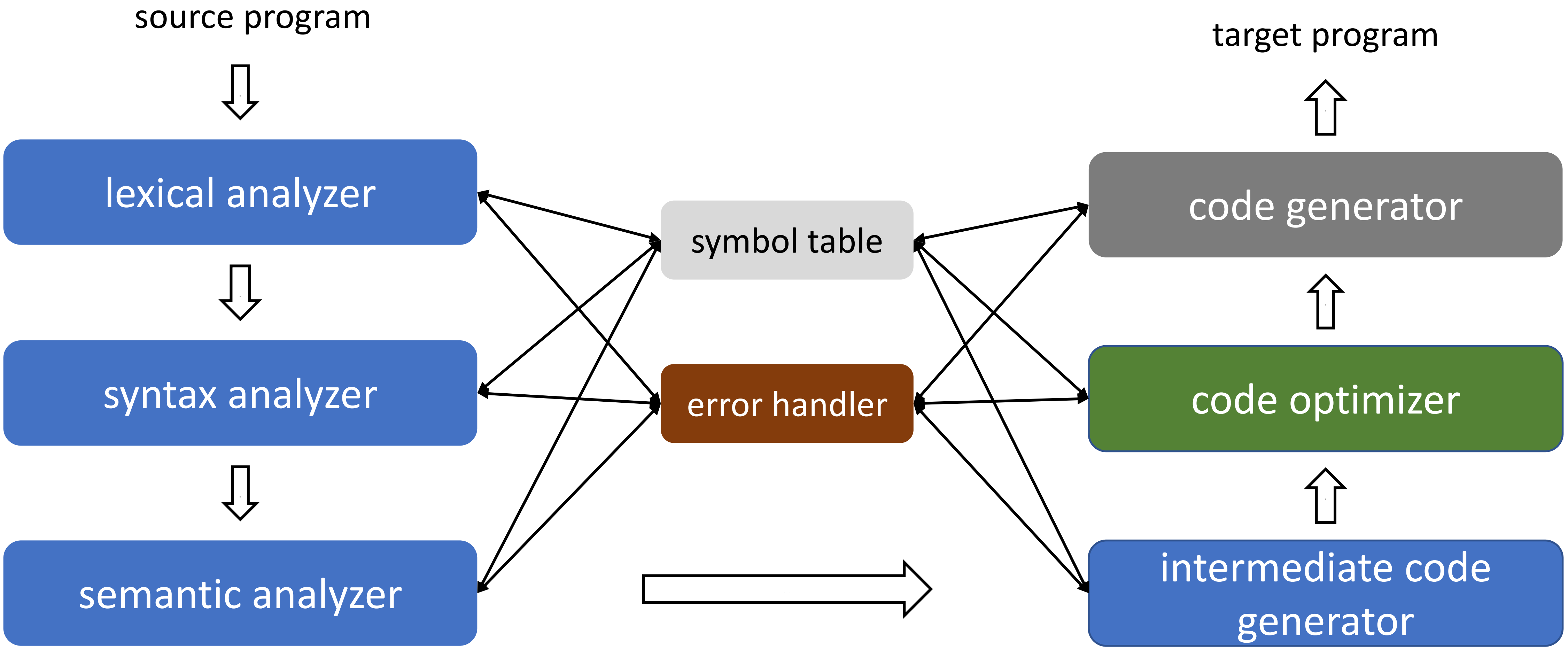
CS 335: Semantic Analysis

Swarnendu Biswas

Semester 2022-2023-II
CSE, IIT Kanpur

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

An Overview of Compilation



source program

lexical analyzer

syntax analyzer

semantic analyzer

symbol table

error handler

intermediate code generator

code optimizer

code generator

target program

Beyond Scanning and Parsing

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

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

```
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;
}
int main() {
  int p, a[10], b[10];
  p = dot_prod(a, b);
  return 0;
}
```

Example static semantic checks that a compiler can perform:

- p, `a`, and b are declared before use
- Number and type of the parameters of `dot_prod( )` are the same in its declaration and use
- Types of p and return type of `dot_prod( )` match

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

$$ProcedureBody \rightarrow Declarations\ Executables$$

Ensures variable declarations go before their uses

- CFGs only deal with syntactic categories and structure
- Enforcing the “declare before use” rule requires knowledge that cannot be encoded in a CFG
- Grammar can specify the positions in an expression where a variable name may occur, but can enforce the “declare before use” rule
  - CFG cannot match one instance of a variable name with another
  - Programming languages also allow to include declarations within executable statements

Questions That Compiler Needs to Answer

Questions

- Has a variable been declared?
- What is the type and size of a variable?
- Is the variable a 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

$x \leftarrow y + z$

$x \leftarrow a + b$

$y, z, a,$ and b can have different types, so code generation may need to be different

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
- Static semantics of languages can be checked at compile time
  - For example, ensure variable are declared before their uses, check that each expression has a correct type, and programs must have valid locations to transfer the control flow.

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;
}
int main() {
  int p; int a[10], b[10];
  p = dot_prod(a, b);
  return 0;
}
```

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
  - Building **efficient** parsers is challenging

- 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 attributes and semantic rules to evaluate the attributes
  - Attributes may be of any type: numbers, strings, pointers to structures
  - Attributes are associated with nodes in the parse tree, and each instance of a grammar symbol in the parse tree has an associated attribute

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

- Attribute grammars are SDDs with no side effects
  - Help track context-sensitive information via attributes

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 (e.g., $X.a$ 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

Attribute Grammar for Signed Binary Numbers

Consider a grammar for signed binary numbers

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

Build an 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

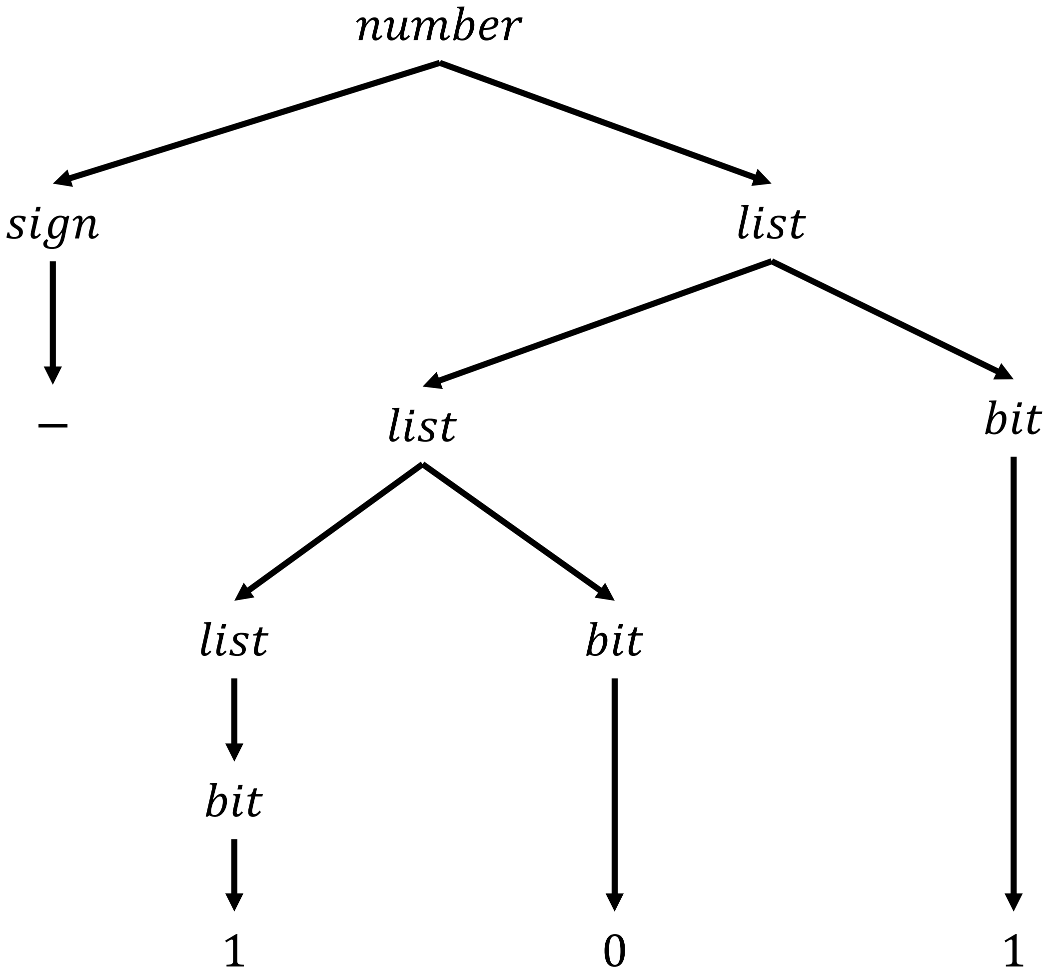
Attribute Grammar for Signed Binary Numbers

Production	Attribute Rule

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

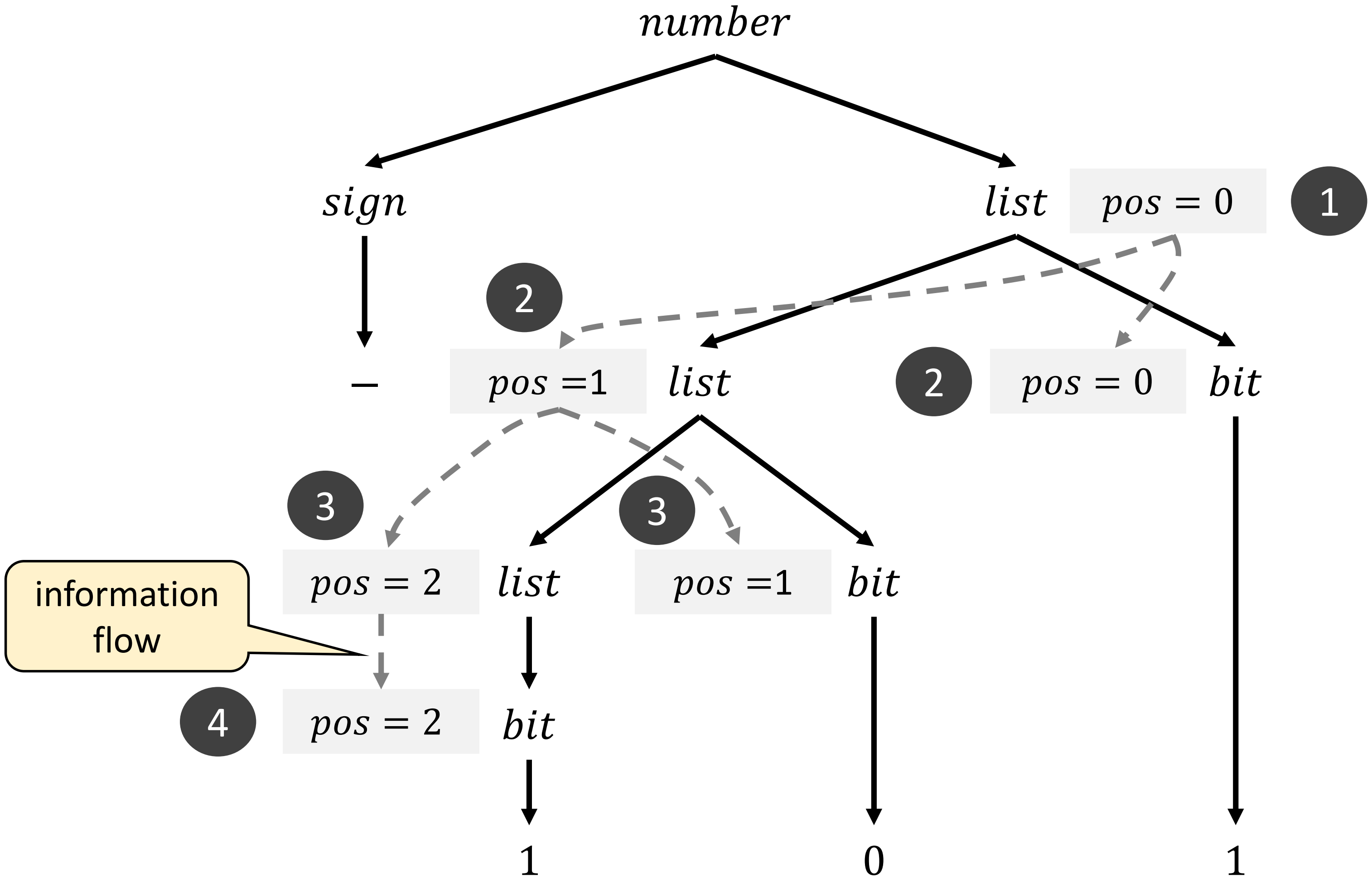
$sign \rightarrow +$	$sign.neg =$ false
$sign \rightarrow -$	$sign.neg =$ true
$list \rightarrow bit$	$bit.pos = list.pos$ $list.val = bit.val$
$list_0 \rightarrow list_1 bit$	$list_1.pos = list_0.pos + 1$ $bit.pos = list_0.pos$ $list_0.val = list_1.val + bit.val$
$bit \rightarrow 0$	$bit.val = 0$
$bit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

Parse Tree for -101



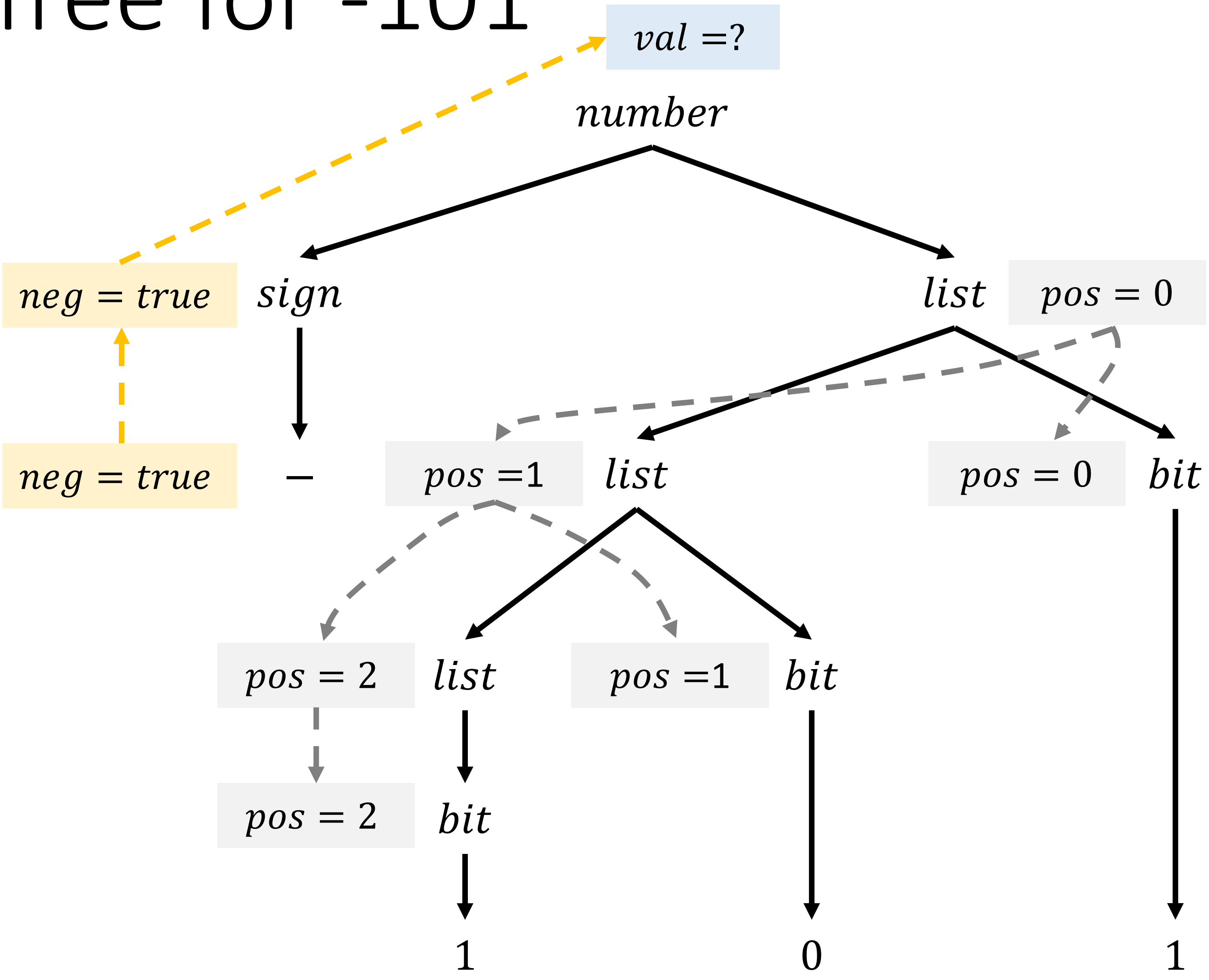
Annotated Parse Tree for -101

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



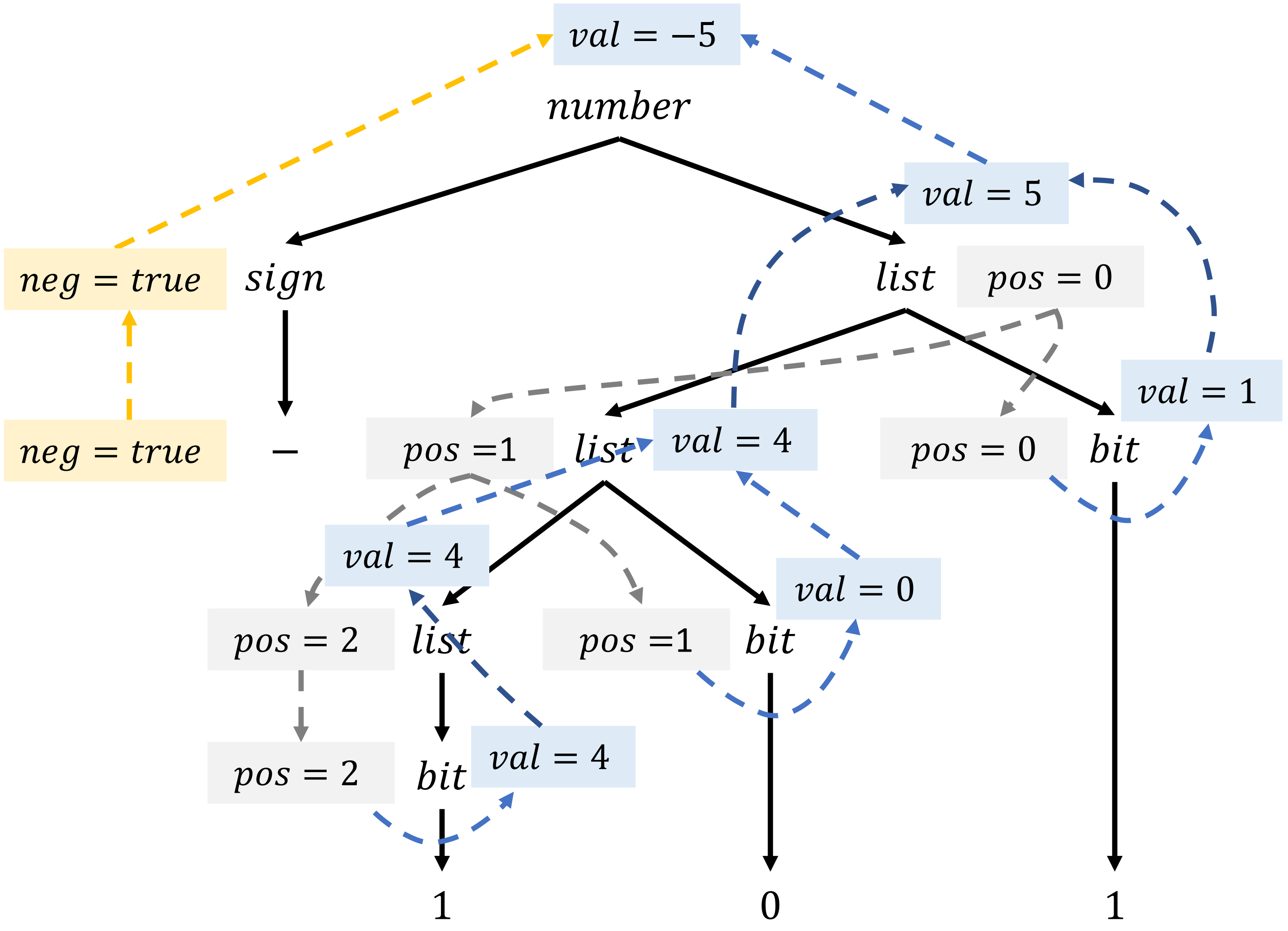
Annotated Parse Tree for -101

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



Annotated Parse Tree for -101

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



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** (e.g., val and neg)
- 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** (e.g., pos)
- 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 $b = f(c_1, c_2, \dots, c_k)$
  - 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

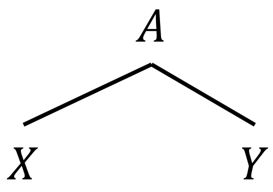
- Start symbol cannot have inherited attributes
- Terminals can have synthesized attributes, but not inherited attributes
  - Attributes for terminals have lexical values that are supplied by the lexical analyzer

Dependency Graph

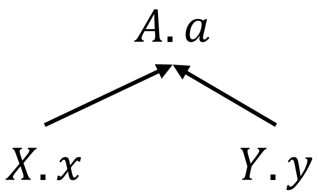
- If an attribute b depends on an attribute c then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes are depicted by a directed graph called dependency graph
- Annotated parse tree shows the values at attributes, while the dependency graph shows how the values need to be computed

Dependency Graph

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

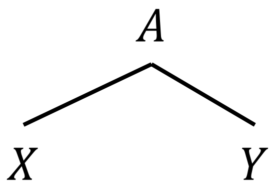


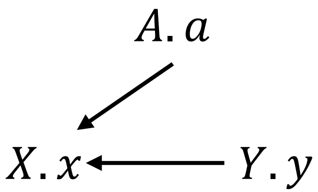
Parse tree



Dependency graph

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



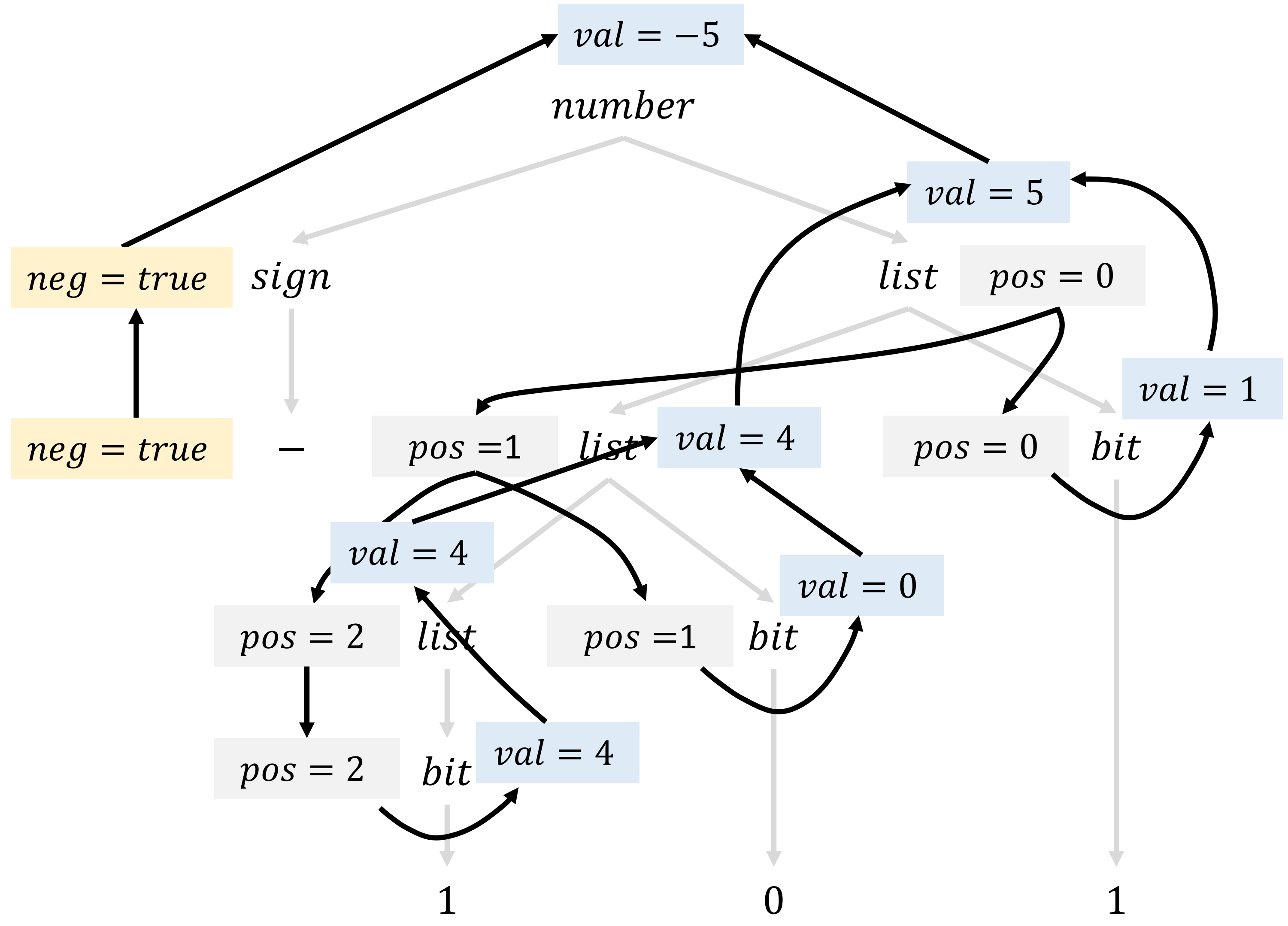


Construct Dependency Graph

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

Example of a Dependence Graph



nodes are attributes

Evaluating an SDD

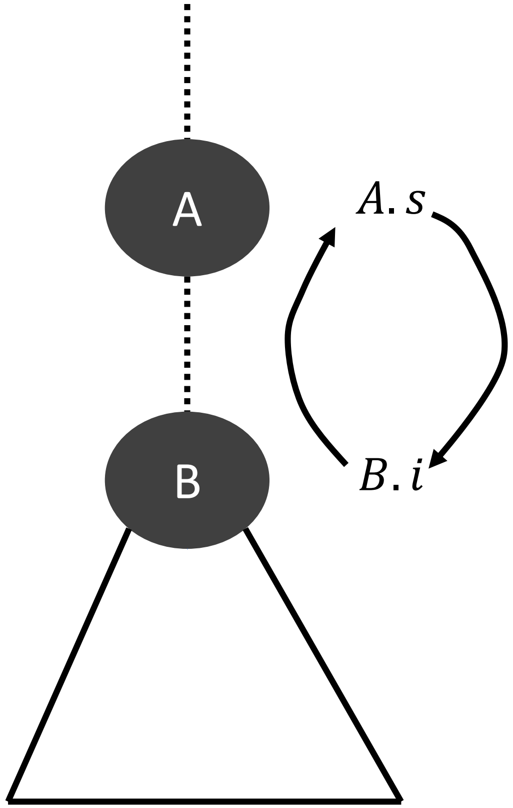
- In what order do we evaluate attributes in an implementation?
  - SDDs do not specify any order of evaluation
  - We must evaluate all the attributes upon which the attribute of a node depends

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

Circular Dependency of Attributes

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

A compiler must deal with circularity appropriately for attribute grammars



Evaluating an SDD

- Parse tree method
  - In the absence of cycles, use topological sort of the dependency graph to find the evaluation order
  - Any topological sort of dependency graph gives a valid partial order in which semantic rules must be evaluated
  - Each rule executes as soon as all its input operands are available
- Rule-based method
  - Semantic rules are analyzed and order of evaluation is predetermined
  - E.g., evaluate $list.pos$ first and $list.val$ later
- Oblivious method
  - Evaluation order ignores the semantic rules, makes repeated left-to-right and right-to-left passes until all attributes have values

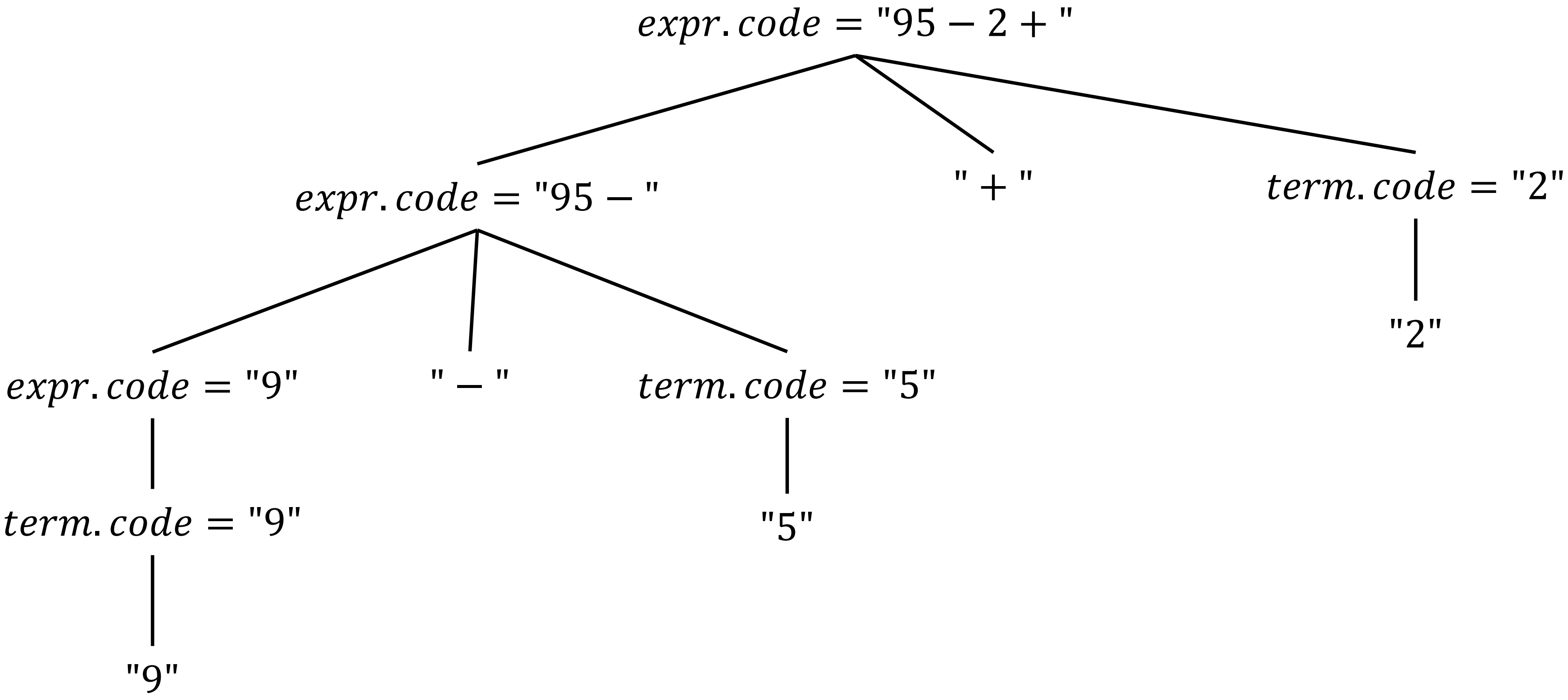
Postfix Notation

- 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



$expr.code = "95 - 2 + "$

$expr.code = "95 - "$

$" + "$

$term.code = "2"$

$"2"$

$expr.code = "9"$

$" - "$

$term.code = "5"$

$term.code = "9"$

$"5"$

$"9"$

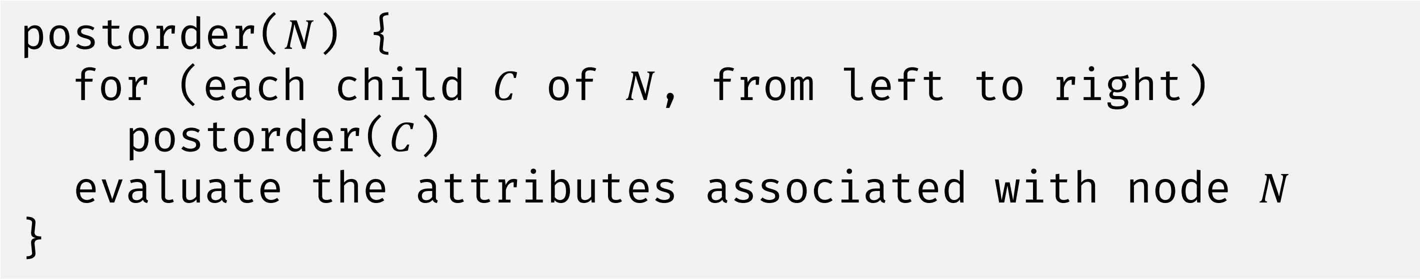
Types of SDDs

- Cycles need to be avoided since the compiler can no longer meaningfully proceed with evaluation
- Expensive to identify whether an arbitrary SDD will have cycles

- S-attributed and L-attributed SDDs **guarantee** no cycles

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
- Semantic rules in a S-attributed definition can be evaluated by a bottom-up or postorder traversal of the parse tree
  - An S-attributed SDD can be implemented naturally in conjunction with an LR parser



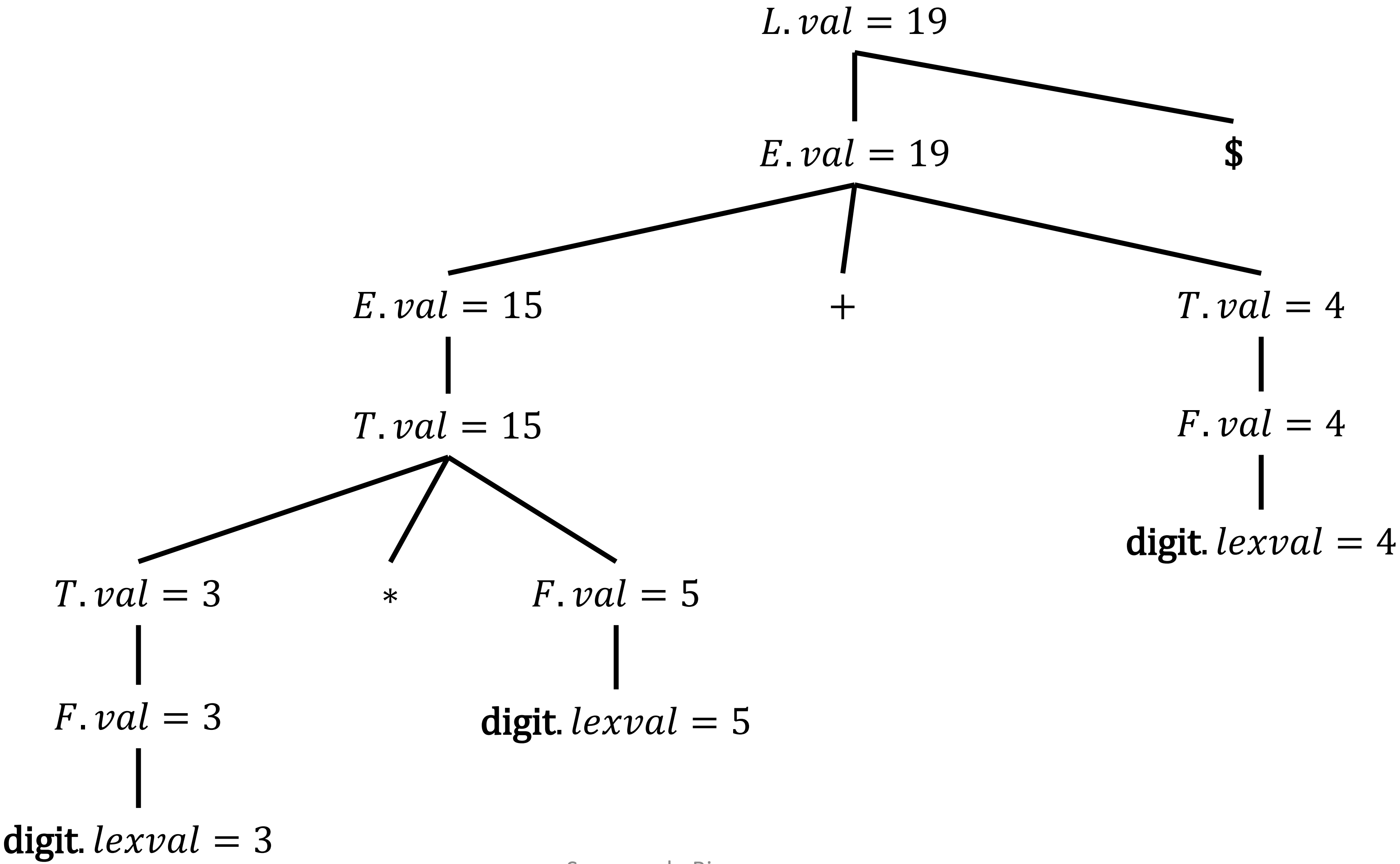
```
postorder($N$) {
  for (each child $C$ of $N$, from left to right)
    postorder($C$)
  evaluate the attributes associated with node $N$
}
```

Example of S-Attributed Definition

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

all attributes are synthesized

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



$L.val = 19$

$E.val = 19$

$

$E.val = 15$

$+$

$T.val = 4$

$T.val = 15$

$F.val = 4$

digit. $lexval = 4$

$T.val = 3$

$*$

$F.val = 5$

$F.val = 3$

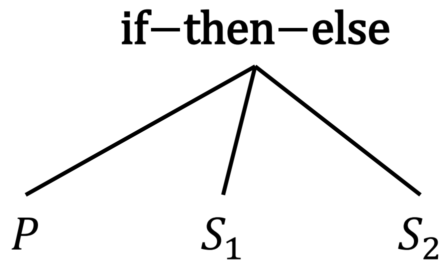
digit. $lexval = 5$

digit. $lexval = 3$

Abstract Syntax Tree (AST)

- Condensed form of a parse tree used for representing language constructs
  - Each leaf is an operand and non-leaf nodes represent operators
  - ASTs do not check for string membership in the language for a grammar
  - ASTs represent relationships between language constructs, do not bother with derivations

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

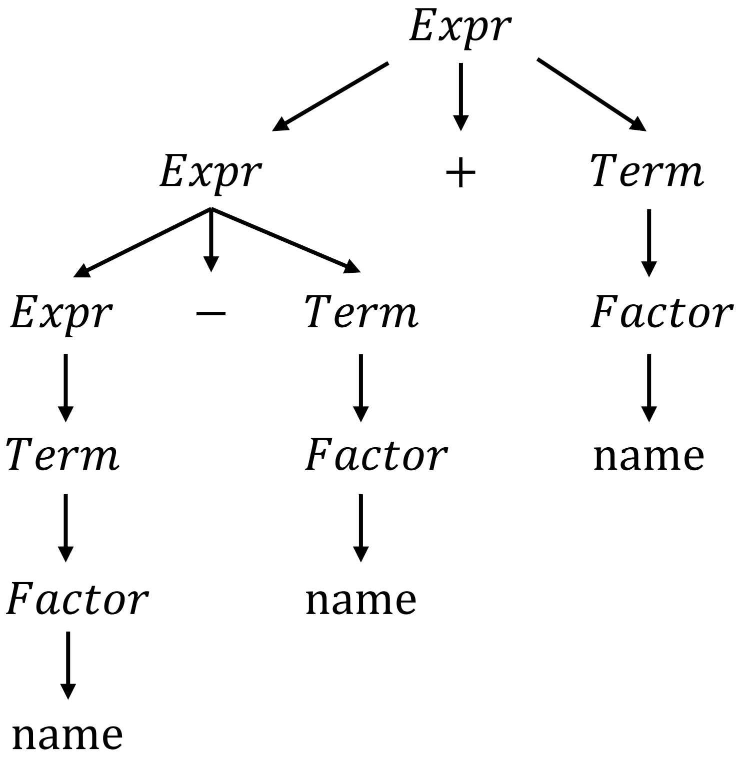
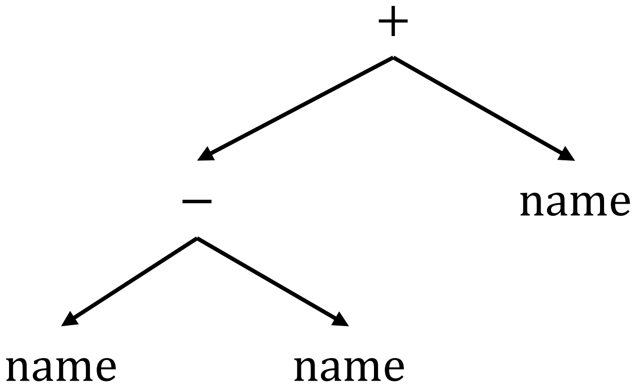


if−then−else

P S_1 S_2

- Parse trees are also called concrete syntax trees

Parse Tree vs Abstract Syntax Tree

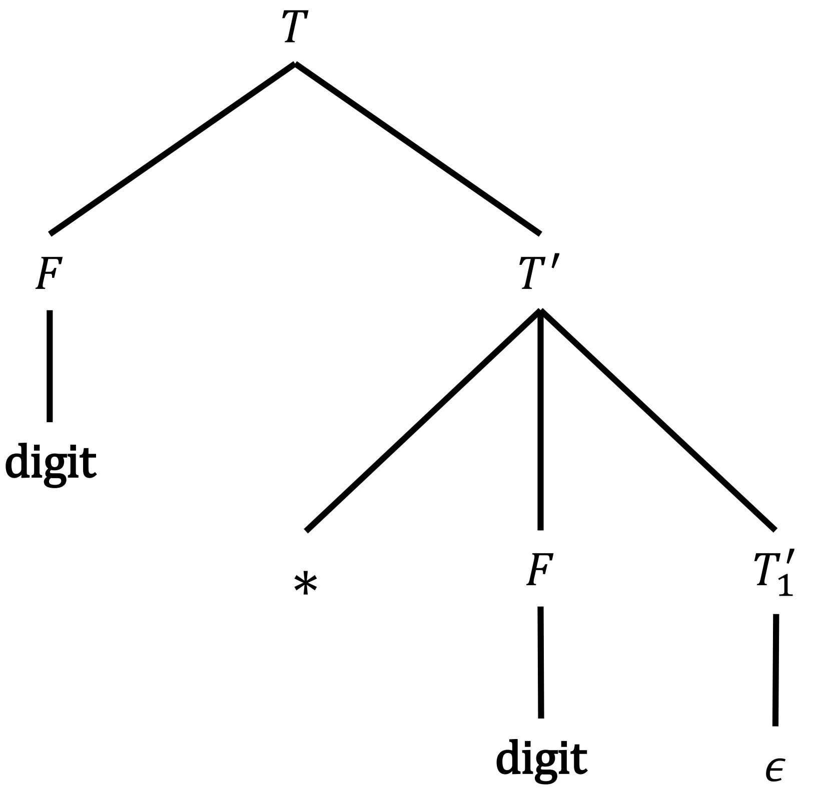
Parse Tree	**Abstract Syntax Tree**
	

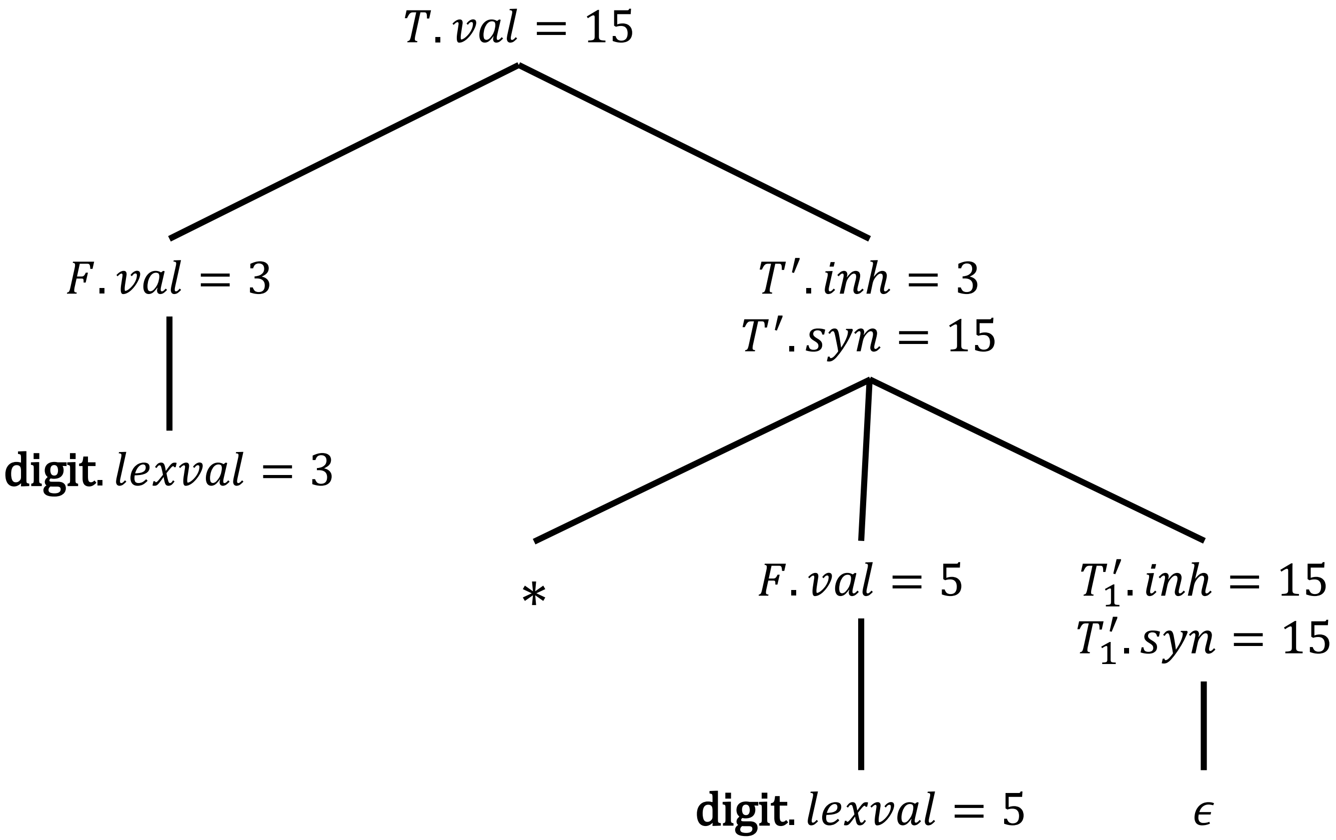
Inherited Attributes

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

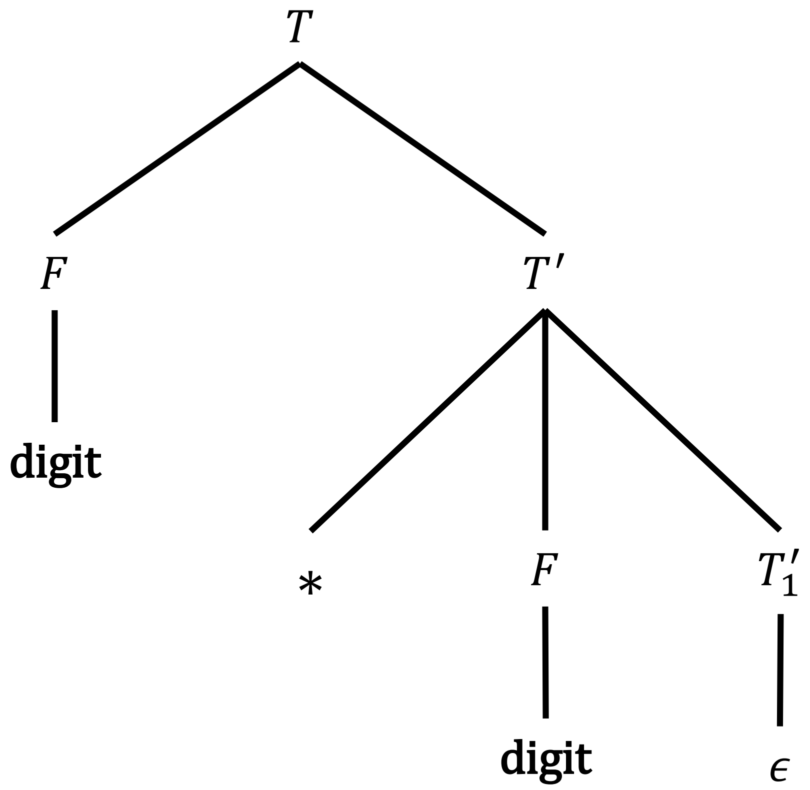
Production	Semantic Rules
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow * FT_1'$	$T_1'.inh = T'.inh \times F.val$ $T'.syn = T_1'.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow$ **digit**	$F.val =$ **digit**.$lexval$

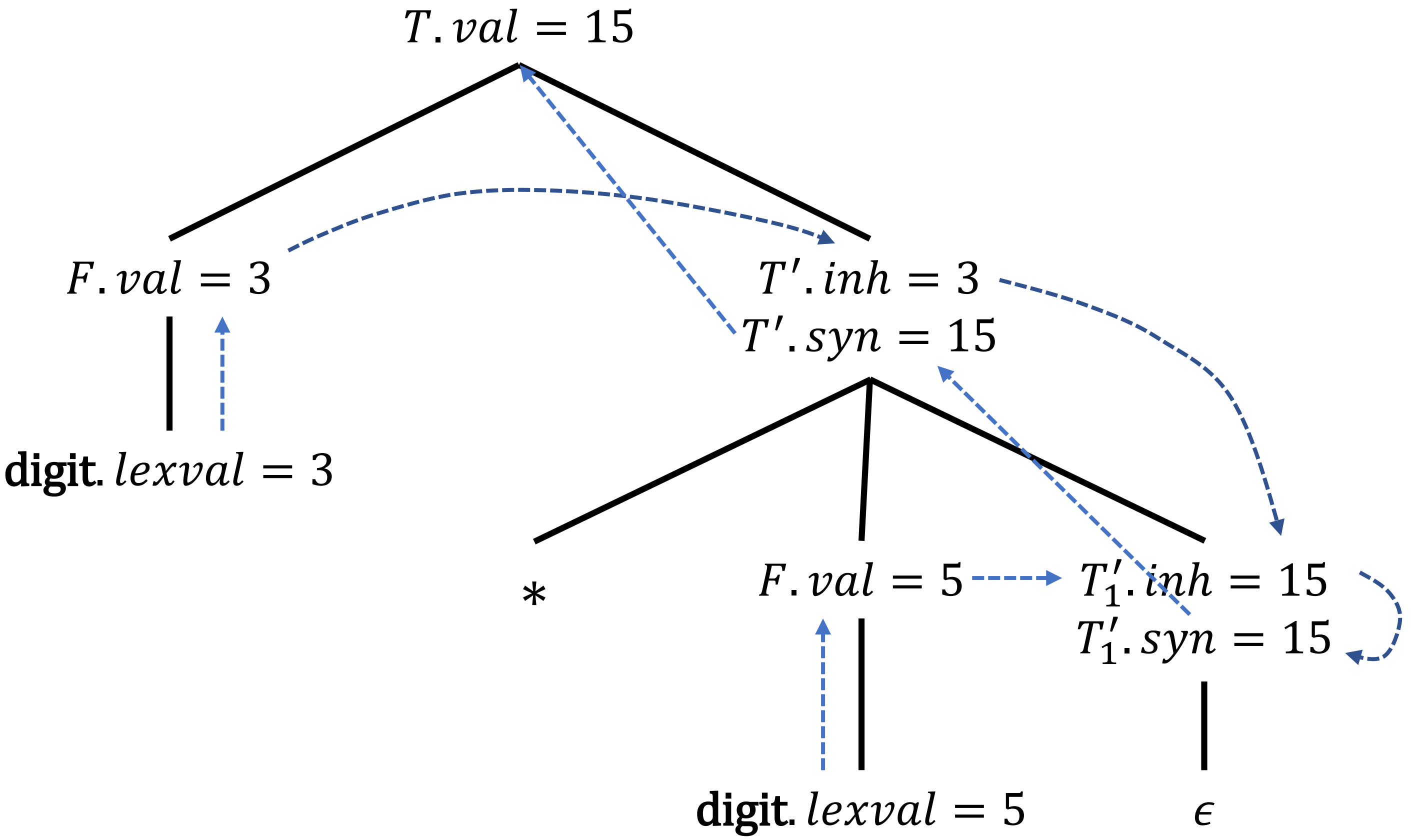
Parse Tree and Annotated Parse Tree for $3 * 5$





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



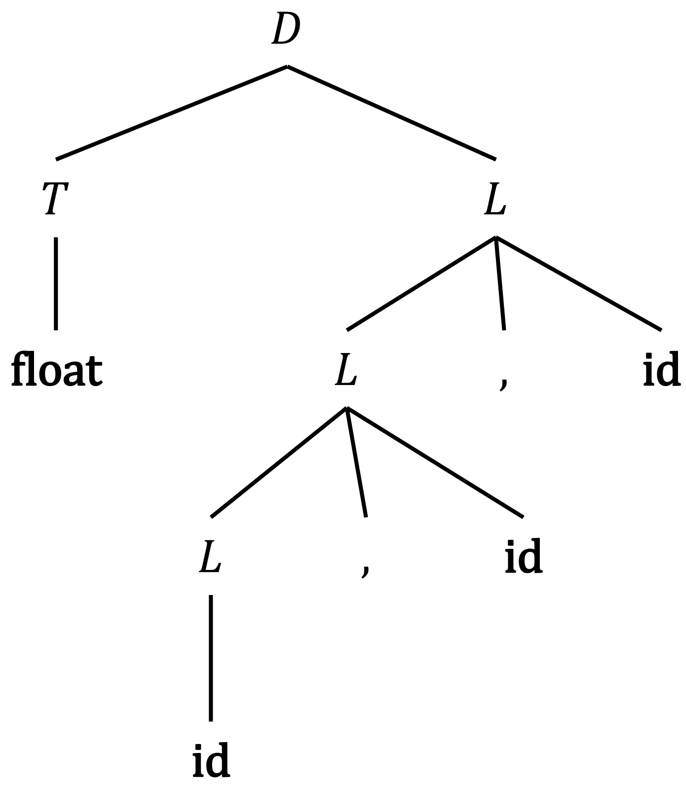


Another Example

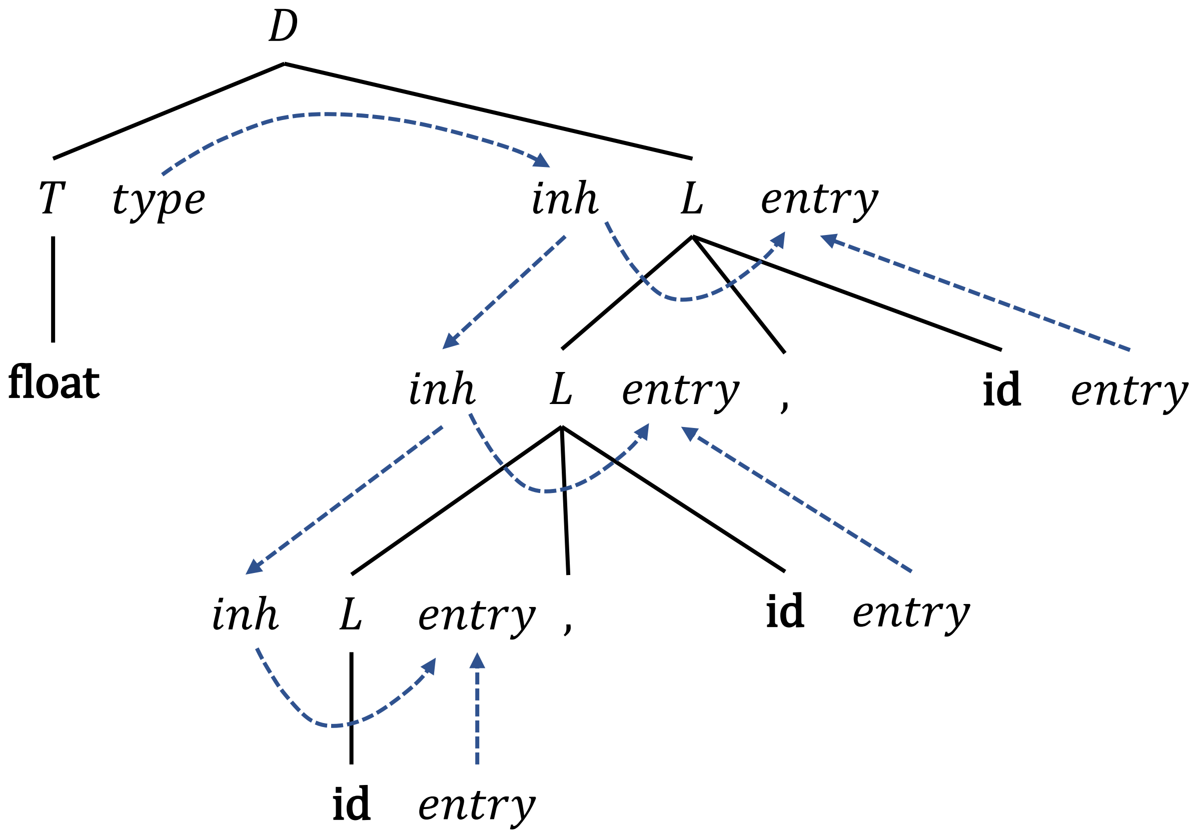
Production	Semantic Rules
$D \rightarrow TL$	$L.in = T.type$
$T \rightarrow$ **float**	$T.type =$ **float**
$T \rightarrow$ **int**	$T.type =$ **int**
$L \rightarrow L_1,$ **id**	$L_1.in = L.in;\ addtype(\mathbf{id}.entry, L.in)$
$L \rightarrow$ **id**	$addtype(\mathbf{id}.entry, L.in)$

$addtype()$ installs $L.in$ as the type of the symbol table object pointed to by **id**.$entry$ (implies a side effect)

Parse Tree for “**float** x, y, z”



Dependency Graph for **float** x, y, z



Notes about Inherited Attributes

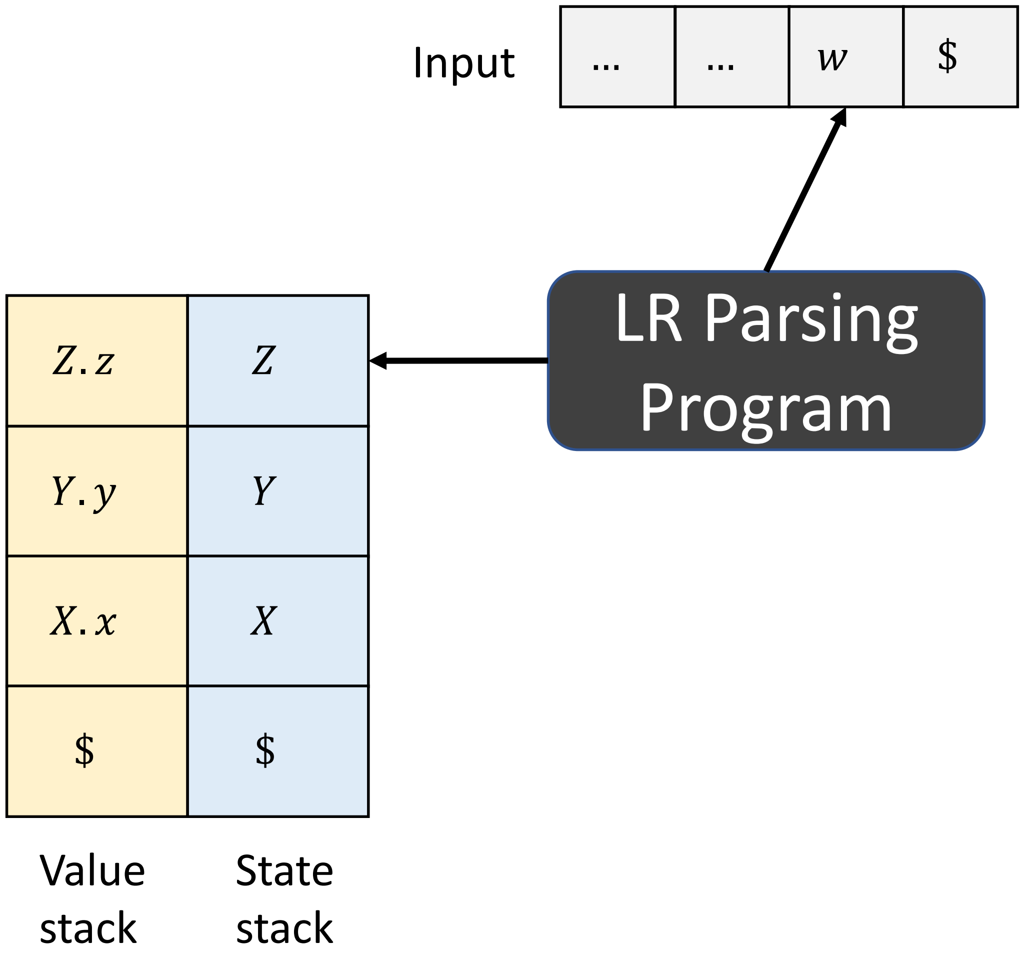
- Always possible to rewrite a SDD to use only synthesized attributes
  - Inherited attributes can be simulated with synthesized attributes and helper functions
- May be more logical to use both synthesized and inherited attributes
- Inherited attributes usually cannot be evaluated by a simple preorder traversal of the parse tree
  - Attributes may depend on both left and right siblings!
  - Attributes that do not depend on right children can be evaluated by a preorder traversal

---

How can an inherited attribute be simulated using a synthesized attribute?

Bottom-up Evaluation of S-Attributed Definitions

- Suppose $A \rightarrow XYZ$, and semantic rule is $A.a = f(X.x, Y.y, Z.z)$
- Attributes can be computed during bottom-up parsing
  - Extend the stack to hold values
  - On reduction, value of new synthesized attribute $A.a$ is computed from the attributes on the stack



Example S-Attributed Definition

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	Symbols	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, or
  ii. Suppose $A \rightarrow X_1 X_2 \ldots X_n$ and $X_i.a$ is an inherited attribute. $X_i.a$ can be computed using
    a) Only inherited attributes from A, or
    b) Either inherited or synthesized attributes associated with $X_1, \ldots, X_{i-1}$, or
    c) 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**$.lexval$

Are these SDDs 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
$A \rightarrow BC$	$C.i = f_4(A.i)$ $B.i = f_5(C.s)$ $A.s = f_6(B.s)$

S-Attributed and L-Attributed Definitions

Every S-attributed grammar is also a L-attributed grammar

All L-attributed grammars are not S-attributed

Challenges with Attribute Grammars

i. Rules only involve local information (i.e., attributes pertaining to symbols in the production)
    - Needs additional attributes and copy rules to use non-local information, which increases memory and run-time overhead

ii. Results can be scattered across attributes in the parse tree

iii. Works in conjunction with a parse tree or an AST
    - A compiler implementation may not build either

Syntax-Directed Translation

Recap SDDs

- Syntax-directed definition (SDD)
  - Defines a set of attributes and translations at every node of the parse tree, output is available at the root
  - Functional style which hides implementation details
    - Evaluation order is not specified among multiple attributes for a production
    - Only requirement is there should not be any circularity

Associating Semantic Rules with Productions

- Syntax-directed translation (SDT)
  - **Program fragments** are embedded as semantic actions in production body
    - Generates code while parsing
  - Indicates order in which semantic actions are to be evaluated

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

  - Executable specification of an SDD, easier to implement, and can be more efficient since the compiler can avoid constructing a parse tree and a dependency graph
  - Yacc/Bison uses translation schemes

SDT for Infix to Postfix Translation

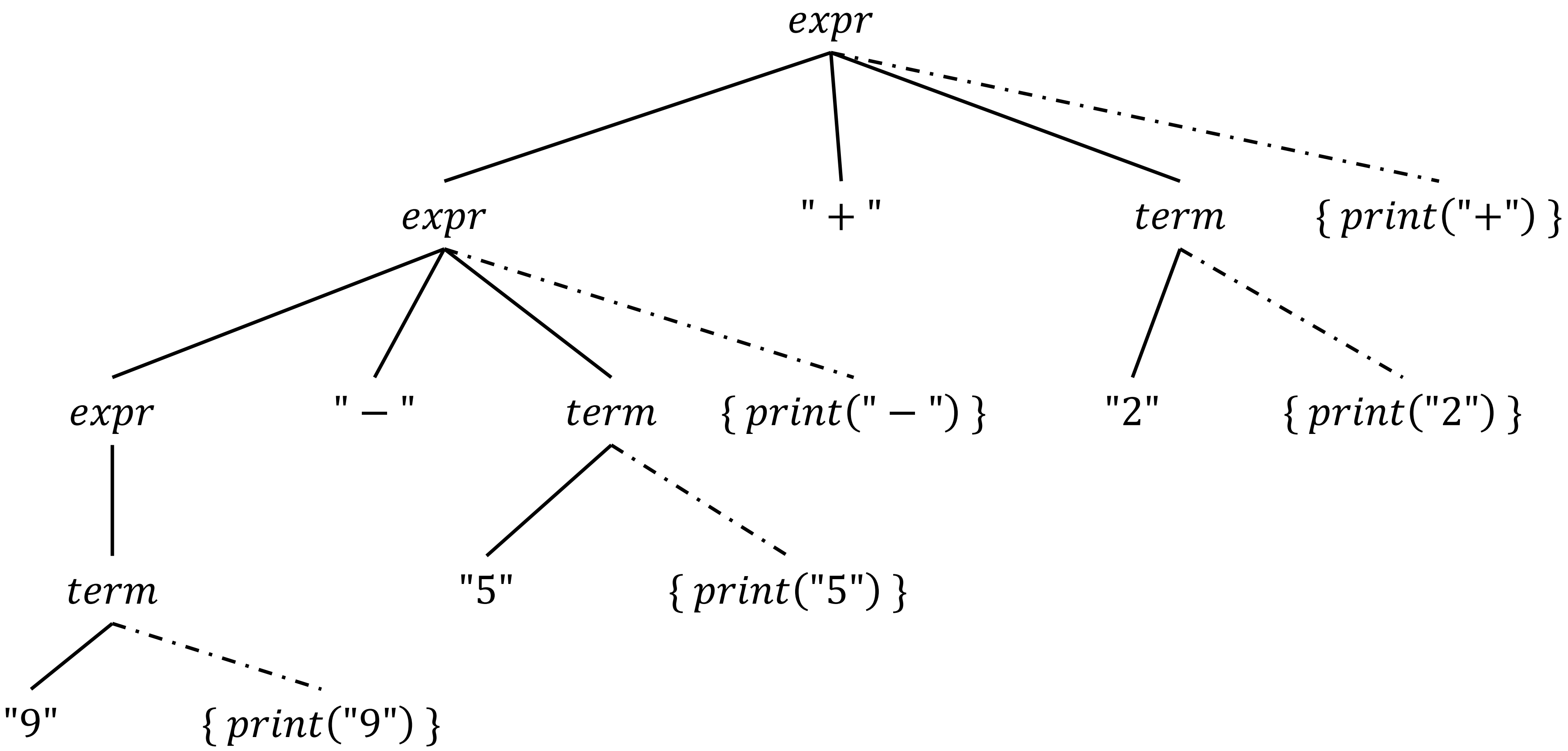
SDD

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

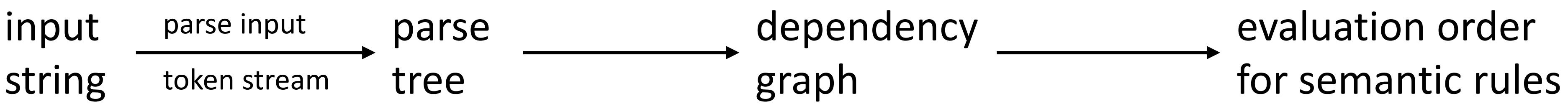
SDT

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

SDT Actions


$expr$
$expr$
$"+"$
$term$
$\{ print("+") \}$
$expr$
$"-"$
$term$
$\{ print("-") \}$
$"2"$
$\{ print("2") \}$
$term$
$"5"$
$\{ print("5") \}$
$"9"$
$\{ print("9") \}$

SDDs and SDTs



input string $\xrightarrow[\text{token stream}]{\text{parse input}}$ 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

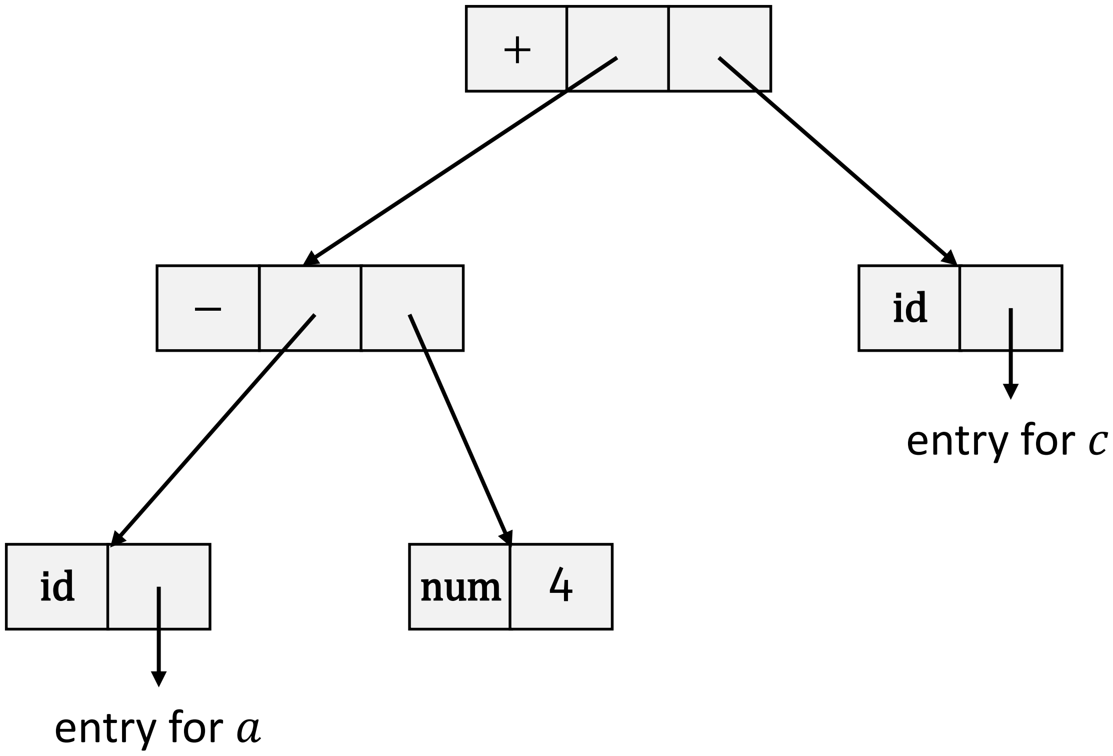
Construction of AST for Expressions

- **Idea**: Construct subtrees for subexpressions by creating an operator and operand nodes
- Internal node: $Node(op, c_1, c_2, \ldots, c_k)$
  - Create a node with label op, and k fields for k children
- Leaf node: $Leaf(op, val)$
  - Create a node with label op, and val is the lexical value

Creating an AST

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

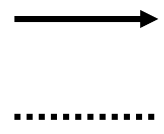
1. $p_1 = \textbf{new}\ Leaf(\textbf{id}, entrya)$
2. $p_2 = \textbf{new}\ Leaf(\textbf{num}, 4)$
3. $p_3 = \textbf{new}\ Node(\text{“}-\text{”}, p_1, p_2)$
4. $p_4 = \textbf{new}\ Leaf(\textbf{id}, entryc)$
5. $p_5 = \textbf{new}\ Node(\text{“}+\text{”}, p_3, p_4)$

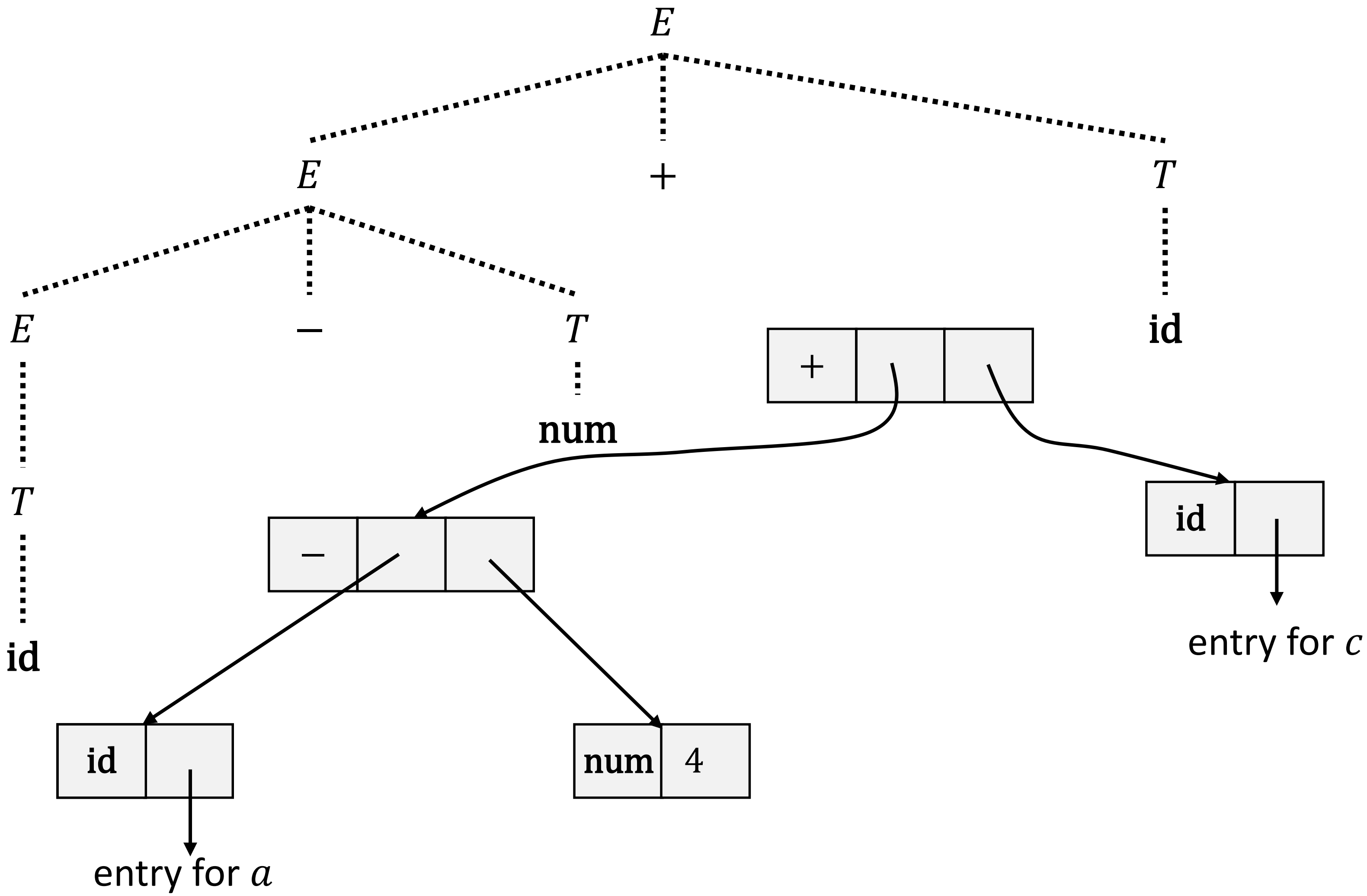


S-Attributed Definition for Constructing Syntax Trees

Production	Semantic Action
$E \rightarrow E_1 + T$	$E.node = \mathbf{new}\ Node("+", E_1.node, T.node)$
$E \rightarrow E_1 - T$	$E.node = \mathbf{new}\ Node("-", E_1.node, T.node)$
$E \rightarrow T$	$E.node = T.node$
$T \rightarrow (E)$	$T.node = E.node$
$T \rightarrow$ **id**	$T.node = \mathbf{new}\ Leaf(\mathbf{id}, \mathbf{id}.entry)$
$T \rightarrow$ **num**	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$

Construction of AST for $a - 4 + c$

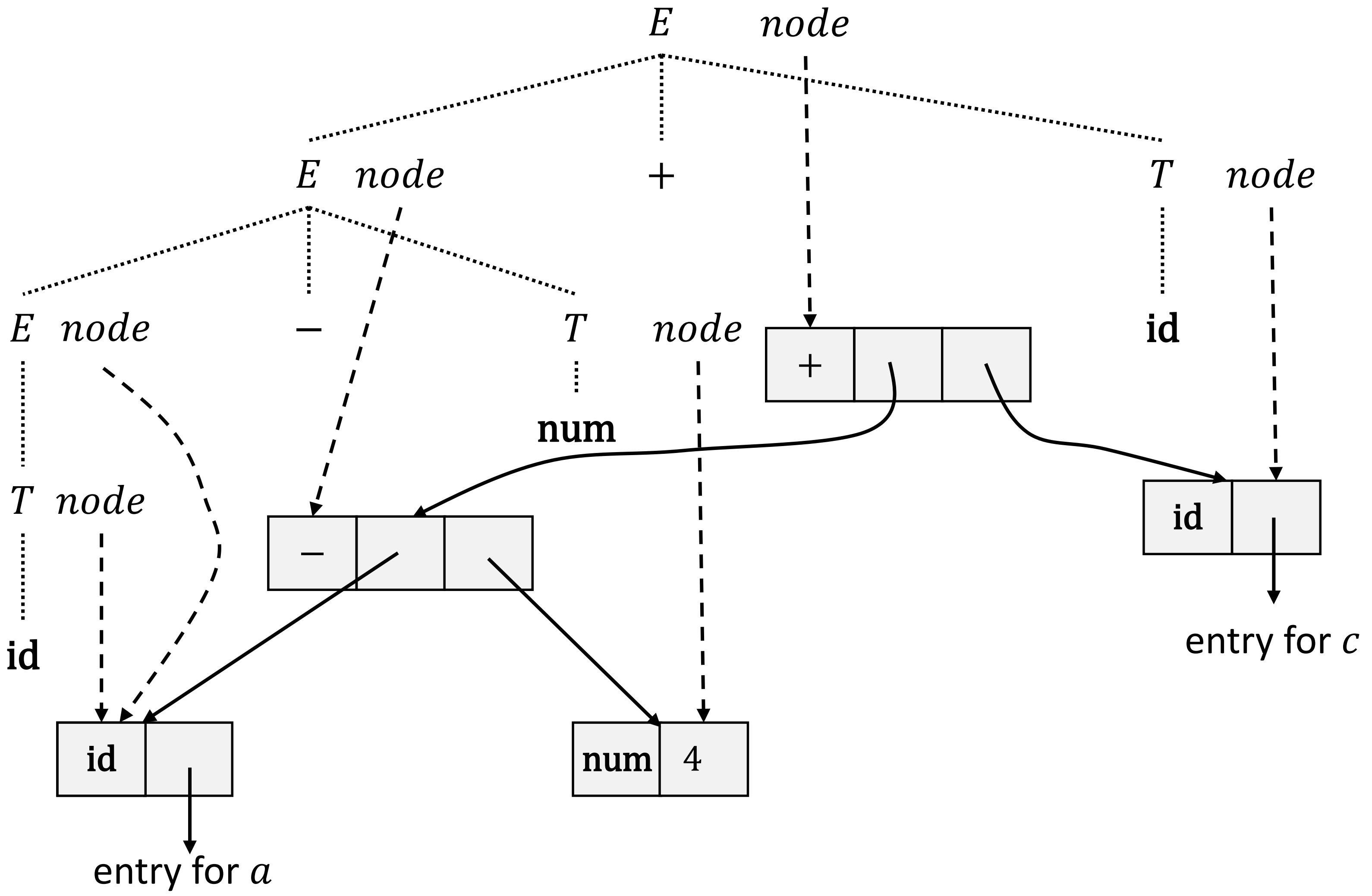




Construction of AST for $a - 4 + c$

AST edge

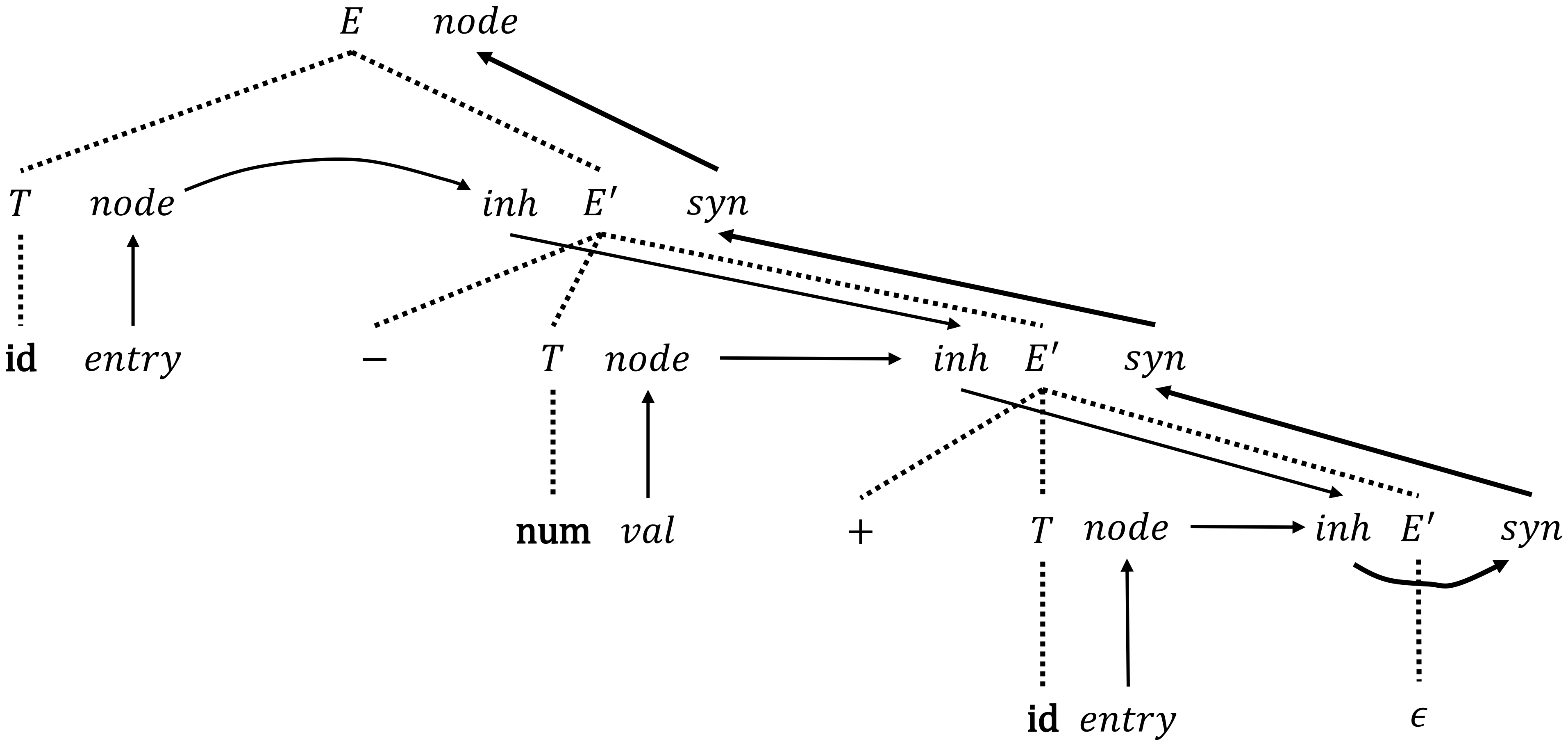
Parse Tree edge



L-Attributed Definition for Constructing Syntax Trees

Production	Semantic Action
$E \rightarrow TE'$	$E.node = E'.syn$ $E'.inh = T.node$
$E' \rightarrow +TE_1'$	$E_1'.inh = new\ Node(" + ", E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \rightarrow -TE_1'$	$E_1'.inh = new\ Node(" - ", E'.inh, T.node)$ $E'.syn = E_1'.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$T \rightarrow (E)$	$T.node = E.node$
$T \rightarrow \mathbf{id}$	$T.node = \mathbf{new}\ Leaf(\mathbf{id}, \mathbf{id}.entry)$
$T \rightarrow \mathbf{num}$	$T.node = \mathbf{new}\ Leaf(\mathbf{num}, \mathbf{num}.val)$

Dependency Graph for $a - 4 + c$



Implementing SDTs

- Any SDT can be implemented by
  1. building a parse tree
  2. performing the actions in a left-to-right depth-first order, i.e., preorder traversal

- SDTs are often implemented during parsing, possibly without a parse tree, provided
  - Underlying grammar is LR and the SDD is S-attributed, or
  - Underlying grammar is LL and the SDD is L-attributed

Design of Translation Schemes

- Make all attribute values available when the semantic action is executed
- When semantic action involves only synthesized attributes, the action can be put at the end of the production

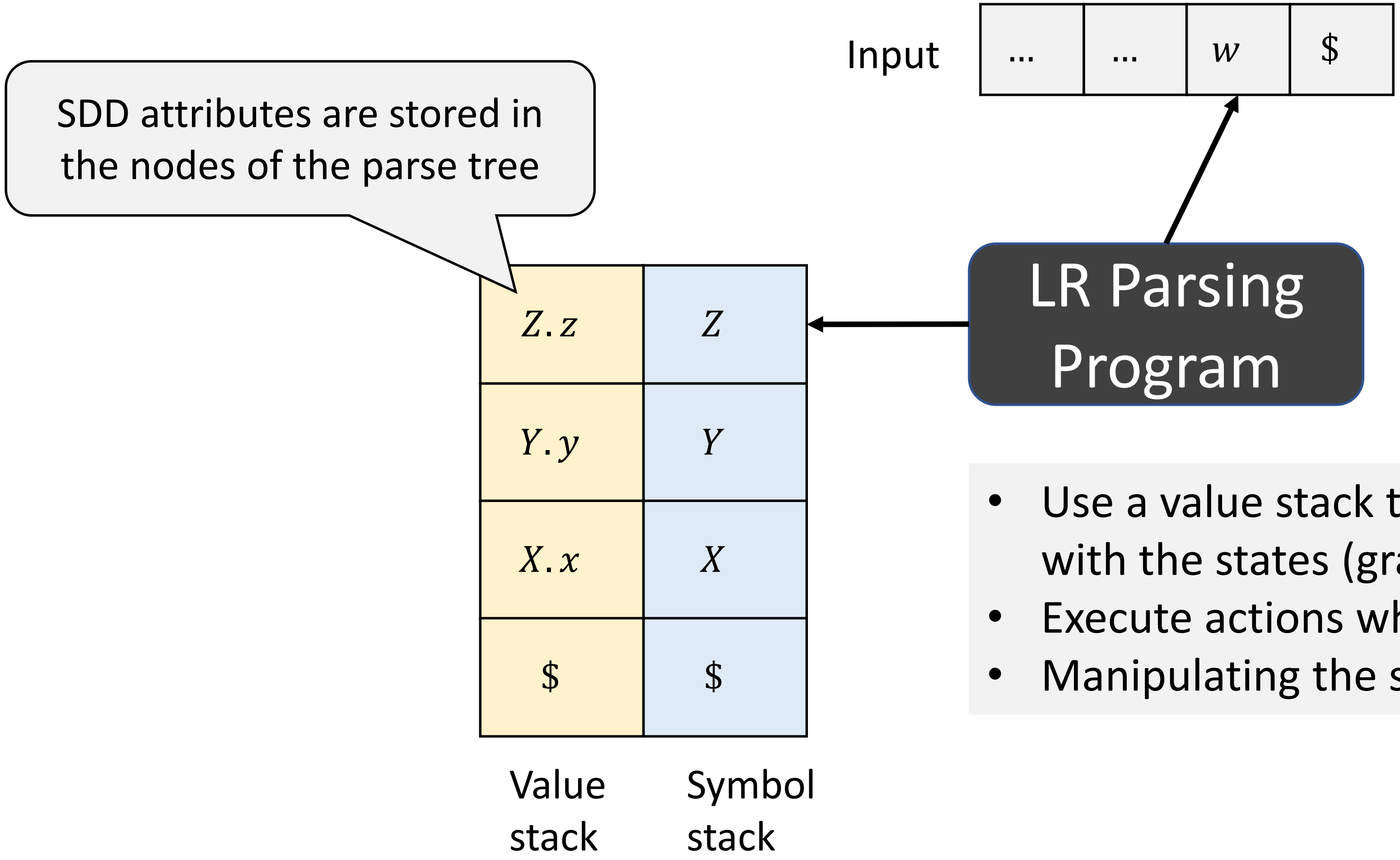
Postfix SDT for the Desk Calculator

- Consider S-attributed SDD for a bottom-up grammar
  - We can construct an SDT with actions at the end of each production
- SDT with all actions at the right-end of a production is called postfix SDT

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

action is executed when the body is reduced to the head of the production

Implementing Postfix SDTs During LR Parsing



- Use a value stack to maintain attributes along with the states (grammar symbols)
- Execute actions when reductions take place
- Manipulating the stack is done by the LR parser

Implementing Postfix SDTs with Bottom-up Parsing

Production	Semantic Action
$L \rightarrow E\$$	$\{\, print(stack[top-1].val); top = top - 1 \,\}$
$E \rightarrow E_1 + T$	$\{\, stack[top-2].val = stack[top-2].val + stack[top].val; top = top - 2; \}$
$E \rightarrow T$	
$T \rightarrow T_1 * F$	$\{\, stack[top-2].val = stack[top-2].val \times stack[top].val; top = top - 2; \}$
$T \rightarrow F$	
$F \rightarrow (E)$	$\{\, stack[top-2].val = stack[top-1].val; top = top - 2; \}$
$F \rightarrow \textbf{digit}$	

Yacc uses \$\$, \$1, \$2, … to refer to the semantic values in the current production

SDT with Actions Inside Productions

$$B \rightarrow X \{ a \} Y$$

- For bottom-up parsing, execute action a as soon as X occurs on top of the stack
- For top-down parsing, execute action a just before expanding nonterminal Y or checking for terminal Y in the input

Example of an SDT Problematic for Parsing

$L \rightarrow E\ \$$

$E \rightarrow \{\, print("+");\ \}\ \ E_1 + T$

$E \rightarrow T$

$T \rightarrow \{\, print("*");\}\ \ T_1 * F$

$T \rightarrow F$

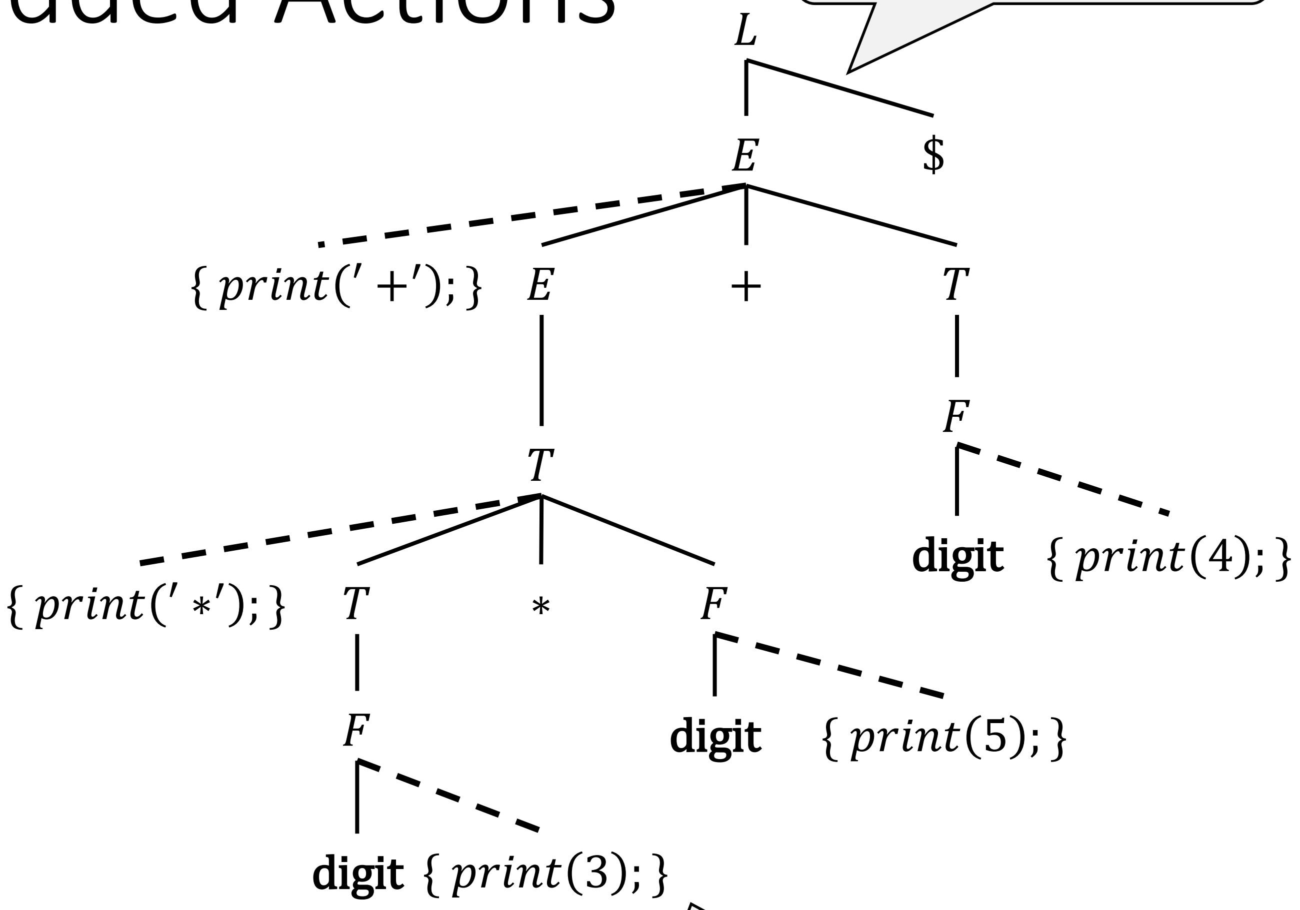
$F \rightarrow (E)$

$F \rightarrow \mathbf{digit}\ \ \{\, print(\mathbf{digit}.lexval);\ \}$

Needs to print even before seeing what is there next on the input

Parse Tree with Embedded Actions

- Parse the input and produce a parse tree
- Examine each interior node N for production $A \rightarrow \alpha$
  - Add additional children to N for the actions in α, in left-to-right order
- Perform a **preorder traversal** of the tree and execute the action as a node labeled by an action is visited



Parse tree for 3*5+4

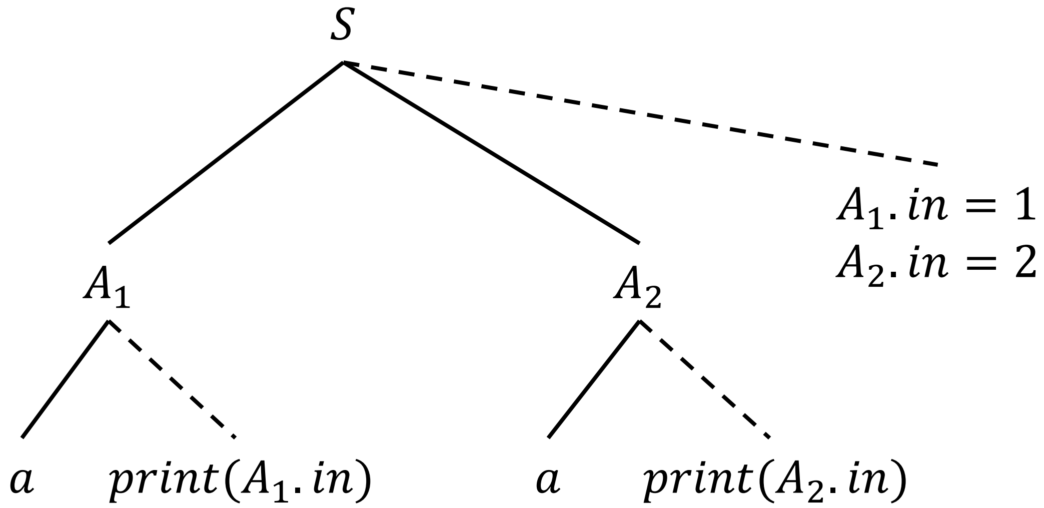
Traversing the tree in preorder generates the prefix +*354

Design Rules for L-attributed SDDs

- An inherited attribute for a symbol in the body of a production must be computed in an action **before** the symbol
- A synthesized attribute for the nonterminal on the LHS can only be computed when all the attributes it references have been computed
  - The action is usually put at the end of the production

$S \rightarrow A_1 A_2 \{ A_1.in = 1, A_2.in = 2 \}$

$A \rightarrow a \{ print(A.in) \}$



What will happen on a DFS?

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools, 2nd edition, 2.3, 5.1-5.4.
- K. Cooper and L. Torczon. Engineering a Compiler, 2nd edition, 4.1, 4.3, 4.4.
- M. Scott. Programming Language Pragmatics, 4th edition, Chapter 4.