CS 335: Semantic Analysis

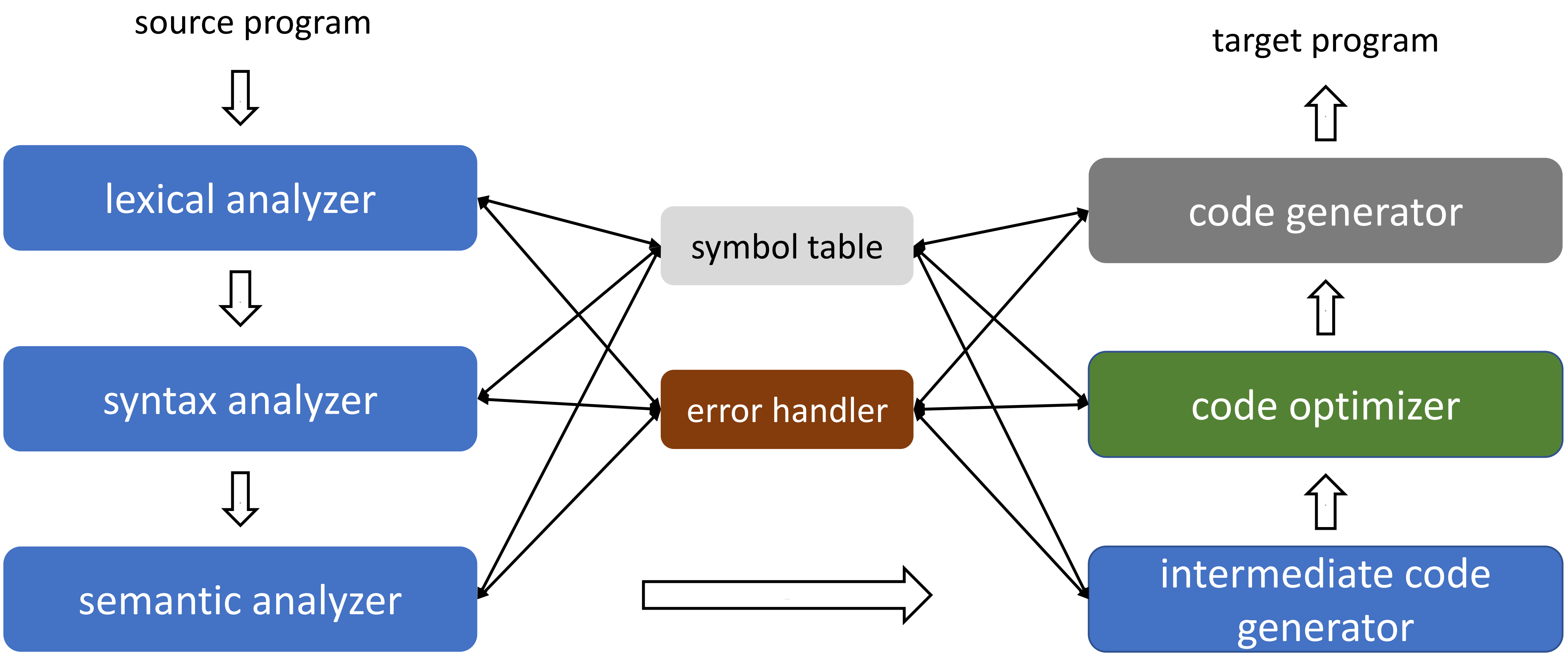
Swarnendu Biswas

Semester 2019-2020-II

CSE, IIT Kanpur

Content influenced by many excellent references, see References slide for acknowledgements.

An Overview of Compilation



Beyond Scanning and Parsing

- Example 1

```
std::string x;
int y;
y = x + 3
```

- Example 2

```
int a, b;
a = b + c
```

```
int dot_prod(int x[], int y[]) {
  int d, i;
  d = 0;
  for (i=0; i<10; i++)
    d += x[i]*y[i];
  return d;
}

main() {
  int p, a[10], b[10];
  p = dot_prod(a,b);
}
```

Beyond Scanning and Parsing

- A compiler must do more than just recognize whether a sentence belongs to a programming language grammar
  - An input program can be grammatically correct but may contain other errors that prevent compilation
  - Lexer and parser cannot catch all program errors
- Some language features cannot be modeled using context-free grammar (CFG)
  - Whether a variable has been declared before use?
  - Parameter types and numbers match in the declaration and use of a function
  - Types match on both sides of an assignment

Limitations with CFGs

$$ProcedureBody \rightarrow Declarations\ Executables$$

- CFGs deal with syntactic categories rather than specific words
  - Grammar can specify the positions in an expression where a variable name may occur
- Enforcing the “declare before use” rule requires knowledge that cannot be encoded in a CFG
  - CFG cannot match one instance of a variable name with another

Questions That Compiler Needs to Answer

Questions

- Has a variable been declared?
- What is the type and size of a variable?
- Is the variable scalar or an array?
- Is an array access A[i][j][k] consistent with the declaration?
- Does the name “x” correspond to a variable or a function?
- If x is a function, how many arguments does it take?
- What kind of value, if any, does a function x return?
- Are all invocations of a function consistent with the declaration?
- Track inheritance relationship
- Ensure that classes and its methods are not multiply defined

Questions That Compiler Needs to Answer

$x \leftarrow y + z$

$x \leftarrow a + b$

Compilers need to understand the structure of the computation to **translate** the input program

Semantic Analysis

- Finding answers to these questions is part of the semantic analysis phase
  - For example, ensure variable are declared before their uses and check that each expression has a correct type
  - These are static semantics of languages

Checking Dynamic Semantics

- Dynamic semantics of languages need to be checked at run time
  - Whether an overflow will occur during an arithmetic operation?
  - Whether array bounds will be exceeded during execution?
  - Whether recursion will exceed stack limits?

- Compilers can generate code to check dynamic semantics

```
int dot_prod(int x[], int y[]) {
  int d, i;
  d = 0;
  for (i=0; i<10; i++)
    d += x[i]*y[i];
  return d;
}

main() {
  int p; int a[10], b[10];
  p = dot_prod(a,b);
}
```

How does a compiler answer these questions?

- Compilers track additional information for semantic analysis
  - For example, types of variables, function parameters, and array dimensions
    - Type information is stored in the symbol table or the syntax tree
  - Used not only for semantic validation but also for subsequent phases of compilation
  - The information required may be non-local in some cases
- Semantic analysis can be performed during parsing or in another pass that traverses the IR produced by the parser

How does a compiler answer these questions?

- Use formal methods like context-sensitive grammars
- Use ad-hoc techniques using symbol table
- Static semantics of PL can be specified using attribute grammars
  - Attribute grammars are extensions of context-free grammars

Attribute Grammar Framework

Syntax-Directed Definition

- A syntax-directed definition (SDD) is a context-free grammar with rules and attributes
  - A SDD specifies the values of attributes by associating semantic rules with the grammar productions

Production	Semantic Rule
$E \rightarrow E_1 + T$	$E.code = E_1.code \| \| T.code \| \| " + "$

- Attribute grammars are SDDs with no side effects

Syntax-Directed Definition

- Generalization of CFG where each grammar symbol has an associated set of attributes
  - Let $G = (T, NT, S, P)$ be a CFG and let $V = T \cup NT$
  - Every symbol $X \in V$ is associated with a set of attributes (for e.g., denoted by $X.a$ and $X.b$)
  - Each attribute takes values from a specified domain (finite or infinite), which is its type
    - Typical domains of attributes are, integers, reals, characters, strings, booleans, and structures
  - New domains can be constructed from given domains by mathematical operations such as cross product and map
- Values of attributes are computed by semantic rules

Example

- Consider a grammar for signed binary numbers

$$number \rightarrow sign\ list$$
$$sign \rightarrow +|-$$
$$list \rightarrow list\ bit \mid bit$$
$$bit \rightarrow 0 \mid 1$$

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

Example

- Consider a grammar for signed binary numbers

$$
\begin{aligned}
&number \rightarrow sign\ list \\
&sign \rightarrow +\mid - \\
&list \rightarrow list\ bit \mid bit \\
&bit \rightarrow 0 \mid 1
\end{aligned}
$$

- Build attribute grammar that annotates $number$ with the value it represents

- Associate attributes with grammar symbols

Symbol	Attributes
$number$	val
$sign$	neg
$list$	pos, val
bit	pos, val

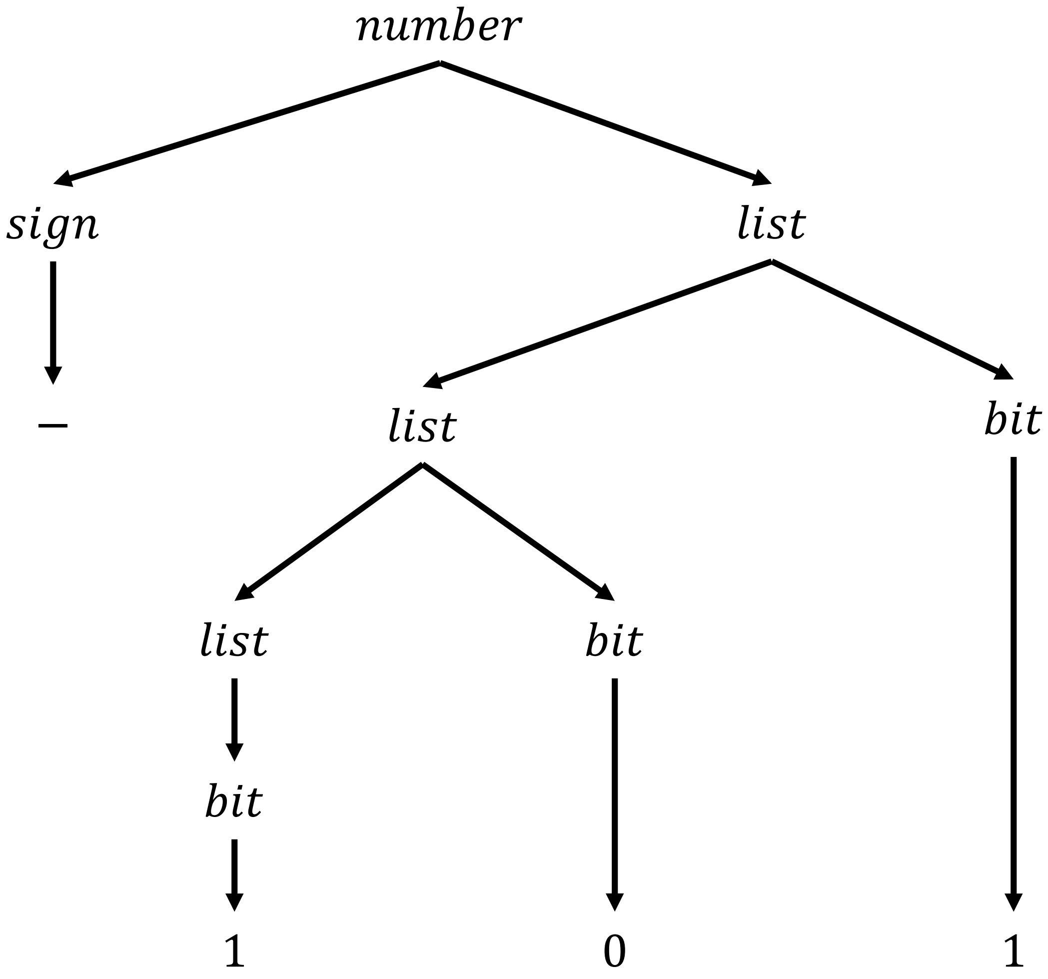
Example Attribute Grammar

Production	Attribute Rule

$number \rightarrow sign\ list$	$list.pos = 0$
	if $sign.neg$:
	$number.val = -list.val$
	else:
	$number.val = -list.val$

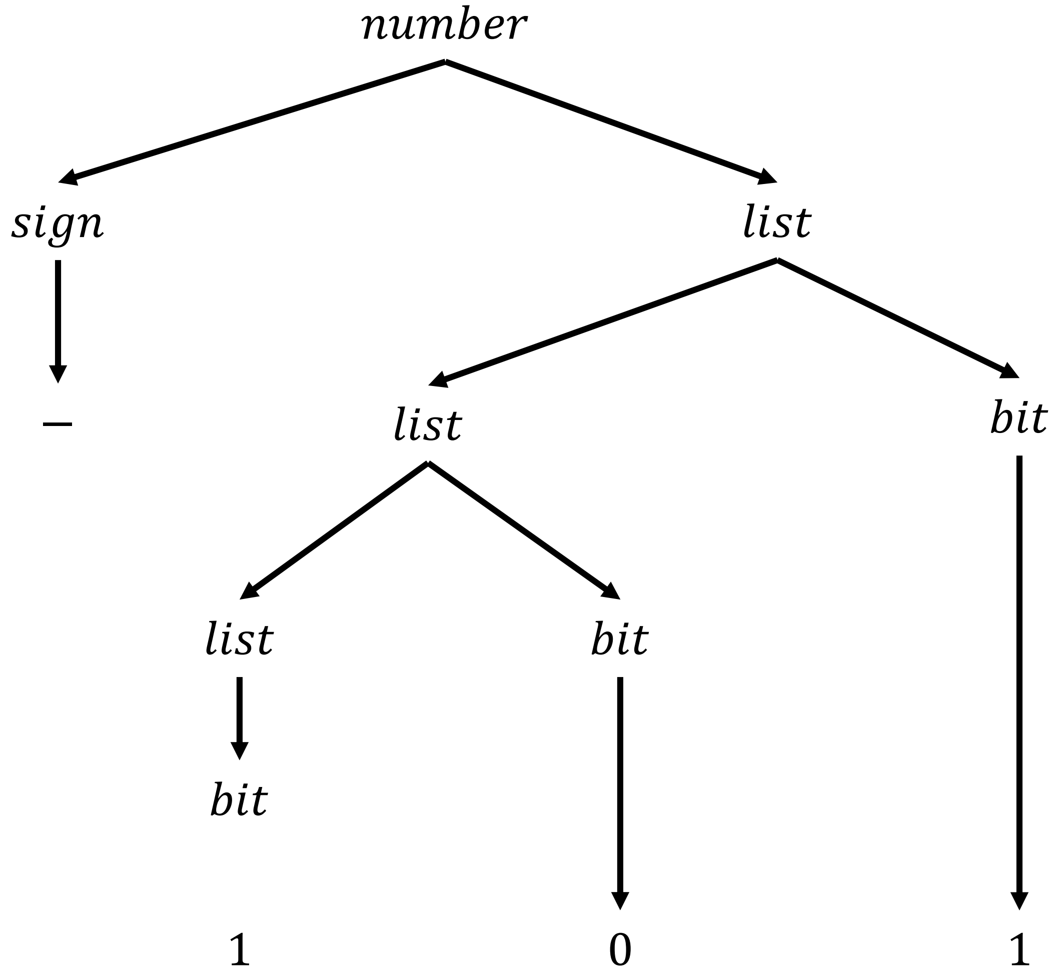
$sign \rightarrow +$	$sign.neg =$ false
$sign \rightarrow -$	$sign.neg =$ true
$list \rightarrow bit$	$bit.pos = list.pos$ $list.val = bit.val$
$list_0 \rightarrow list_1 bit$	$list_1.pos = list_0.pos + 1$ $bit.pos = list_0.pos$ $list_0.val = list_1.val + bit.val$
$bit \rightarrow 0$	$bit.val = 0$
$bit \rightarrow 1$	$bit.val = 2^{bit.pos}$

Parse Tree



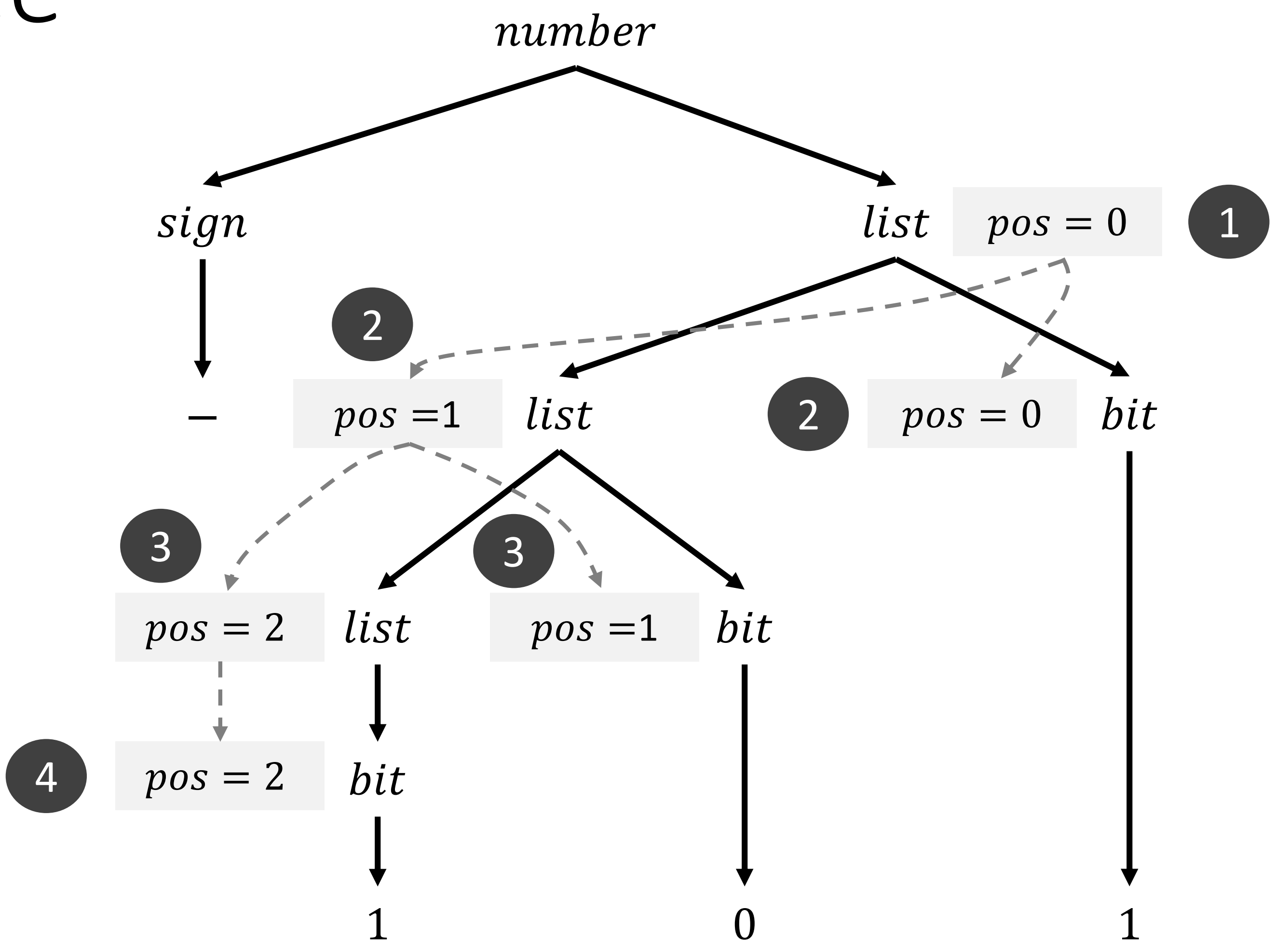
Annotated Parse Tree

- A parse tree showing the value(s) of its attribute(s) is called an annotated parse tree



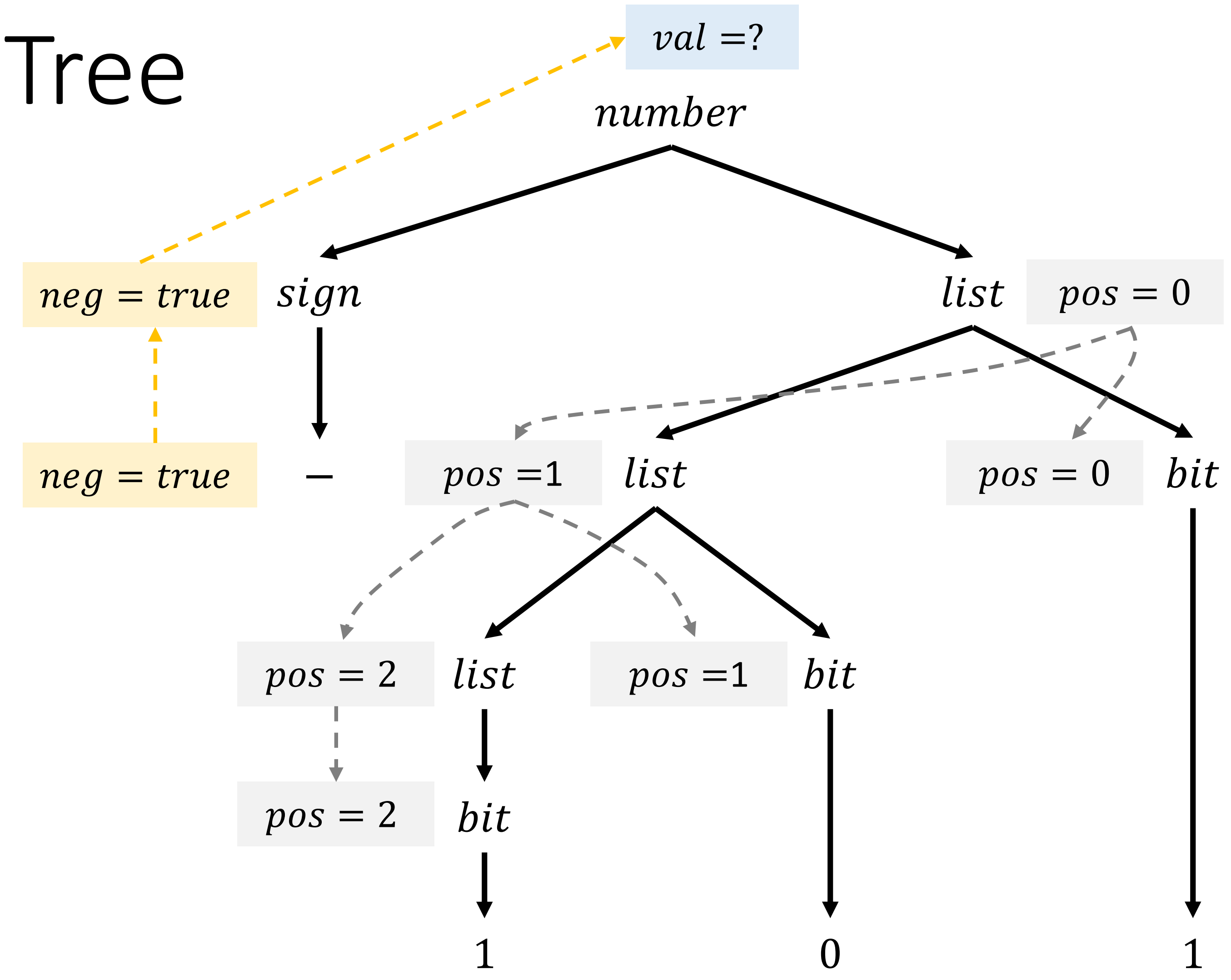
Annotated Parse Tree

- A parse tree showing the value(s) of its attribute(s) is called an annotated parse tree



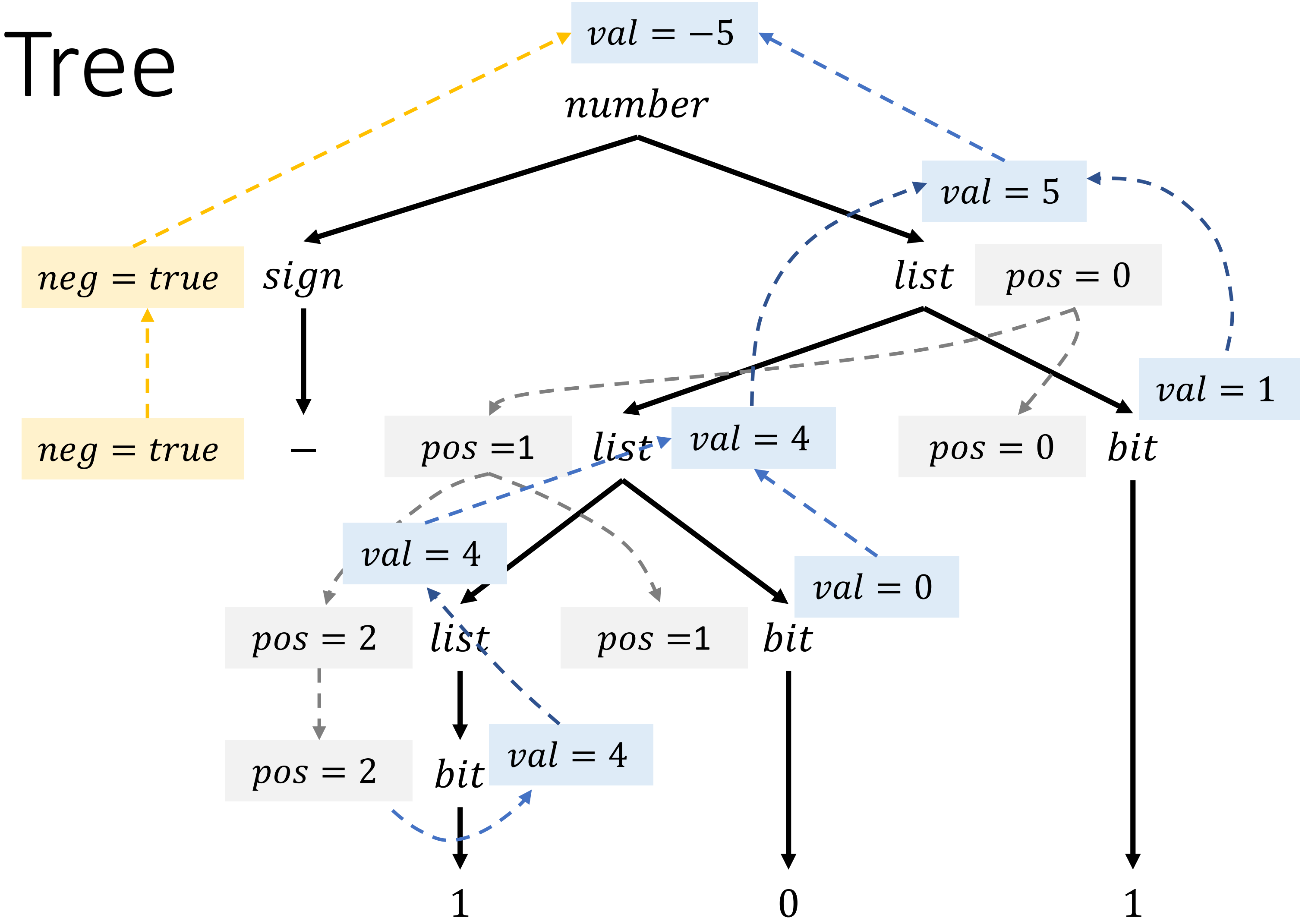
Annotated Parse Tree

- A parse tree showing the value(s) of its attribute(s) is called an annotated parse tree



Annotated Parse Tree

- A parse tree showing the value(s) of its attribute(s) is called an annotated parse tree

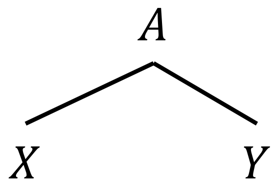


Dependency Graph

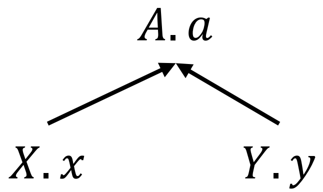
- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes are depicted by a directed graph called dependency graph

Dependency Graph

- Suppose $A.a = f(X.x, Y.y)$ is a semantic rule for $A \rightarrow XY$

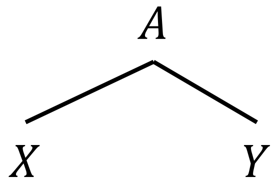


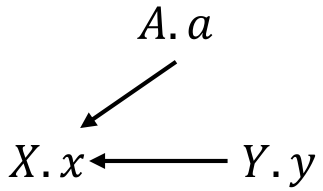
Parse tree



Dependency graph

- Suppose $X.x = f(A.a, Y.y)$ is a semantic rule for $A \rightarrow XY$





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 $b = f(c_1, c_2, \dots, c_k)$ do   // Associated with production at node n
        for $i = 1$ to k do
            construct an edge from c_i to b

Evaluating an SDD

- In what order do we evaluate attributes?
  - SDDs do not specify any order of evaluation
  - We must evaluate all the attributes upon which the attribute of a node depends
  - Any topological sort of dependency graph gives a valid order in which semantic rules must be evaluated

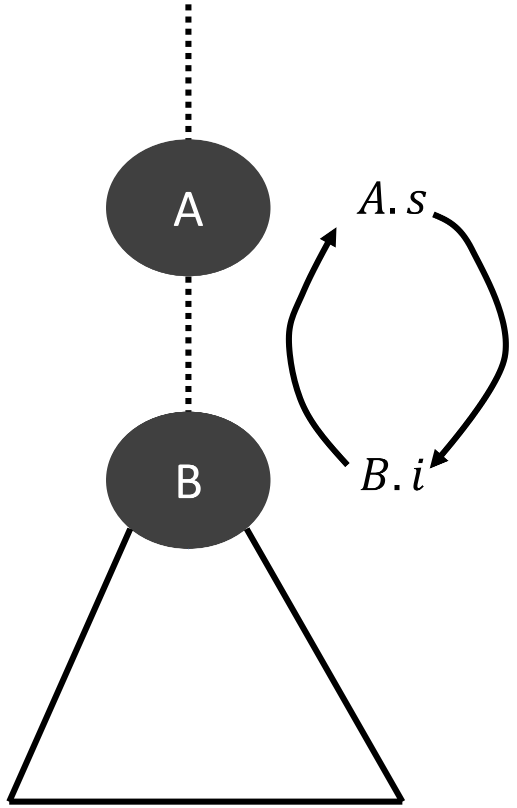
- For SDD's with both synthesized and inherited attributes, there is no guarantee of an order of evaluation existing

Evaluating an SDD

- Parse tree method
  - Use topological sort of the dependency graph to find the evaluation order
- Rule-based method
  - Semantic rules are analyzed and order of evaluation is predetermined
- Oblivious method
  - Evaluation order ignores the semantic rules

Circular Dependency of Attributes

Production	Semantic Rules
$A \rightarrow B$	$A.s = B.i$ $B.i = A.s + 1$



Types of Nonterminal Attributes

Synthesized

- Value of a synthesized attribute for a nonterminal A at a node N is computed from the values of children nodes and N itself
- Defined by a semantic rule associated with a production at N such that the production has A as its head

Inherited

- Value of an inherited attribute for a nonterminal B at a node N is computed from the values at N's parent, N itself, and N's siblings
- Defined by a semantic rule associated with the production at the parent of N such that the production has B in its body

Syntax-Directed Definition

- A grammar production $A \rightarrow \alpha$ has an associated semantic rule $b = f(c_1, c_2, \ldots, c_k)$
  - b is a synthesized attribute of A and c_1, c_2, ..., c_k are attributes of symbols in the production
  - b is an inherited attribute of a symbol in the body, and c_1, c_2, ..., c_k are attributes of symbols in the production

- Start symbol does not have inherited attributes

Terminal Attributes

- Terminals can have synthesized attributes, but not inherited attributes
- Attributes for terminals have lexical values that are supplied by the lexical analyzer

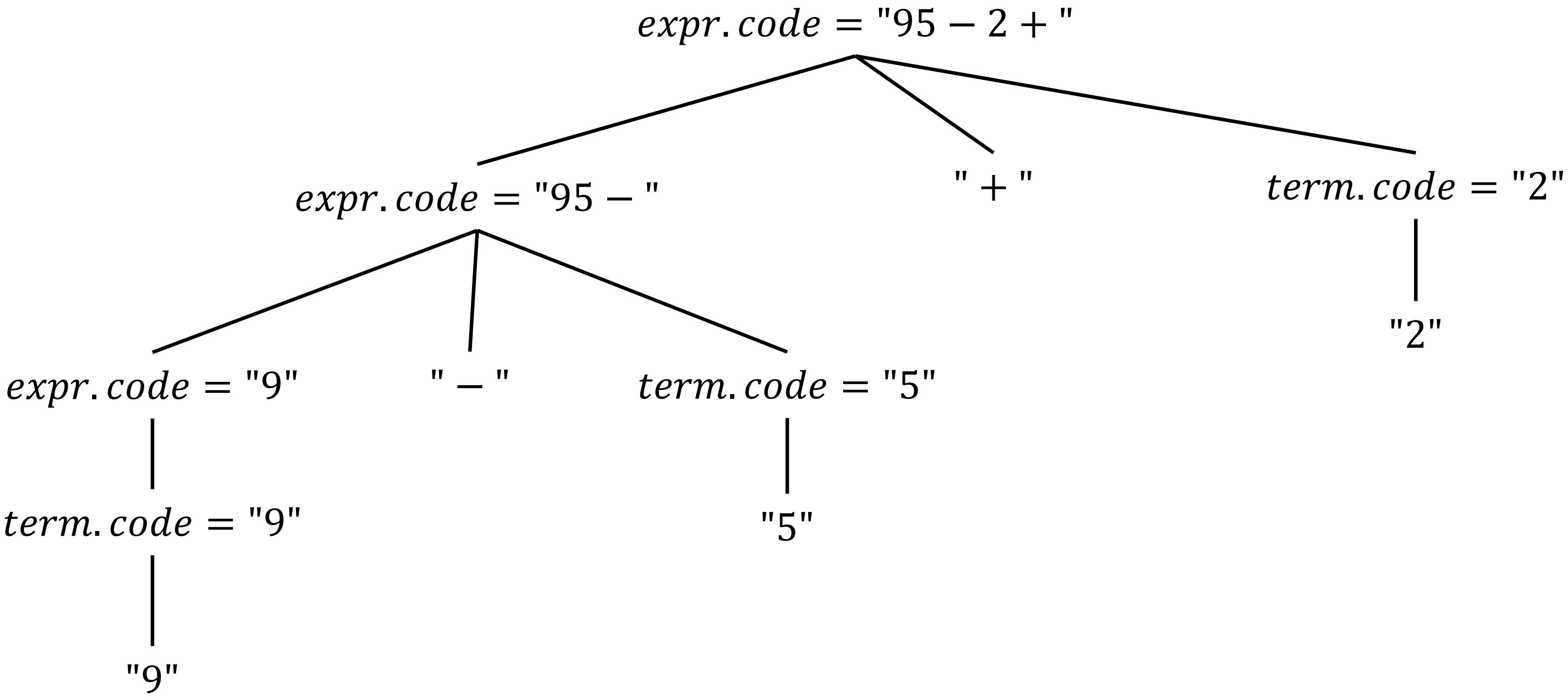
Postfix Notation

- Postfix notation for an expression E is defined inductively
  - If E is a variable or constant, then postfix notation is E
  - If $E = E_1 \mathrm{op} E_2$ where op is any binary operator, then the postfix notation is $E_1' E_2' \mathrm{op}$, where E_1' and E_2' are postfix notations for E_1 and E_2 respectively
  - If $E = (E_1)$, then postfix notation for E_1 is the notation for E

SDD for Infix to Postfix Translation

Production	Semantic Rules
$expr \rightarrow expr_1 + term$	$expr.code = expr_1.code \| \| term.code \| \| "+"$
$expr \rightarrow expr_1 - term$	$expr.code = expr_1.code \| \| term.code \| \| " - "$
$expr \rightarrow term$	$expr.code = term.code$
$term \rightarrow 0 \mid 1 \mid \ldots \mid 9$	$term.code = "0"$ $term.code = "1"$ … $term.code = "9"$

Annotated Parse Tree



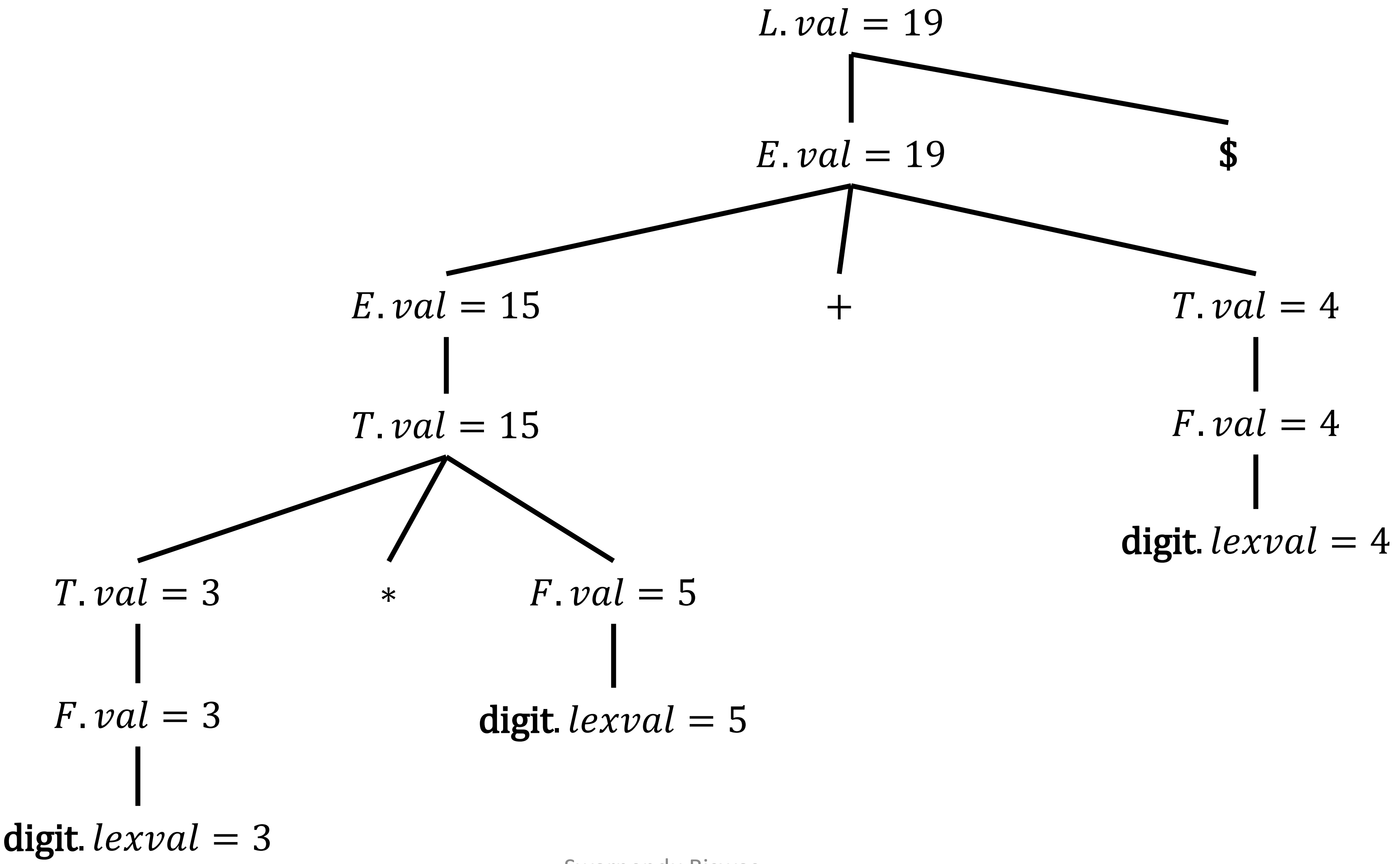
S-Attributed Definition

- An SDD that involves only synthesized attributes is called S-attributed definition
  - Each rule computes an attribute for the head nonterminal from attributes taken from the body of the production
- Synthesized attributes can be evaluated in a bottom-up order
  - An S-attributed SDD can be implemented naturally in conjunction with an LR parser

Example SDD

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

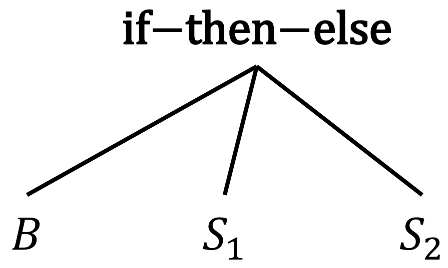
Annotated Parse Tree for $3 * 5 + 4\ \$$



Abstract Syntax Tree (AST)

- Condensed form of a parse tree used for representing language constructs
  - ASTs do not check for string membership in the language for a grammar
  - ASTs represent relationships between language constructs, do not bother with derivations

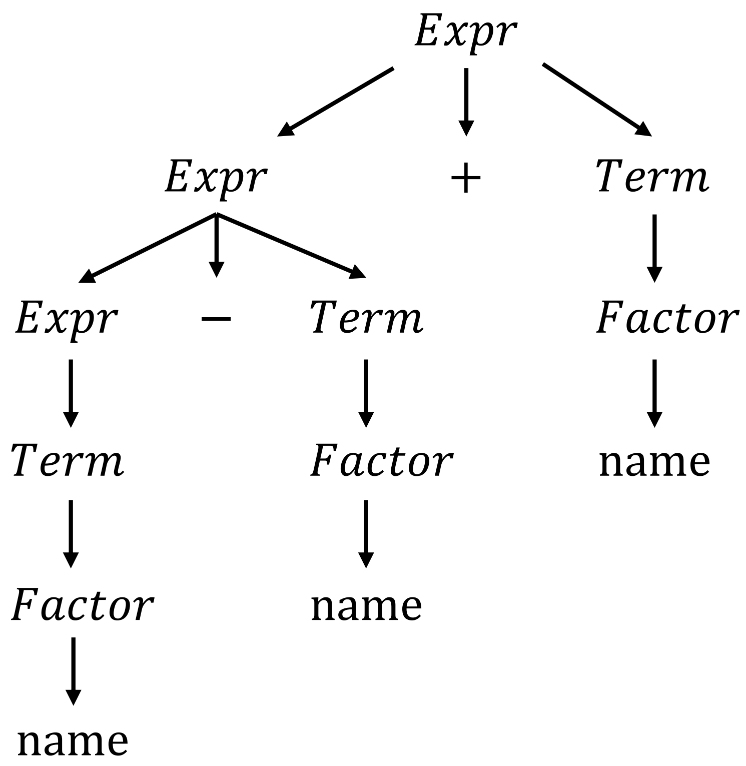
$$S \rightarrow \textbf{if } P \textbf{ then } S_1 \textbf{else } S_2$$

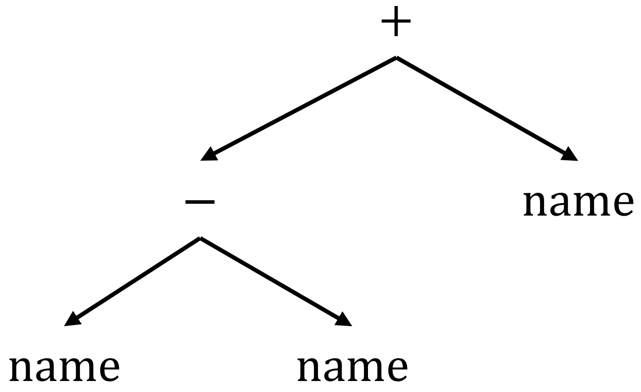


- Parse trees are also called concrete syntax trees

Parse Tree and Abstract Syntax Tree

Parse Tree	**Abstract Syntax Tree**



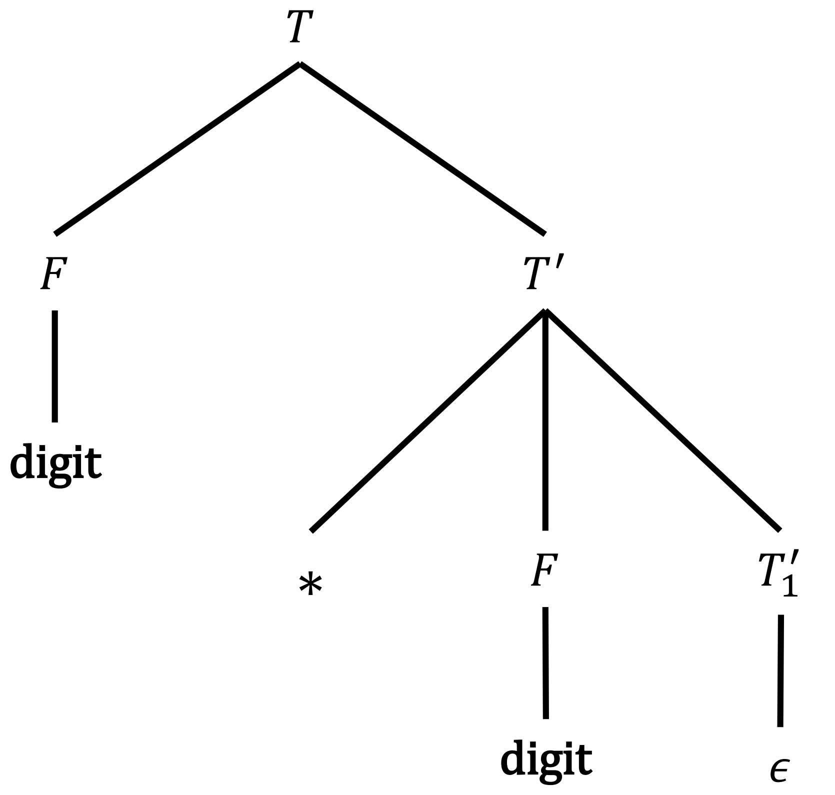


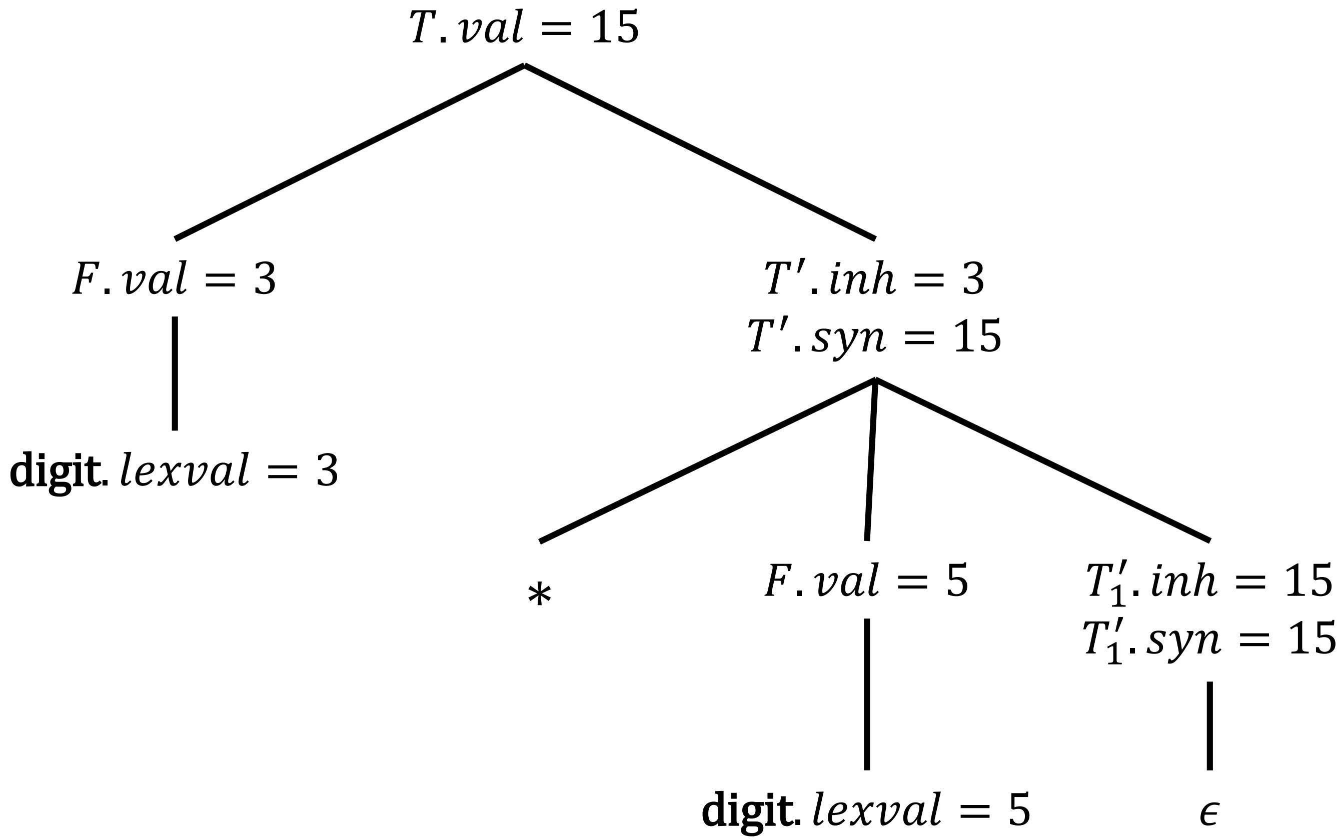
Inherited Attributes

- Useful when the structure of the parse tree **does not match** the abstract syntax of the source code

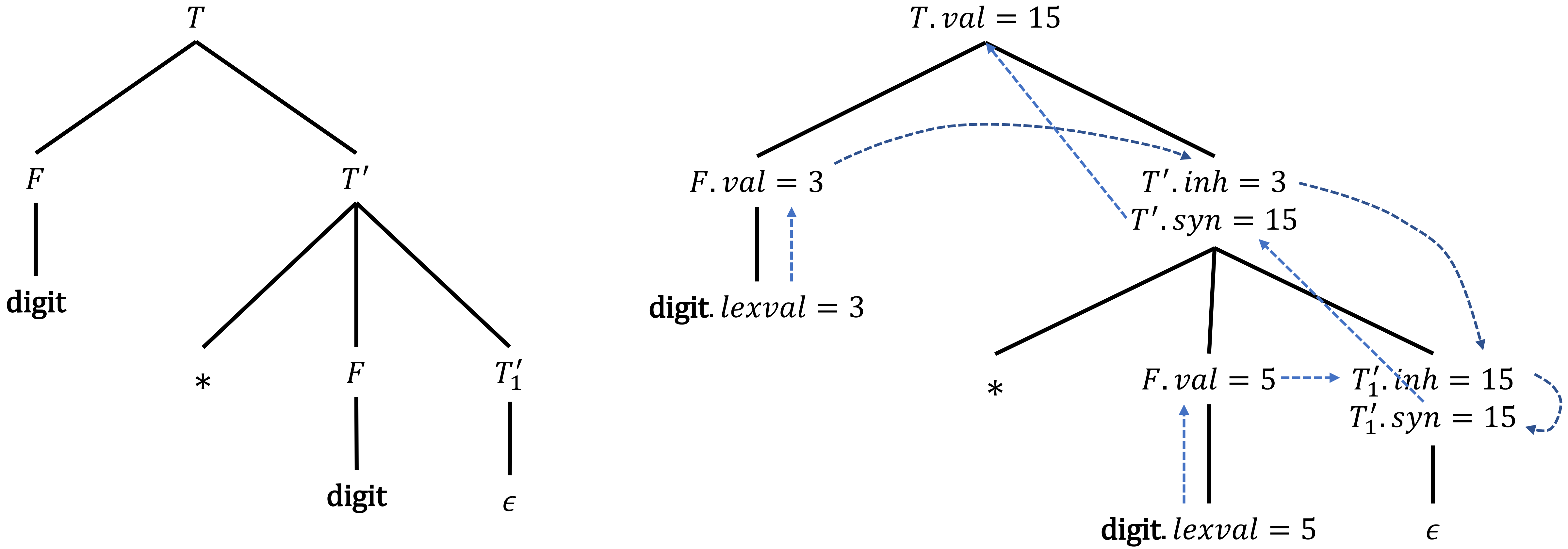
Production	Semantic Rules
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow * FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**.$lexval$

Parse Tree and Annotated Parse Tree for $3 * 5$





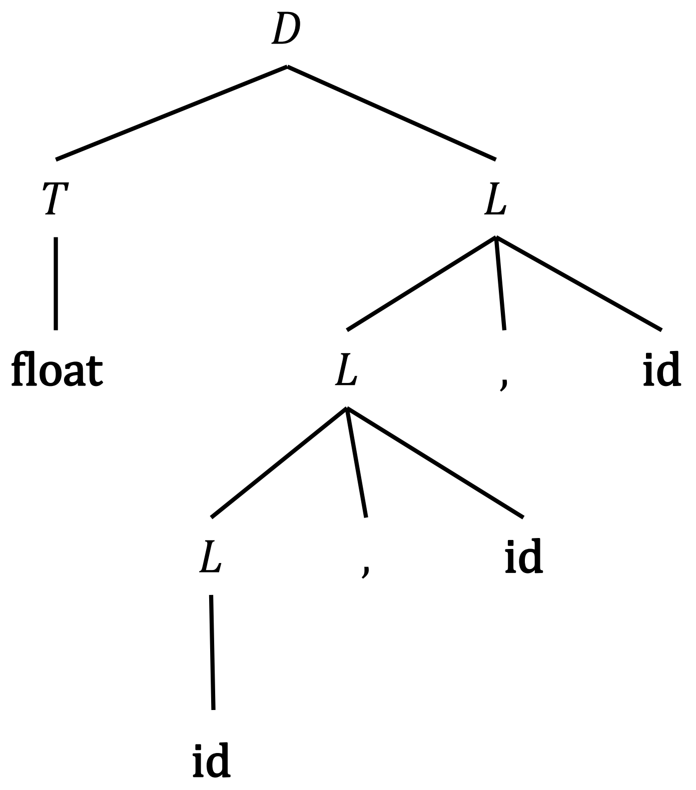
Parse Tree and Annotated Parse Tree for $3 * 5$



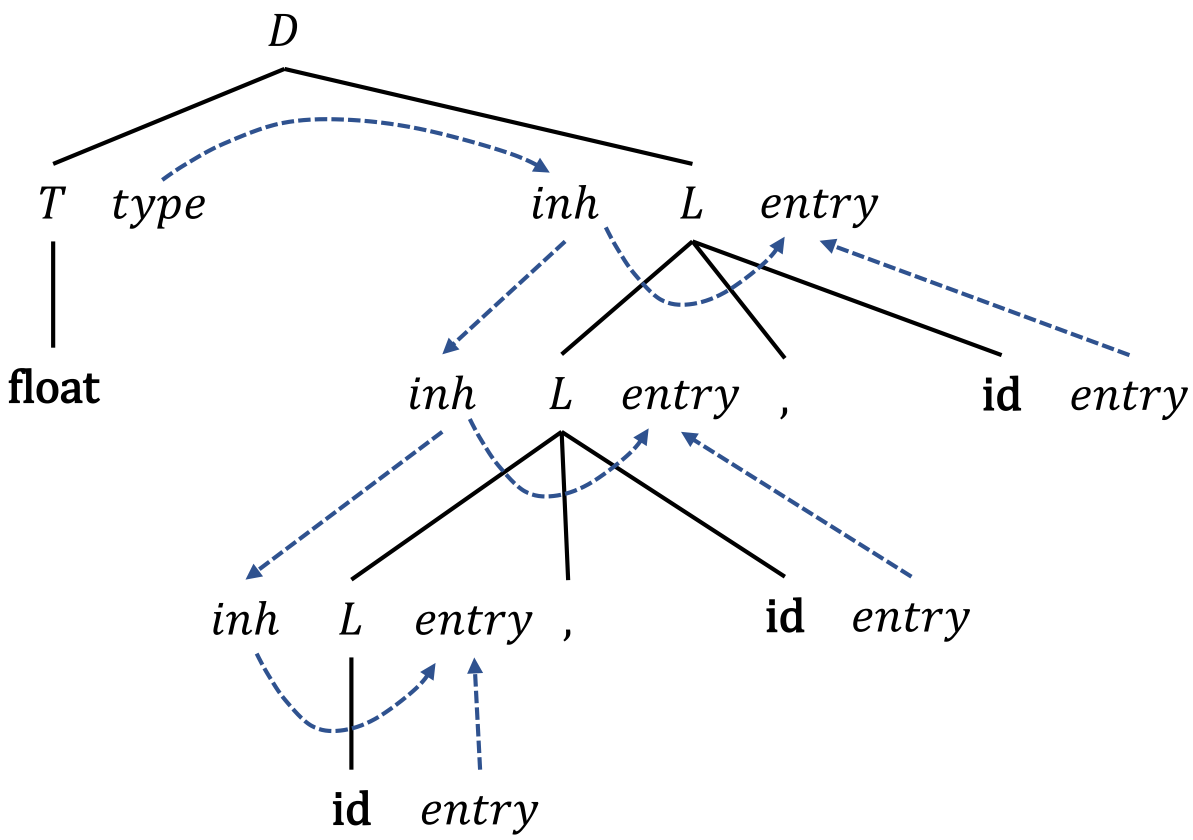
Another Example

Production	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow$ **float**	$T.type =$ **float**
$T \rightarrow$ **int**	$T.type =$ **int**
$L \rightarrow L_1,$ **id**	$L_1.in = L.in;\ addtype($**id**$.entry, L.in)$
$L \rightarrow$ **id**	$addtype($**id**$.entry, L.in)$

Parse Tree for "**float** x, y, z"



Annotated Parse Tree for **float** x, y, z



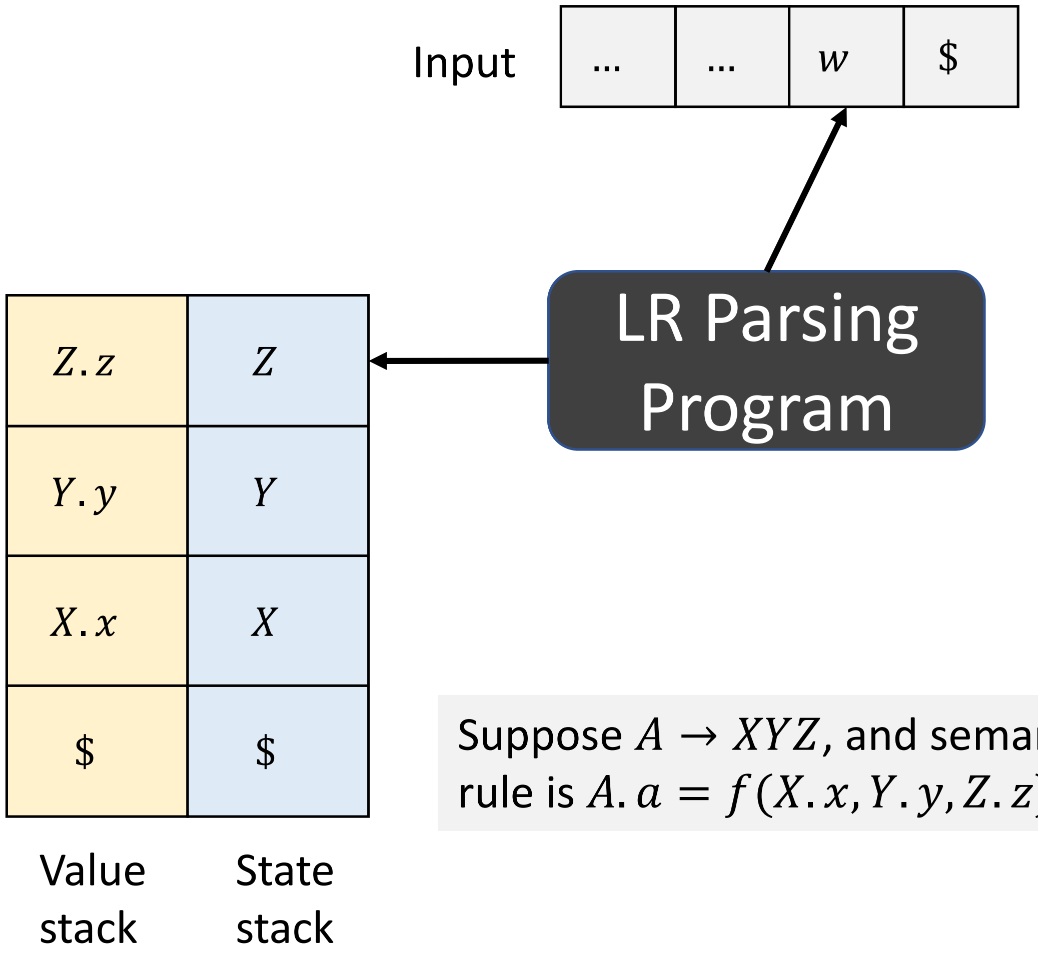
Evaluating S-Attributed Definitions

- Attributes can be evaluated with a postorder traversal of the parse tree

```
postorder(N) {
    for (each child C of N, from left to right)
        postorder(C)
    evaluate the attributes associated with node N
```

Bottom-up Evaluation of S-Attributed Definitions

- Can be computed during bottom-up parsing
- On reduce action, value of new synthesized attribute is computed from the attributes on the stack
  - Extend stack to hold the values also



Suppose $A \rightarrow XYZ$, and semantic rule is $A.a = f(X.x, Y.y, Z.z)$

Example SDD

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

Bottom-up Evaluation of S-Attributed Definitions

Value	State	Input	Action
$\$$	$\$$	$3 * 5 + 4\$$	Shift
$\$3$	$\$\mathbf{digit}$	$* 5 + 4\$$	Reduce by $F \rightarrow \mathbf{digit}$
$\$3$	$\$F$	$* 5 + 4\$$	Reduce by $T \rightarrow F$
$\$3$	$\$T$	$* 5 + 4\$$	Shift
$\$3$	$\$T *$	$5 + 4\$$	Shift
$\$3\ 5$	$\$T * \mathbf{digit}$	$+4\$$	Reduce by $F \rightarrow \mathbf{digit}$
$\$3\ 5$	$\$T * F$	$+4\$$	Reduce by $T \rightarrow T * F$
$\$15$	$\$T$	$+4\$$	Reduce by $E \rightarrow T$
$\$15$	$\$E$	$+4\$$	Shift
$\$15$	$\$E +$	$4\$$	Shift
$\$15\ 4$	$\$E + \mathbf{digit}$	$\$$	Reduce by $F \rightarrow \mathbf{digit}$
$\$15\ 4$	$\$E + F$	$\$$	Reduce by $T \rightarrow F$
$\$15\ 4$	$\$E + T$	$\$$	Reduce by $E \rightarrow E + T$
$\$19$	$\$E$	$\$$	…

L-Attributed Definitions

- Each attribute must be either
    I. Synthesized
    II. Suppose $A \rightarrow X_1X_2 \dots X_n$ and $X_i.a$ is an inherited attribute. $X_i.a$ can be computed using only inherited attributes from A, or either inherited or synthesized attributes associated with $X_1, \dots, X_{i-1}$
    III. Inherited or synthesized attributes associated with X_i

Production	Semantic Rules
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow * FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**$.lexval$

Is the SDD S- or L-attributed?

Production	Semantic Rules
$A \rightarrow BC$	$A.a = B.b_1$ $B.b_2 = f(A.a, C.c)$

Production	Semantic Rules
$A \rightarrow BC$	$B.i = f_1(A.i)$ $C.i = f_2(B.s)$ $A.s = f_3(C.s)$

Production	Semantic Rules
$A \rightarrow BC$	$C.i = f_4(A.i)$ $B.i = f_5(C.s)$ $A.s = f_6(B.s)$

Syntax-Directed Translation

Associating Semantic Rules with Productions

- Syntax-directed definition (SDD)
  - Abstract high-level specification which hides implementation details
  - Explicit order of evaluation is not specified but there should not be any circularity
- Syntax-directed translation (SDT)
  - Program fragments are embedded as semantic actions in the body of a production
    - Executable specification of an SDD
  - Indicates order in which semantic rules are to be evaluated

$$rest \rightarrow +term \; \{ \; print("+") \; \} \; rest_1$$

SDT for Infix to Postfix Translation

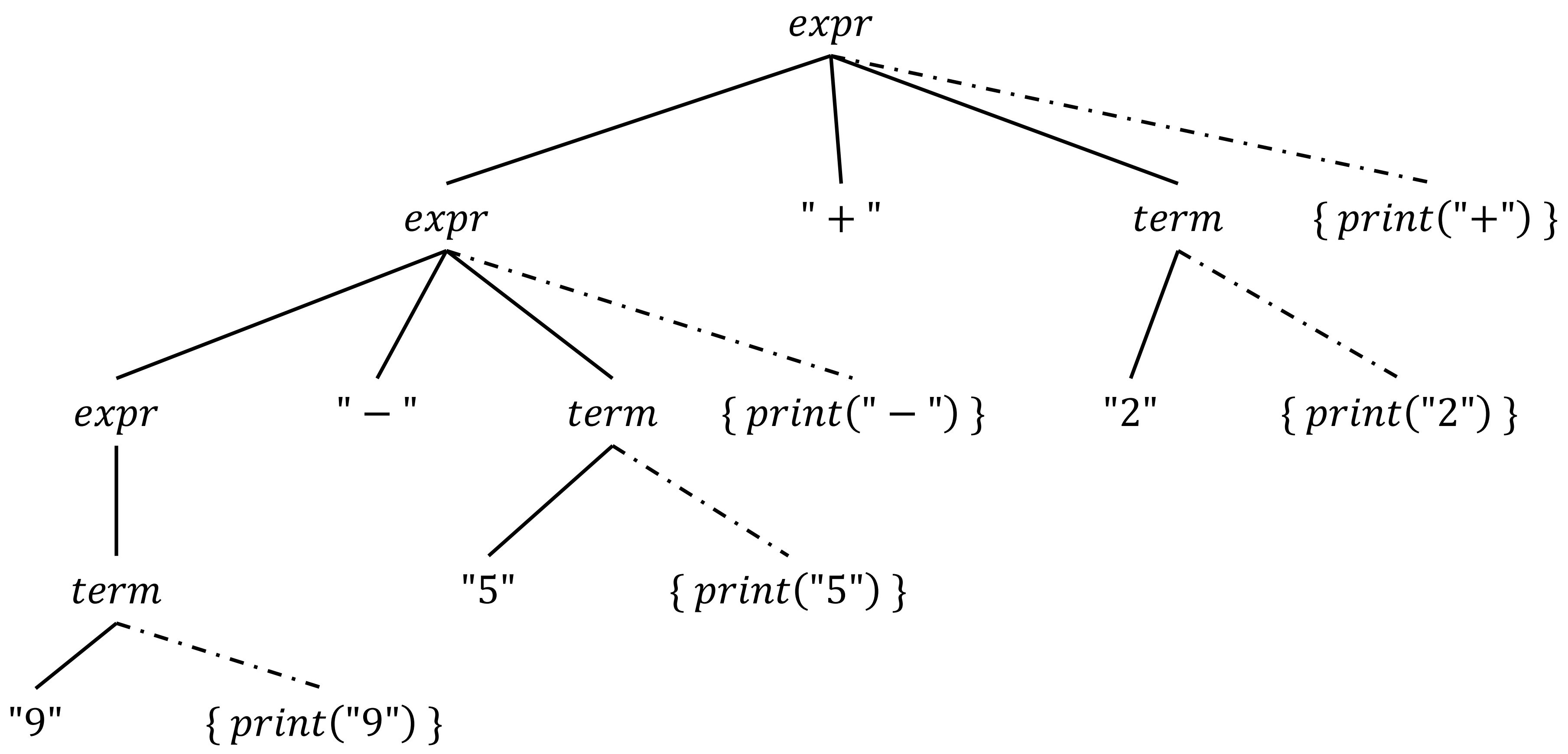
SDD

Production	Semantic Rules
$expr \rightarrow expr_1 + term$	$expr.code = expr_1.code \| \| term.code \| \| "+"$
$expr \rightarrow expr_1 - term$	$expr.code = expr_1.code \| \| term.code \| \| " - "$
$expr \rightarrow term$	$expr.code = term.code$
$term \rightarrow 0 \mid 1 \mid \ldots \mid 9$	$term.code = "0"$ $term.code = "1"$ … $term.code = "9"$

SDT

Production	Semantic Rules
$expr \rightarrow expr_1 + term$	$\{ print("+") \}$
$expr \rightarrow expr_1 - term$	$\{ print(" - ") \}$
$expr \rightarrow term$	
$term \rightarrow 0 \mid 1 \mid \ldots \mid 9$	$\{ print("0") \}$ $\{ print("1") \}$ … $\{ print("9") \}$

SDT Actions


expr
expr
" + "
term
{ print("+") }
expr
" − "
term
{ print(" − ") }
"2"
{ print("2") }
term
"5"
{ print("5") }
"9"
{ print("9") }

SDDs and SDTs

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

- Evaluation of the semantic rules may
  - Generate code
  - Save information in the symbol table
  - Issue error messages
  - Perform any other activity

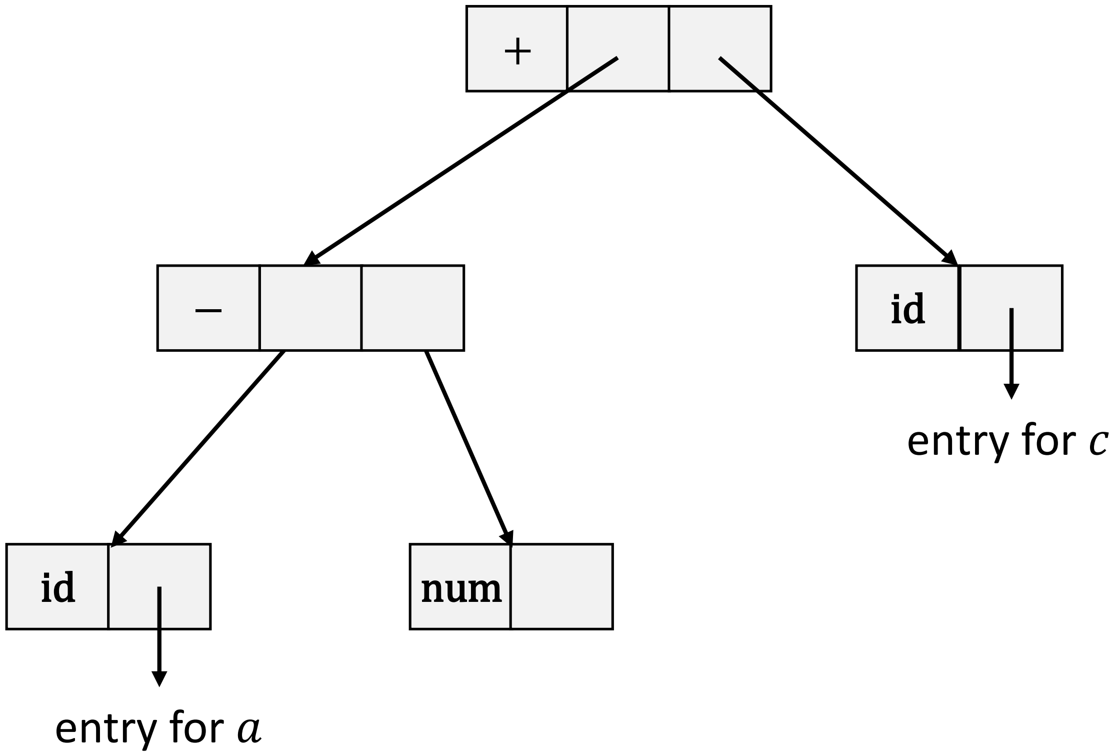
Construction of AST for Expressions

- Idea: Construct subtrees for subexpressions by creating an operator and operand nodes
- Operator: `mknode(op, left, right)`
  - Create an operator node with label op and two pointers to left and right children
- Identifier: `mkleaf(id, entry)`
  - Create an identifier node with label id and a pointer to symbol table entry for the id
- Number: `mkleaf(num, val)`
  - Create a number node with label num and the value

Creating an AST

- Following sequence of function calls create an AST for $a - 4 + c$

1. $p_1 = mkleaf(\mathbf{id}, entrya)$
2. $p_2 = mkleaf(\mathbf{num}, 4)$
3. $p_3 = mknode(\text{“}-\text{”}, p_1, p_2)$
4. $p_4 = mkleaf(\mathbf{id}, entryc)$
5. $p_5 = mknode(\text{“}+\text{”}, p_3, p_4)$



SDT for Constructing Syntax Trees

Production	Semantic Rules
$E \rightarrow E_1 + T$	$E.nptr = mknode("+", E_1.nptr, T.nptr)$
$E \rightarrow E_1 - T$	$E.nptr = mknode(" - ", E_1.nptr, T.nptr)$
$E \rightarrow T$	$E.nptr = T.nptr$
$T \rightarrow (E)$	$T.nptr = E.nptr$
$T \rightarrow$ **id**	$T.nptr = mkleaf(\textbf{id}, \textbf{id}.entry)$
$T \rightarrow$ **num**	$T.nptr = mkleaf(\textbf{num}, \textbf{num}.val)$

Construction of AST for $a - 4 + c$