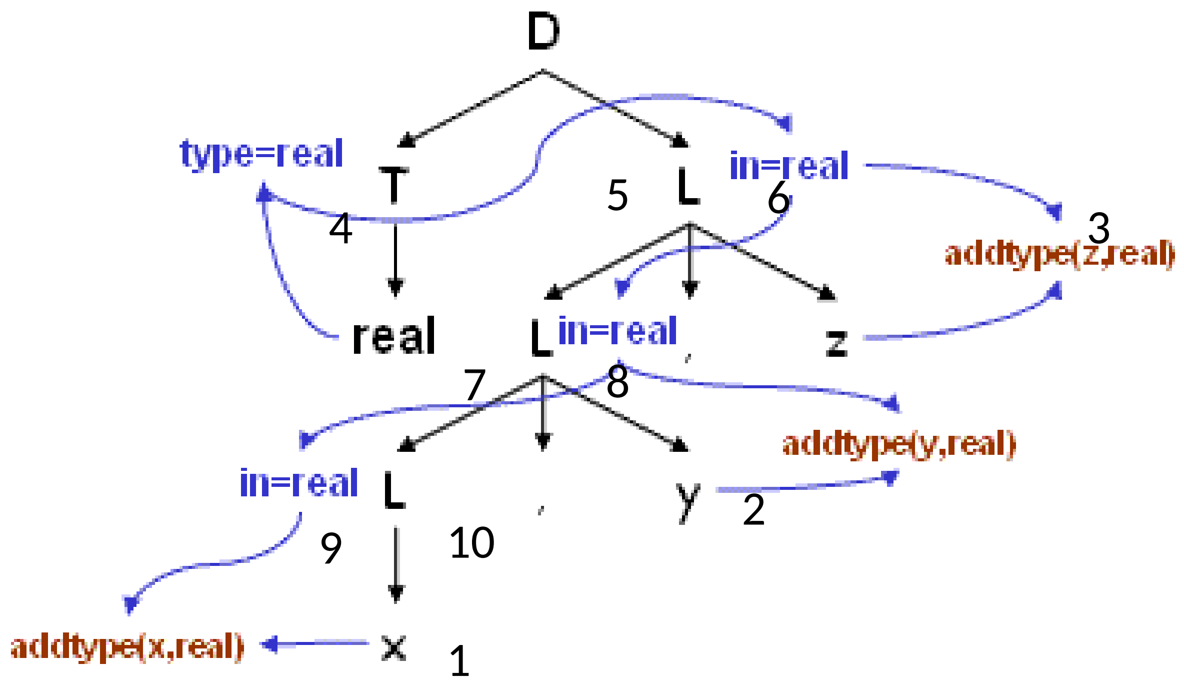
we create a dependency graph

$$\begin{tikzcd}
 & E.val & \\
E_1.val \arrow[ur] & & E_2.val \arrow[ul]
\end{tikzcd}$$

The synthesized attribute E.val depends on E1.val and E2.val hence the two edges one each from E 1 .val & E 2 .val

Example

- dependency graph for real id1, id2, id3
- put a dummy synthesized attribute b for a semantic rule that consists of a procedure call



The figure shows the dependency graph for the statement real id1, id2, id3 along with the parse tree. Procedure calls can be thought of as rules defining the values of dummy synthesized attributes of the nonterminal on the left side of the associated production. Blue arrows constitute the dependency graph and black lines, the parse tree. Each of the semantic rules addtype (id.entry, L.in) associated with the L productions leads to the creation of the dummy attribute.

Evaluation Order

Any topological sort of dependency graph gives a valid order in which semantic rules must be evaluated

a4 = real

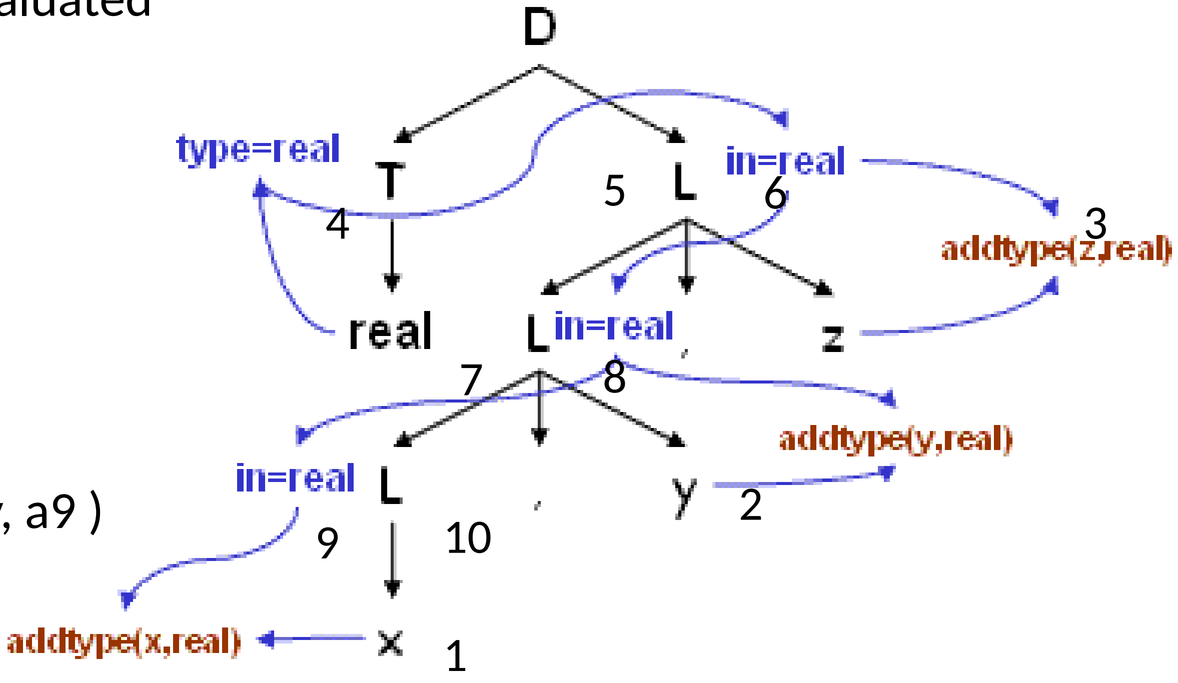
a5 = a4

addtype(id3.entry, a5)

a7 = a5

addtype(id2.entry, a7)

a9 := a7 addtype(id1.entry, a9)



A topological sort of a directed acyclic graph is any ordering m1, m2, m3 ..... mk of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes. Thus if mi -> mj is an edge from mi to mj then mi appears before mj in the ordering. The order of the statements shown in the slide is obtained from the topological sort of the dependency graph in the previous slide. 'an' stands for the attribute associated with the node numbered n in the dependency graph. The numbering is as shown in the previous slide.

Abstract Syntax Tree

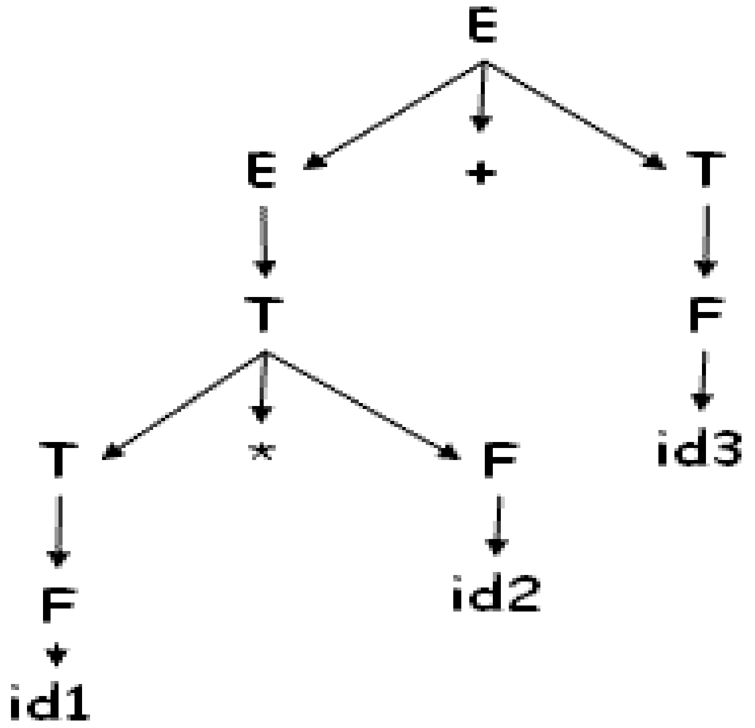
- Condensed form of parse tree
- Useful for representing language constructs.
- The production S *if B then s1 else s2* may appear as

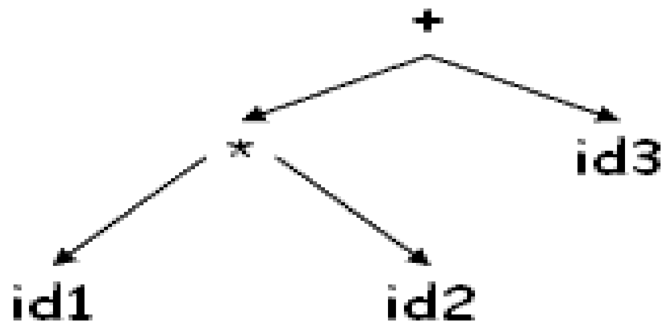
$$\begin{tikzcd}
 & \text{if-then-else} \arrow[dl] \arrow[d] \arrow[dr] & \\
B & s1 & s2
\end{tikzcd}$$

Abstract Syntax tree

- Operators and keywords do not appear as leaves, rather they are associated with the interior node that would be parent of those leaves in the parse tree.

- Chain of single productions may be collapsed





Constructing Abstract Syntax tree for expression

- Each node can be represented as a record
  - *operators* : one field for operator, remaining fields ptrs to operands mknode(op,left,right)
  - *identifier* : one field with label id and another ptr to symbol table mkleaf(id,entry)
  - *number* : one field with label num and another to keep the value of the number

    mkleaf(num,val)

- Each node in an abstract syntax tree can be implemented as a record with several fields.

- In the node for an operator one field identifies the operator (called the label of the node) and the remaining contain pointers to the nodes for operands.

- Nodes of an abstract syntax tree may have additional fields to hold values (or pointers to values) of attributes attached to the node.

- The functions given in the slide are used to create the nodes of abstract syntax trees for expressions. Each function returns a pointer to a newly created note.

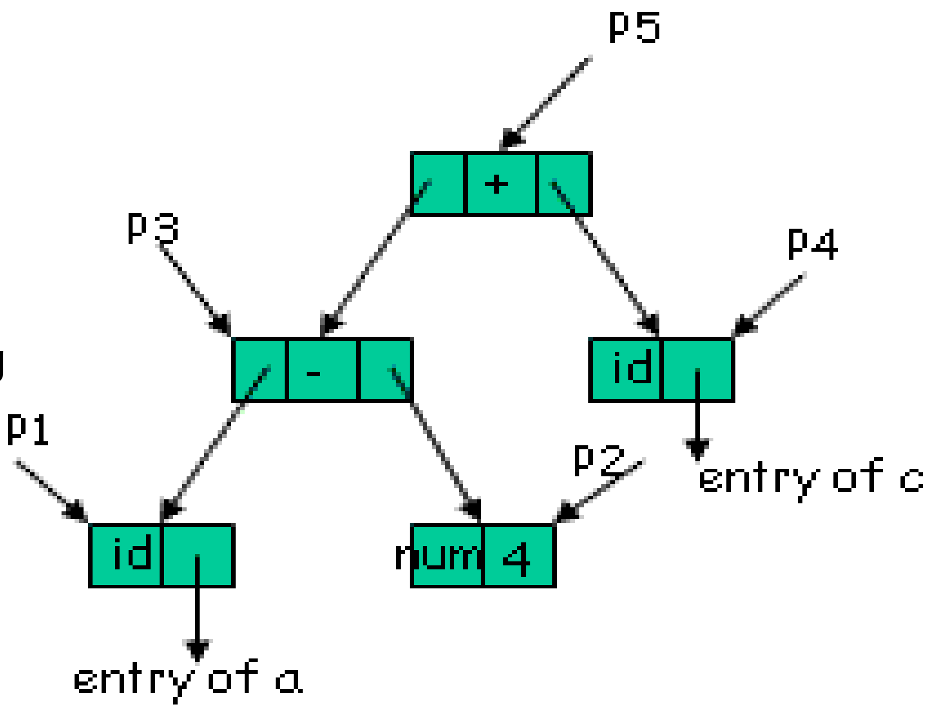
Example

the following

sequence of function

calls creates a parse

tree for a- 4 + c using bottom-up parsing



P 1 = mkleaf(id, entry.a)

P 2 = mkleaf(num, 4)

P 3 = mknode(-, P 1 , P 2)

P 4 = mkleaf(id, entry.c)

P 5 = mknode(+, P 3 , P 4)

Example

An example showing the formation of an abstract syntax tree by the given function calls for the expression a-4+c.The call sequence can be explained as:

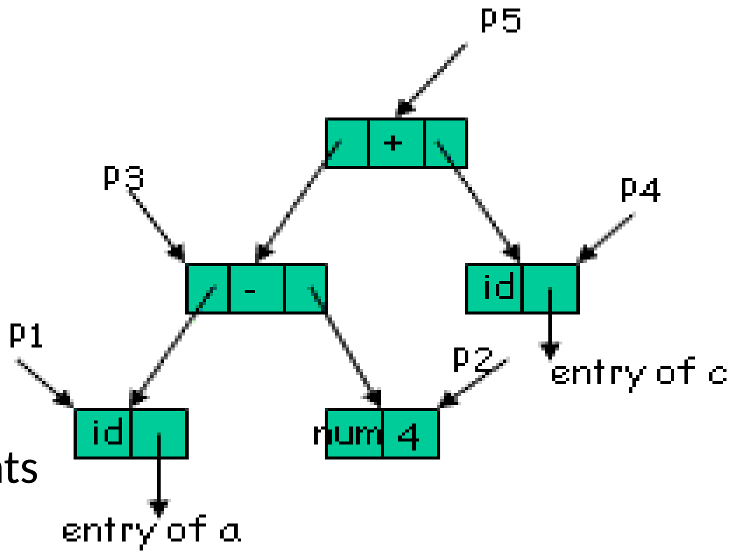
1. P1 = mkleaf(id,entry.a) : A leaf node made for the identifier “a” and an entry for “a” is made in the symbol table.

2. P2 = mkleaf(num,4) : A leaf node made for the number “4”.

3. P3 = mknode(-,P1,P2) : An internal node for the “-”. It takes the previously made nodes as arguments and represents the expression “a-4”.

4. P4 = mkleaf(id,entry.c) : A leaf node made for the identifier “c” and an entry for “c” is made in the symbol table.

5. P5 = mknode(+,P3,P4) : An internal node for the “+”. It takes the previously made nodes as arguments and represents the expression “a- 4+c”.

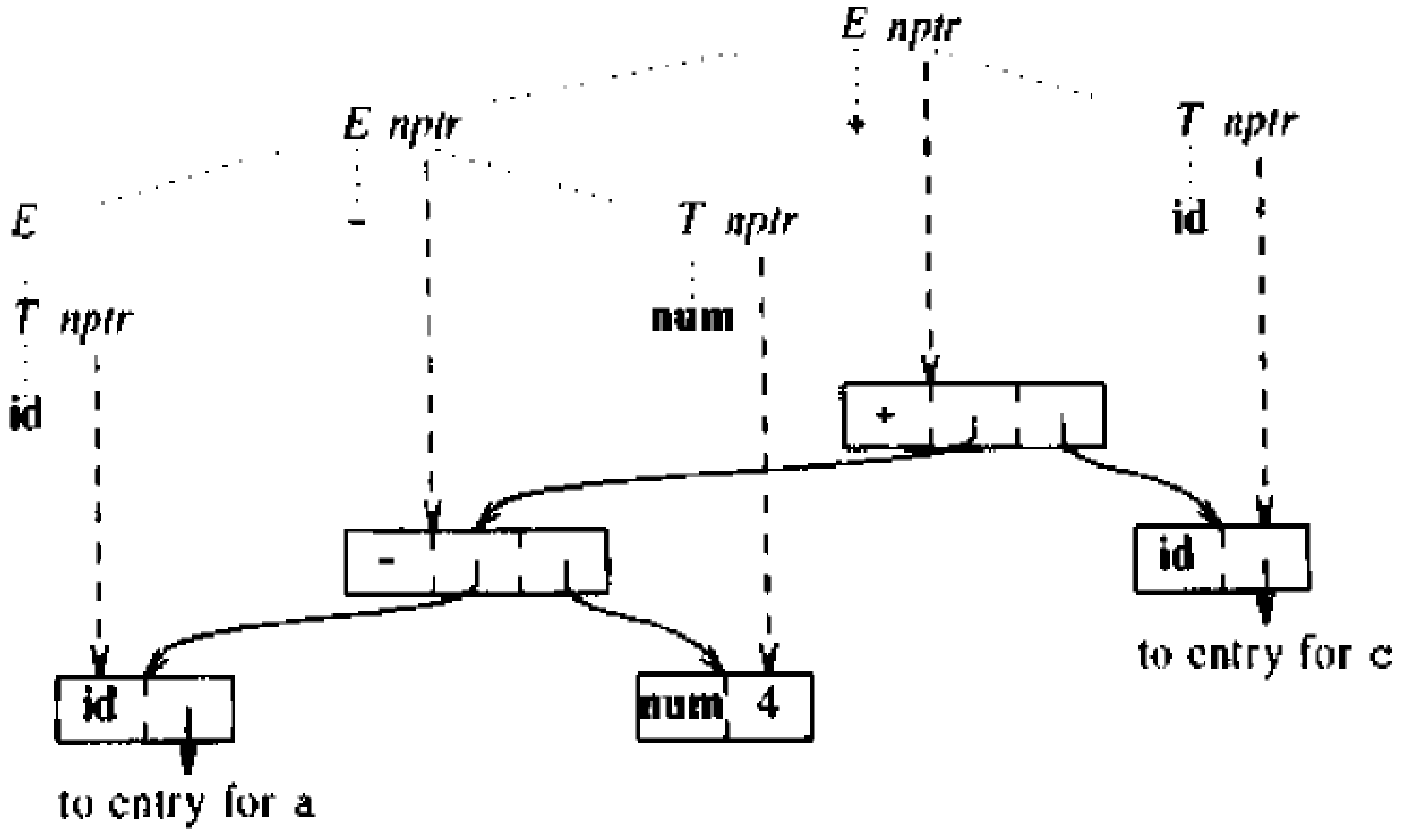


A syntax directed definition for constructing syntax tree for an expression

E ▭ E_1 + T	E.ptr = mknode(+, E_1. *ptr*, T.*ptr*)
E ▭ T	E.ptr = T. *ptr*
T ▭ T_1 * F	T.ptr := mknode(*, T_1. *ptr*, F.*ptr*)
T ▭ F	T.ptr := F. *ptr*
F ▭ (E)	F.ptr := E. *ptr*
F ▭ id	F.ptr := mkleaf(id, id.*entry*)
F ▭ num	F.ptr := mkleaf(num, num.*val*)

Now we have the syntax directed definitions to construct the parse tree for a given grammar.

Construction of a syntax tree for a-4+c



Directed Acyclic Graph (DAG) for Expression

- A DAG has leaves corresponding to atomic operands and interior codes corresponding to operators.

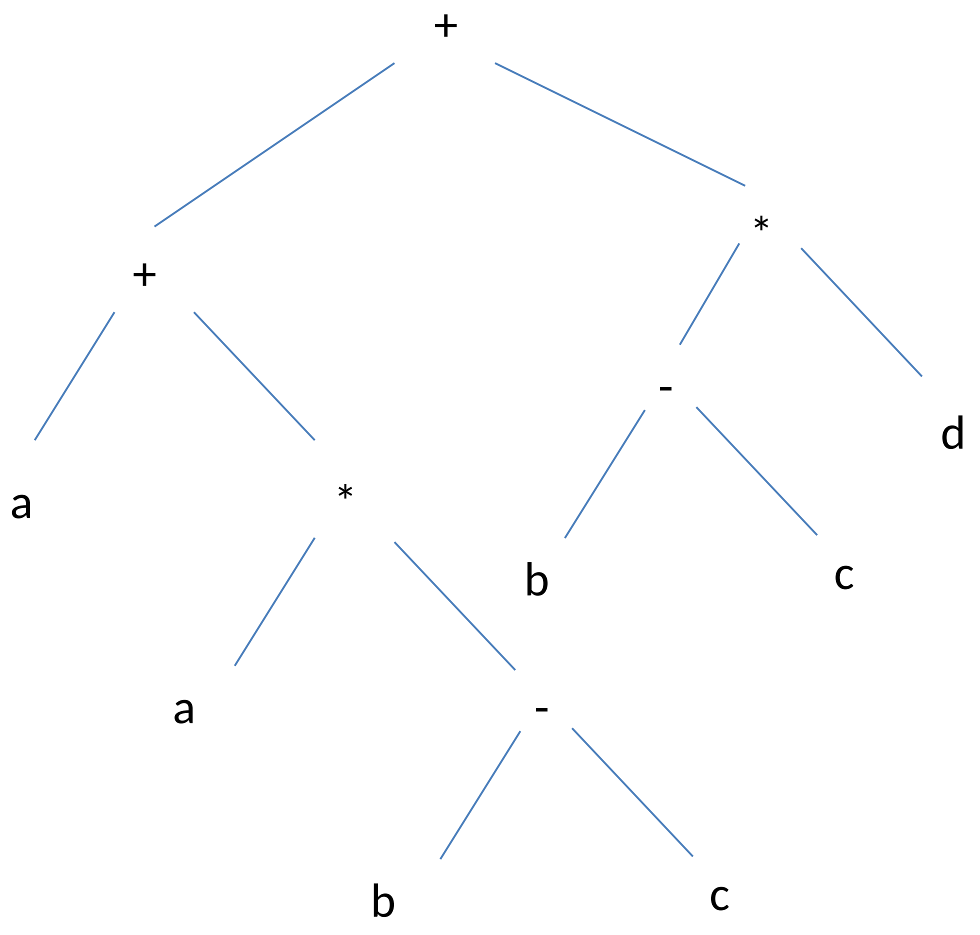
- The difference is that a node N in a DAG has more than one parent if N represents a common sub-expression.

- In a syntax tree, the tree for the common sub expression would be replicated as many times as the sub-expression appears in the original expression.

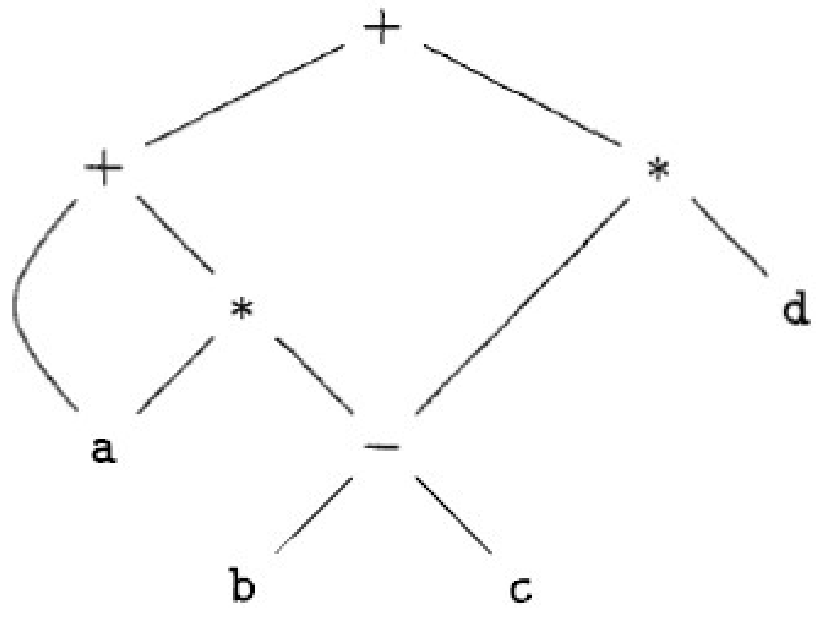
- Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

DAG for the expression

a + a * (b - c) + (b - c) * d



Abstract Syntax Tree



DAG

DAG for the expression

E ▭ E_1 + T	E.ptr = mknode(+, E_1. *ptr*, T.*ptr*)
E ▭ T	E.ptr = T. *ptr*
T ▭ T_1 * F	T.ptr := mknode(*, T_1. *ptr*, F.*ptr*)
T ▭ F	T.ptr := F. *ptr*
F ▭ (E)	F.ptr := E. *ptr*
F ▭ id	F.ptr := mkleaf(id, id.*entry*)
F ▭ num	F.ptr := mkleaf(num, num.*val*)

- The above SDD can construct either syntax trees or DAG 's.

- It was used to construct syntax trees in previously where functions Leaf and Node created a fresh node each time they were called.

- It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned.

- For instance, before constructing a new node, Node(op, left, right) , we check whether there is already a node with label op, and children left and right, in that order. If so, Node returns the existing node; otherwise, it creates a new node .

Dag for Expression

P 1 = makeleaf(id,a)

P 2 = makeleaf(id,a)

P 3 = makeleaf(id,b)

P 4 = makeleaf(id,c)

P 5 = makenode(-,P 3 ,P 4)

P 6 = makenode(*,P 2 ,P 5)

P 7 = makenode(+,P 1 ,P 6)

P 8 = makeleaf(id,b)

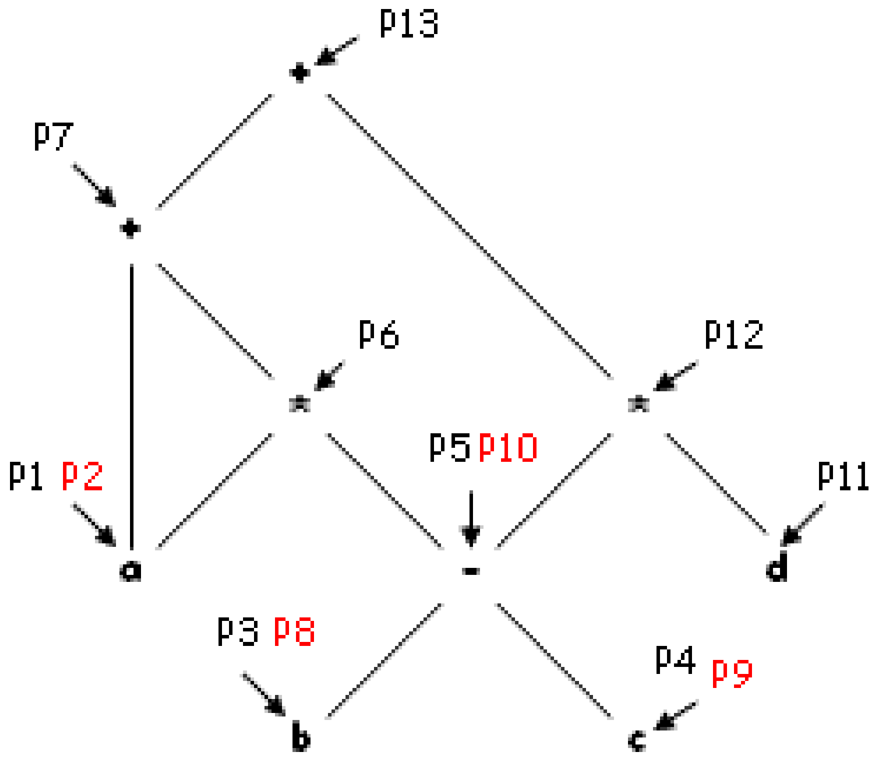
P 9 = makeleaf(id,c)

P 10 = makenode(-,P 8 ,P 9)

P 11 = makeleaf(id,d)

P 12 = makenode(*,P 10 ,P 11)

P 13 = makenode(+,P 7 ,P 12)



Dag for Expression

A directed acyclic graph (DAG) for the expression : a + a * (b V c) + (b V c) * d.
All the function calls are made as in the order shown. Whenever the required node is already present, a pointer to it is returned so that a pointer to the old node itself is obtained. A new node is made if it did not exist before. The function calls can be explained as:

- P1 = makeleaf(id,a)
  A new node for identifier Qa R made and pointer P1 pointing to it is returned.
- P2 = makeleaf(id,a)
  Node for Qa R already exists so a pointer to that node i.e. P1 returned.
- P3 = makeleaf(id,b)
  A new node for identifier Qb R made and pointer P3 pointing to it is returned.
- P4 = makeleaf(id,c)
  A new node for identifier Qc R made and pointer P4 pointing to it is returned.
- P5 = makenode(-,P3,P4)
  A new node for operator Q- R made and pointer P5 pointing to it is returned. This node becomes the parent of P3,P4.
- P6 = makenode(*,P2,P5)
  A new node for operator Q- R made and pointer P6 pointing to it is returned. This node becomes the parent of P2,P5.

Dag for Expression

- P7 = makenode(+,P1,P6)

  A new node for operator Q+ R made and pointer P7 pointing to it is returned. This node becomes the parent of P1,P6.

- P8 = makeleaf(id,b)

  Node for Qb R already exists so a pointer to that node i.e. P3 returned.

- P9 = makeleaf(id,c)

  Node for Qc R already exists so a pointer to that node i.e. P4 returned.

- P10 = makenode(-,P8,P9)

  A new node for operator Q- R made and pointer P10 pointing to it is returned. This node becomes the parent of P8,P9.

- P11 = makeleaf(id,d)

  A new node for identifier Qd R made and pointer P11 pointing to it is returned.

- P12 = makenode(*,P10,P11)

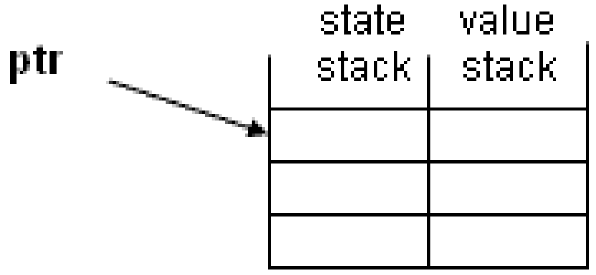
  A new node for operator Q* R made and pointer P12 pointing to it is returned. This node becomes the parent of P10,P11.

- P13 = makenode(+,P7,P12)

  A new node for operator Q+ R made and pointer P13 pointing to it is returned. This node becomes the parent of P7, P12.

Bottom-up evaluation of S-attributed definitions

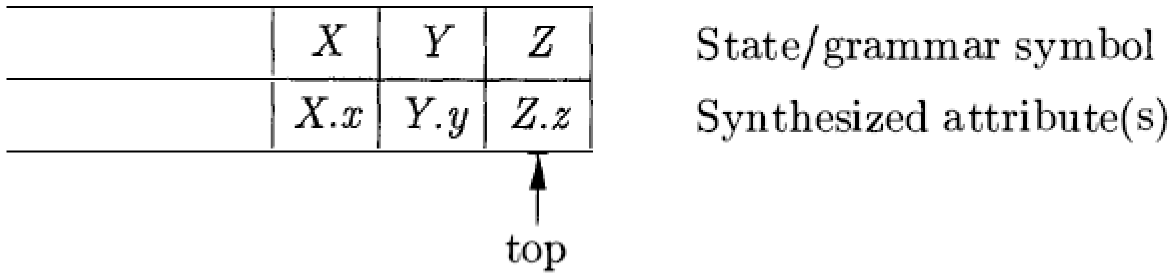
- Can be evaluated while parsing.
- Whenever reduction is made, value of new synthesized attribute is computed from the attributes on the stack
- Extend stack to hold the values also
- The current top of stack is indicated by ptr top



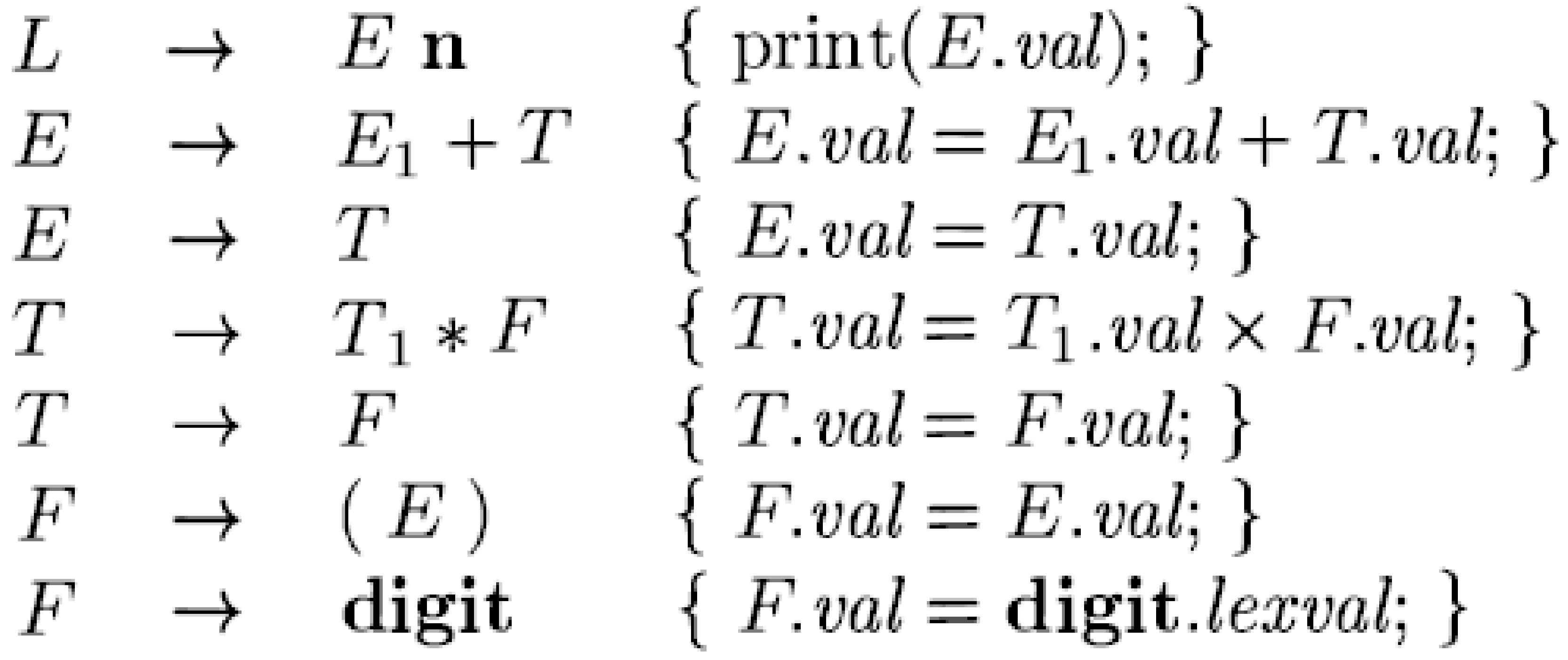
1. Synthesized attributes are evaluated using the attributes of the children nodes only.
2. So in a bottom up evaluation, if the attributes are maintained on a stack then the attributes of nodes higher up in the parse tree can be evaluated.
3. The stack is extended to hold the state as well as the value.
4. Top of the stack is maintained in a location pointed by the pointer top.

Bottom-up evaluation of S-attributed definitions

- 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)
- An example of a parser stack while parsing of A.a := f(X.x,Y.y,Z.z).
- Top is decremented by two because three values viz. X,Y and Z are popped and value of A has to be pushed in.



Example: desk calculator



L	$\rightarrow$	$E\ \mathbf{n}$	$\{\ \text{print}(E.val);\ \}$
E	$\rightarrow$	$E_1 + T$	$\{\ E.val = E_1.val + T.val;\ \}$
E	$\rightarrow$	T	$\{\ E.val = T.val;\ \}$
T	$\rightarrow$	$T_1 * F$	$\{\ T.val = T_1.val \times F.val;\ \}$
T	$\rightarrow$	F	$\{\ T.val = F.val;\ \}$
F	$\rightarrow$	$(\ E\ )$	$\{\ F.val = E.val;\ \}$
F	$\rightarrow$	$\mathbf{digit}$	$\{\ F.val = \mathbf{digit}.lexval;\ \}$

Example: desk calculator

- Suppose that the stack is kept in an array of records called stack, with top a cursor to the top of the stack. Thus, stack[top] refers to the top record on the stack, stack[top - 1] to the record below that, and so on.

- Also, we assume that each record has a field called val, which holds the attribute of whatever grammar symbol is represented in that record. Thus, we may refer to the attribute E.val that appears at the third position on the stack as stack[top - 2] . val.

- For instance, in the second production, E ▭ E_1 + T, we go two positions below the top to get the value of E_1 , and we find the value of T at the top. The resulting sum is placed where the head E will appear after the reduction, that is, two positions below the current top. The reason is that after the reduction, the three topmost stack symbols are replaced by one.

- After computing E. val, we pop two symbols off the top of the stack, so the record where we placed E. val will now be at the top of the stack.

Example: desk calculator

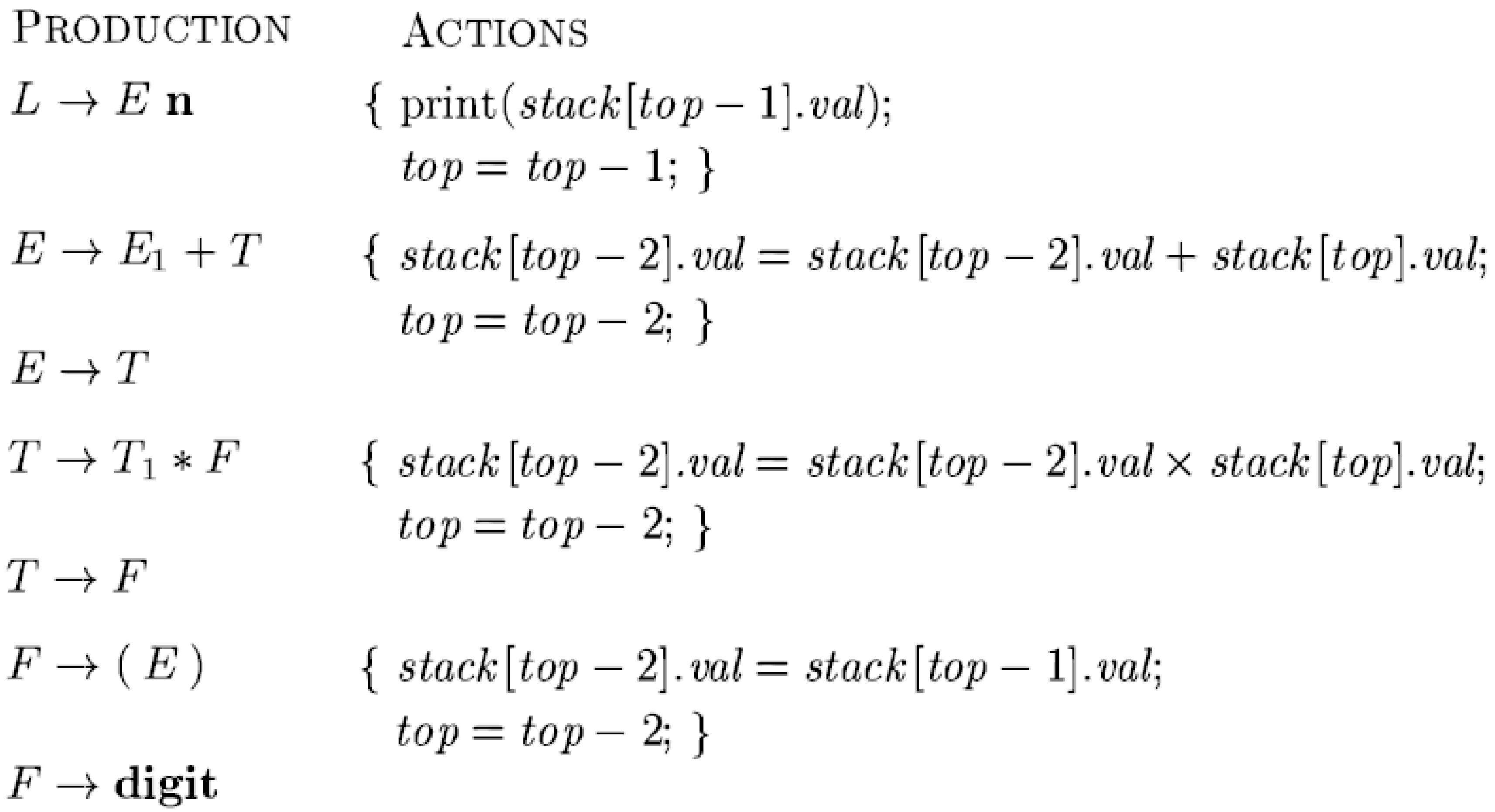
- Before reduction *ntop = top - r +1* where *ntop* is a temporary variable

- After code reduction *top = ntop*

- The code fragments (like *val(ntop) = val(top-1)*) in the implementation of the desk calculator

- When a production with r symbols on then right side is reduced, the value of *ntop* is set to *top - r + 1* . After each code fragment is executed, top is set to *ntop* .

Example: desk calculator



Production	Actions
$L \to E\ \mathbf{n}$	$\{\ \mathrm{print}(stack[top-1].val);$ $top = top - 1;\ \}$
$E \to E_1 + T$	$\{\ stack[top-2].val = stack[top-2].val + stack[top].val;$ $top = top - 2;\ \}$
$E \to T$	
$T \to T_1 * F$	$\{\ stack[top-2].val = stack[top-2].val \times stack[top].val;$ $top = top - 2;\ \}$
$T \to F$	
$F \to (\ E\ )$	$\{\ stack[top-2].val = stack[top-1].val;$ $top = top - 2;\ \}$
$F \to \mathbf{digit}$	

Example: desk calculator

- Evaluation of the synthesized attributes of 3*5+4n by an LR parser during a bottom up pass.
- Initial sequence of events on seeing the input : The parser shifts the state corresponding to token digit (taken as digit here) onto the stack.
- In the second move, the parser reduces by the production F -> digit. It implements the semantic rule F.val = digit.lexval.
- In the third move, the parser reduces by T -> F. No code fragment is associated so the val array is unchanged.
- Rest of the implementation goes on similar lines. After each reduction, the top of the val stack contains the attribute value associated with the left side of the reducing production.

Example: desk calculator

INPUT	STATE	Val	PRODUCTION
3*5+4n			
*5+4n	digit	3	
*5+4n	F	3	F → digit
*5+4n	T	3	T → F
5+4n	T*	3 -	
+4n	T*digit	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+digit	15 - 4	
n	E+F	15 - 4	F → digit
n	E+T	15 - 4	T → F
n	E	19	E → E +T

Semantic Analysis
End of part I

Example

Extend the scheme which has a rule number → sign list .

List replacing number → sign list

number → sign list	list.position := 0
	if sign.negative then
	number.value := - list.value
	else
	number.value := list.value
sign → +	sign.negative := false
sign → -	sign.negative := true
list → bit	bit.position := list.position
	list.value := bit.value
$list_0$ → $list_1$ bit	$list_1$.position := $list_0$.position + 1
	bit.position := $list_0$.position
	$list_0$.value := $list_1$.value + bit.value
bit → 0	bit.value := 0
bit → 1	bit.value := $2^{bit.position}$