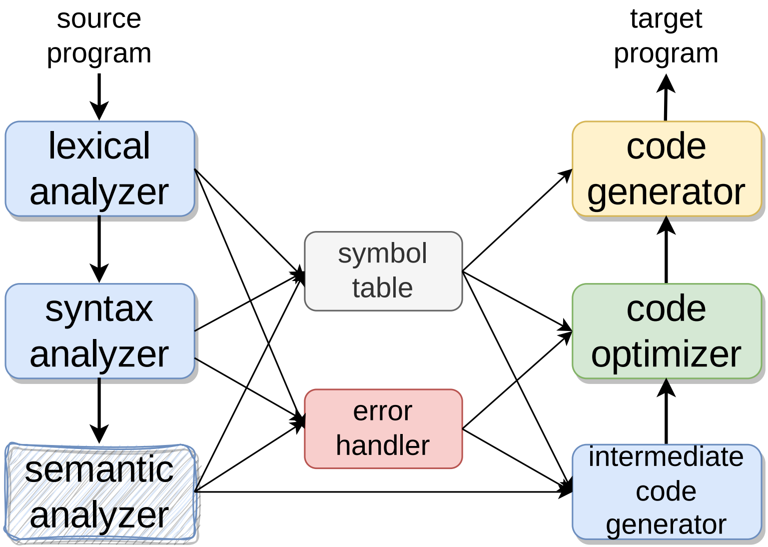
CS 335: Semantic Analysis

Swarnendu Biswas

Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Sem 2023-24-II

An Overview of Compilation



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 = 0, i;
  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 grammars (CFGs)

- 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

Ensures variable declarations go before their uses
$ProcBody \longrightarrow Decls\ Executables$

- 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 cannot 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

Additional Checks a Compiler Needs to Perform

$x \leftarrow y + z$

$x \leftarrow a + b$

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

Additional Checks a Compiler Needs to Perform

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 its declaration?
- Track inheritance relationship
- Ensure that classes and their methods are not multiply-defined

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
- **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

How Does a Compiler Check Semantics?

- 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
  - 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
- Implementation choices
  - 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
- Additional information can be used not only for semantic validation but also during subsequent phases of compilation

Attribute Grammar Framework

Syntax-Directed Definition

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
$E \longrightarrow E_1 + T$	$E.code = E_1.code \| \| T.code \| \| \text{“}+\text{”}$

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) based on 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

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

Build an attribute grammar that annotates a number with the value it represents.

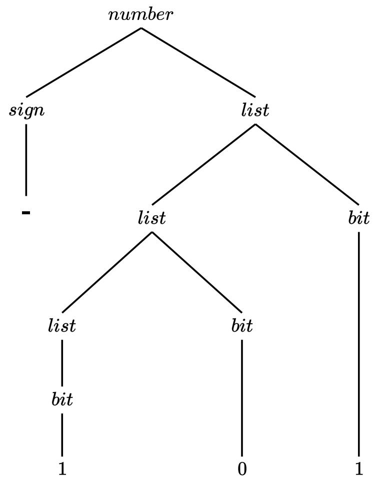
Associate attributes with grammar symbols

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** $sign.neg$: $\quad number.val = -list.val$ **else**: $\quad number.val = list.val$
$sign \rightarrow +$	$sign.neg = \text{false}$
$sign \rightarrow$	$sign.neg = \text{true}$
$list \rightarrow bit$	$bit.pos = list.pos$ $list.val = bit.val$
$list \rightarrow list_1\ bit$	$list_1.pos = list.pos + 1$ $bit.pos = list.pos$ $list.val = list_1.val + bit.val$
$bit \rightarrow 0$	$bit.val = 0$
$bit \rightarrow 1$	$bit.val = 2^{bit.pos}$

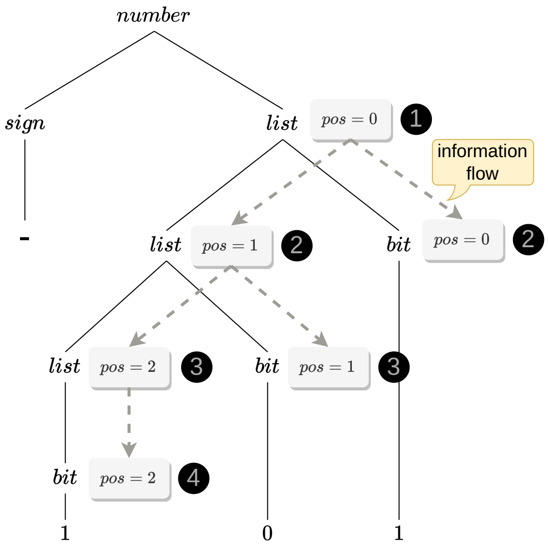
Parse Tree for -101



Annotated Parse Tree for -101

Definition

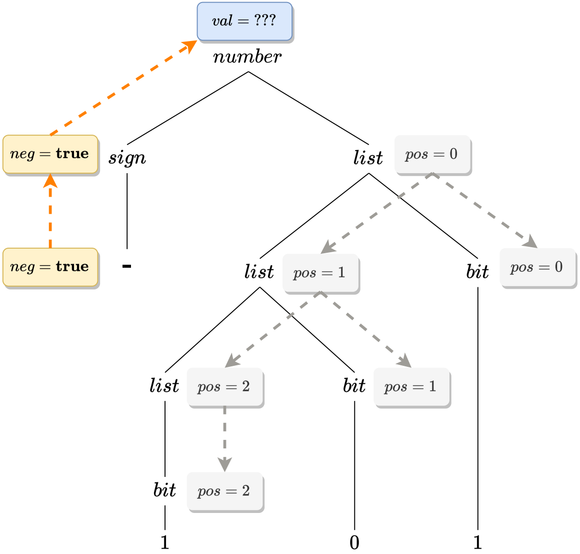
A parse tree showing the value(s) of its attribute(s) is called an **annotated** parse tree



Annotated Parse Tree for -101

Definition

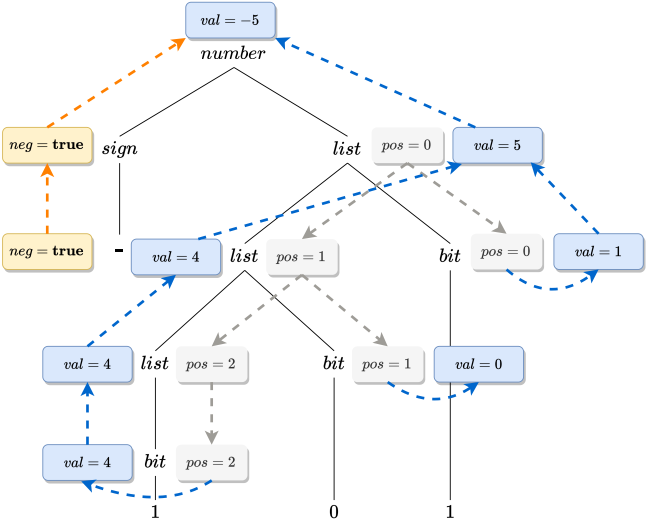
A parse tree showing the value(s) of its attribute(s) is called an **annotated** parse tree



Annotated Parse Tree for -101

Definition

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., `code` 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

Understanding Synthesized and Inherited Attributes

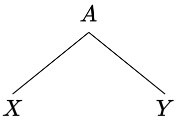
- Suppose a grammar production $A \rightarrow \alpha$ has an associated semantic rule $b = f(c_1, c_2, \ldots c_k)$
  (i) If b is a synthesized attribute of A, then $c_1, c_2, \ldots c_k$ are attributes of symbols in the production
  (ii) If b is an inherited attribute of a symbol in the body, then $c_1,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

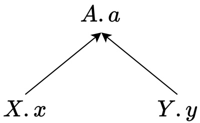
- If an attribute b depends on an attribute c, then the semantic rule for b must be evaluated after the semantic rule for c
- The dependencies among the nodes are depicted by a directed graph called the dependency graph
- Annotated parse tree **shows** the values at attributes, while the dependency graph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

Dependency Graph

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

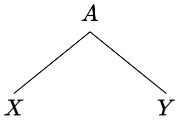


Parse tree

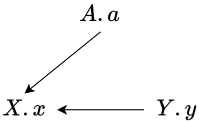


Dependency graph

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



Parse tree



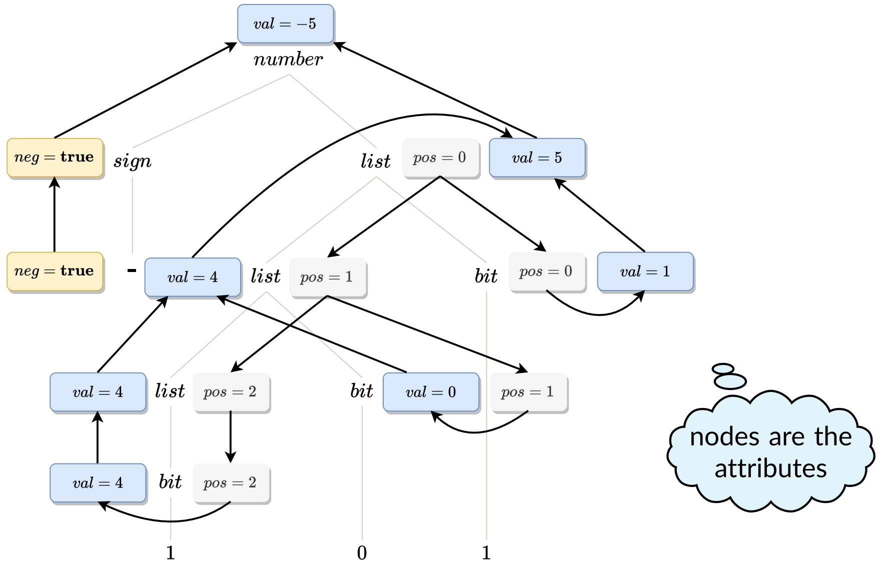
Dependency graph

Construct Dependency Graph

```
for each node n in the parse tree do
  for each attribute a of the grammar symbol n do
    construct a node in the dependency graph for a

for each node n in the parse tree do
  for each semantic rule b = f(c_1, c_2, ..., c_k) do
    // Rule is associated with production at node n
    for i = 1 to k do
      construct an edge from c_i to b
```

Example of a Dependency Graph



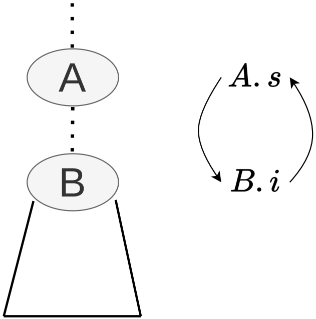
Evaluating an SDD

In what order do we evaluate attributes in an implementation?

- We must evaluate all the attributes upon which the attribute of a node depends
- SDDs do not specify any order of evaluation
- For SDDs with both synthesized and inherited attributes, there is no guarantee of an order of evaluation existing

A compiler must deal with circularity appropriately for attribute grammars

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



Evaluating an SDD

Parse tree method

- In the absence of cycles, topologically sort 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 the order of evaluation is predetermined
- E.g., evaluate *list.pos* first and *list.val* later in slide 14

Oblivious method

- Evaluation order ignores the semantic rules, and makes repeated left-to-right and right-to-left passes until all attributes have values

Types of SDDs

- Cycles should 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 by definition

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 an 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

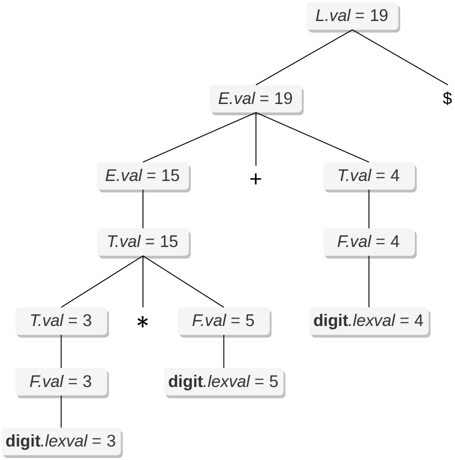
```
postorder(N) {
  for (each child C of N, from left to right)
    postorder(C)
  evaluate the attributes associated with node N
}
```

Example of S-Attributed Definition

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

all attributes are synthesized

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



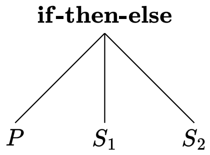
Abstract Syntax Tree (AST)

Definition

An AST is a condensed form of a parse tree that is used for representing language constructs

- Each leaf is an operand and non-leaf nodes represent operators
- ASTs represent relationships between language constructs, do not bother with derivations

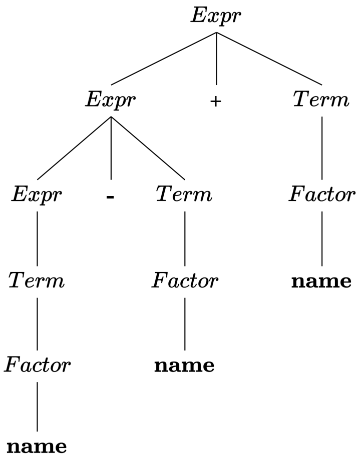
$$S \rightarrow \textbf{if } P \textbf{ then } S_1 \textbf{ else } S_2$$



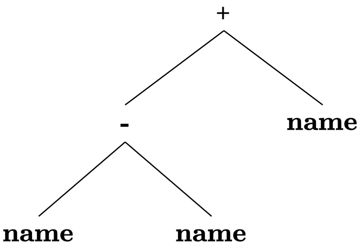
Parse trees are also called concrete syntax trees

Parse Tree vs Abstract Syntax Tree

Parse Tree



Abstract Syntax Tree



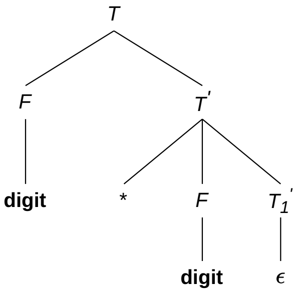
Inherited Attributes

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

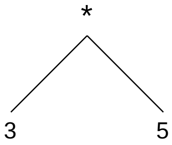
Production	Semantic Rules
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

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

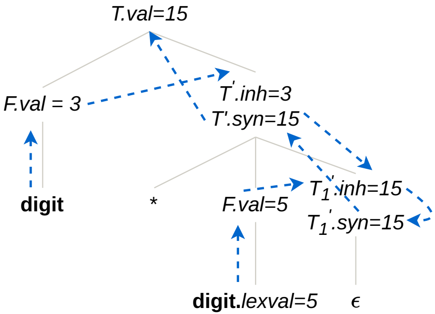
Parse Tree



AST



Annotated Parse Tree

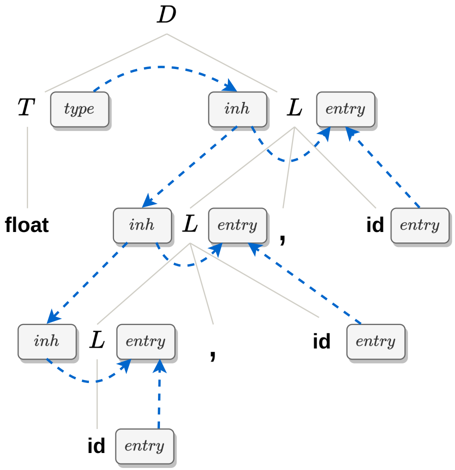


Example SDD with Side Effects

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

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

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



Notes about Inherited Attributes

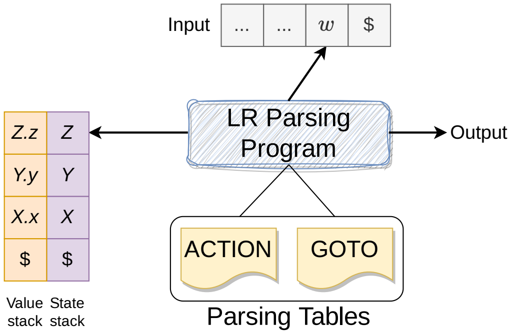
- Always possible to rewrite an SDD to use only synthesized attributes
  - Inherited attributes can be simulated with synthesized attributes and helper functions
- May be more logical to use both synthesized and inherited attributes
- Inherited attributes usually cannot be evaluated by a simple preorder 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 the semantic rule is $A = f(X.x, Y.y, Z.z)$
- 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
$L \longrightarrow E\$$	$L.val = E.val$
$E \longrightarrow E_1 + T$	$E.val = E_1.val + T.val$
$E \longrightarrow T$	$E.val = T.val$
$T \longrightarrow T_1 * F$	$T.val = T_1.val \times F.val$
$T \longrightarrow F$	$T.val = F.val$
$F \longrightarrow (E)$	$F.val = E.val$
$F \longrightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Bottom-up Evaluation of S-Attributed Definition

Value	Stack	Input	Action
\$	\$	$3 * 5 + 4\$$	Shift 3
\$3	\$**digit**	$* 5 + 4\$$	Reduce by $F \rightarrow$ **digit**
\$3	$\$F$	$* 5 + 4\$$	Reduce by $T \rightarrow F$
\$3	$\$T$	$* 5 + 4\$$	Shift *
\$3	$\$T *$	$5 + 4\$$	Shift *
\$3 5	$\$T *$ **digit**	$+ 4\$$	Reduce by $F \rightarrow$ **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 4
\$15 4	$\$E +$ **digit**	\$	Reduce by $F \rightarrow$ **digit**
\$15 4	$\$E + F$	\$	Reduce by $T \rightarrow F$
\$15 4	$\$E + T$	\$	Reduce by $E \rightarrow E + T$
\$19	$\$E$	\$	Accept

L-Attributed Definitions

- Each attribute must be either
  - (i) Synthesized, or
  - (ii) Suppose $A \rightarrow X_1X_2\ldots X_n$ 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.

- Dependences flow from left-to-right among inherited attributes

Production	Semantic Rule
$T \rightarrow FT'$	$T'.inh = F.val$ $T.val = T'.syn$
$T' \rightarrow *FT'_1$	$T'_1.inh = T'.inh \times F.val$ $T'.syn = T'_1.syn$
$T' \rightarrow \epsilon$	$T'.syn = T'.inh$
$F \rightarrow \textbf{digit}$	$F.val = \textbf{digit}.lexval$

Are these SDDs S- or L-attributed?

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

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

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

S-Attributed and L-Attributed Definitions

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

All L-attributed grammars are not S-attributed

Summarizing Attribute Grammars

- Attribute grammars define a set of attributes and translations at every node of the parse tree, the 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

Challenges with Attribute Grammars

- 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
- Results can be **scattered** across attributes in the parse tree
  - Moving important attributes to the root node introduces additional copy instructions
- Works in conjunction with a parse tree or an AST, but a compiler implementation may **not build** either

Syntax-Directed Translation

Syntax-Directed Translation (SDT)

- **Program fragments** are embedded as semantic actions in the production body of a CFG
  - Generates code while parsing
- Indicates the **order** in which semantic actions are to be evaluated

$$rest \rightarrow +term \; \{ \texttt{print}(\text{'}+\text{'}) \} \; rest_1$$

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

SDD for Infix to Postfix Translation

- Postfix notation for an expression E is defined inductively
  - If E is a variable or constant, then postfix notation is E
  - If $E = E_1 \text{ op } E_2$ 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 postfix notation for E

SDD for Infix to Postfix Translation

- Postfix notation for an expression E is defined inductively
  - If E is a variable or constant, then postfix notation is E
  - If $E = E_1$ 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 postfix notation for E

Production	**Semantic Rules**
$expr \longrightarrow expr_1 + term$	$expr.code = expr_1.code \| \| term.code \| \| \text{“} + \text{”}$
$expr \longrightarrow expr_1 - term$	$expr.code = expr_1.code \| \| term.code \| \| \text{“} - \text{”}$
$expr \longrightarrow term$	$expr.code = term.code$
$term \longrightarrow 0 \| 1 \| \ldots \| 9$	$term.code = \text{“0”}$ $term.code = \text{“1”}$ $\ldots$ $term.code = \text{“9”}$

SDT for Infix to Postfix Translation

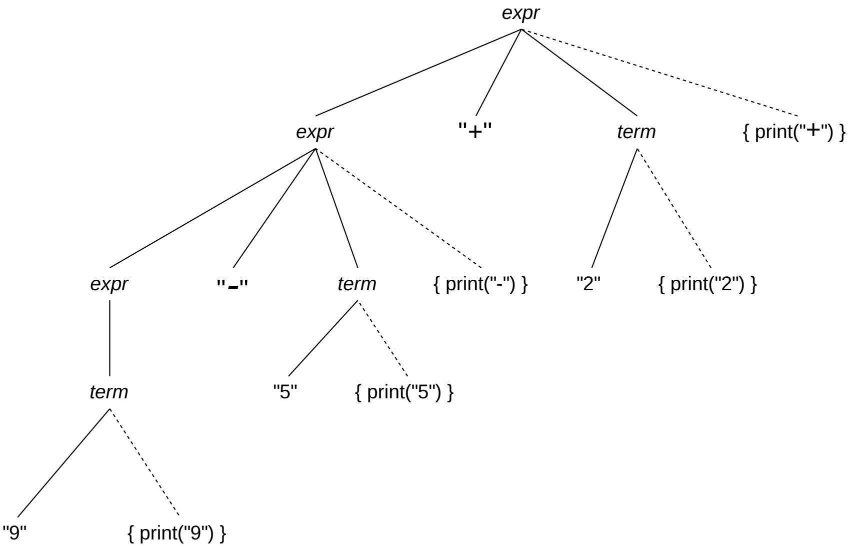
SDD

Production	Semantic Rules
$expr \rightarrow expr_1 + term$	$expr.code = expr_1.code \Vert term.code \Vert ``+"$
$expr \rightarrow expr_1 - term$	$expr.code = expr_1.code \Vert term.code \Vert ``-"$
$expr \rightarrow term$	$expr.code = term.code$
$term \rightarrow 0 \vert 1 \vert \ldots \vert 9$	$term.code = ``0"$ $term.code = ``1"$ $\ldots$ $term.code = ``9"$

SDT

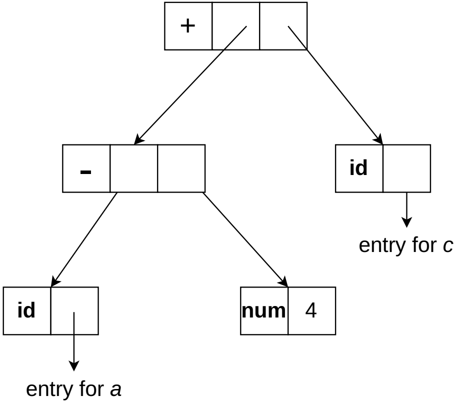
Production	Semantic Actions
$expr \rightarrow expr_1 + term$	$\{\texttt{print}(``+")\}$
$expr \rightarrow expr_1 - term$	$\{\texttt{print}(``-")\}$
$expr \rightarrow term$	
$term \rightarrow 0 \vert 1 \vert \ldots \vert 9$	$\{\texttt{print}(``0")\}$ $\{\texttt{print}(``1")\}$ $\ldots$ $\{\texttt{print}(``9")\}$

SDT Actions to Translate 9 – 5 + 2 to Postfix



Construction of AST for Expressions

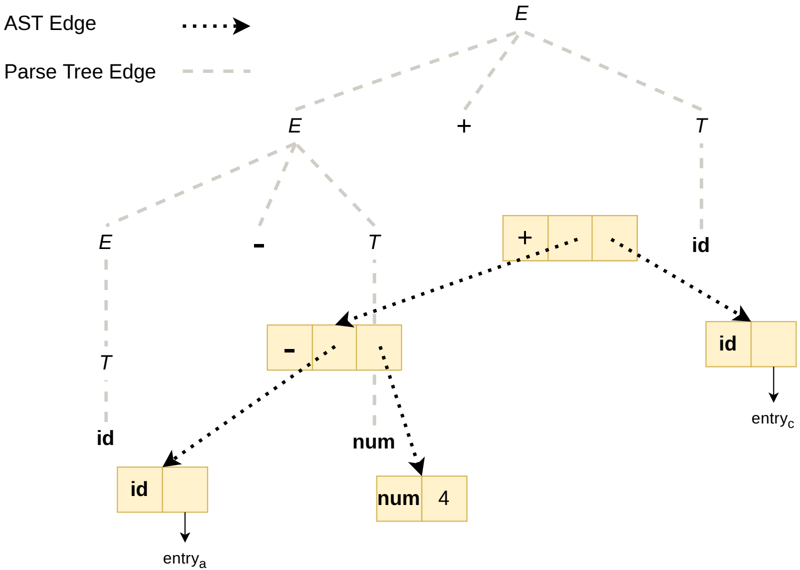
- **Idea**: Construct subtrees for subexpressions by creating an operator and operand nodes
- Helper functions
  - (i) Create an internal node with label *op* and k fields denoting k children with $Node(op, c_1, c_2, \ldots, c_k)$
  - (ii) Create a leaf node with label *op* and *val* as the lexical value with $Leaf(op, val)$
- The following sequence of function calls creates an AST for $a - 4 + c$
  1. $p_1 = \textbf{new } Leaf(\textbf{id}, entry_a)$
  2. $p_2 = \textbf{new } Leaf(\textbf{num}, 4)$
  3. $p_3 = \textbf{new } Node(\text{“}-\text{”}, p_1, p_2)$
  4. $p_4 = \textbf{new } Leaf(\textbf{id}, entry_c)$
  5. $p_5 = \textbf{new } Node(\text{“}+\text{”}, p_3, p_4)$



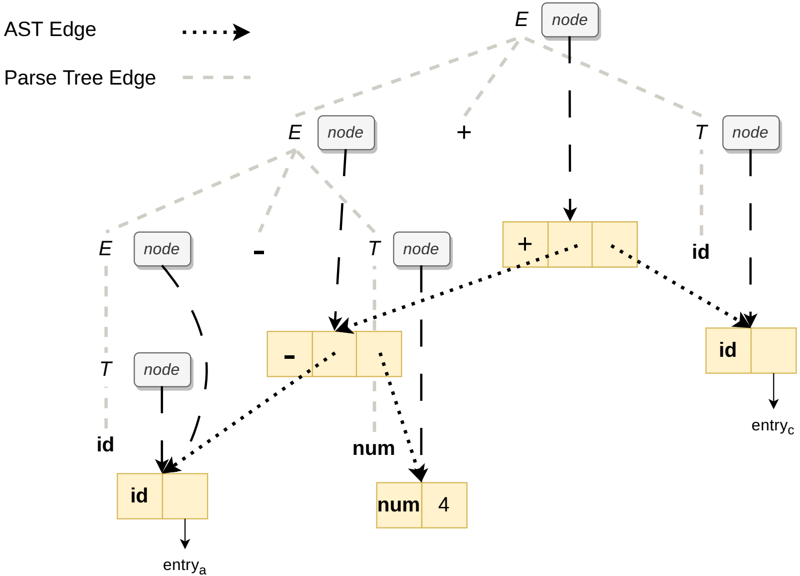
S-Attributed Definition for Constructing ASTs

Production	Semantic Actions
$E \rightarrow E_1 + T$	$E.node = \textbf{new}\ Node(\text{``}+\text{''}, E_1.node, T.node)$
$E \rightarrow E_1 - T$	$E.node = \textbf{new}\ Node(\text{``}-\text{''}, E_1.node, T.node)$
$E \rightarrow T$	$E.node = T.node$
$T \rightarrow (E)$	$T.node = E.node$
$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, entry_{\textbf{id}})$
$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Construction of AST for $a - 4 + c$



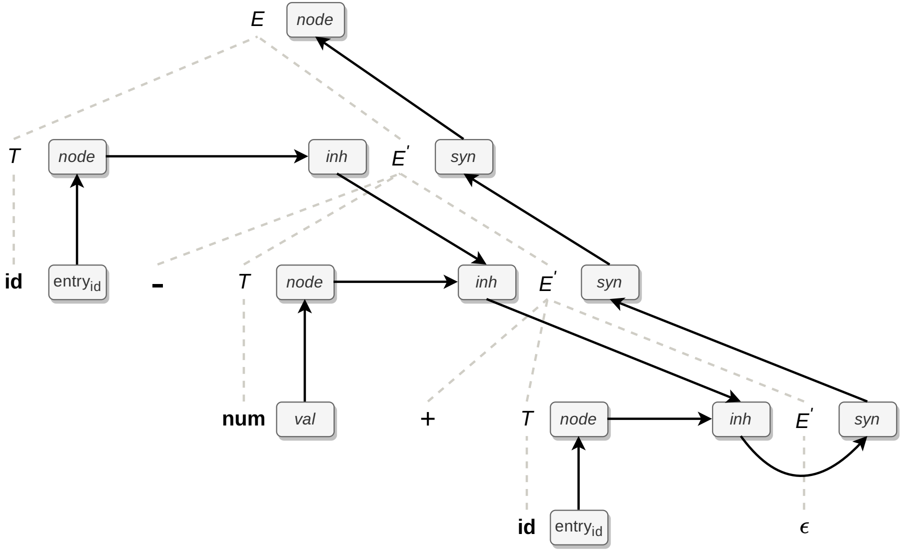
Construction of AST for $a - 4 + c$



L-Attributed Definition for Constructing Syntax Trees

Production	Semantic Actions
$E \rightarrow TE'$	$E.node = E'.syn$ $E'.inh = T.node$
$E' \rightarrow +TE'_1$	$E'_1.inh = \textbf{new}\ Node(\text{"}+\text{"}, E'.inh, T.node)$ $E'.syn = E'_1.syn$
$E' \rightarrow -TE'_1$	$E'_1.inh = \textbf{new}\ Node(\text{"}-\text{"}, E'.inh, T.node)$ $E'.syn = E'_1.syn$
$E' \rightarrow \epsilon$	$E'.syn = E'.inh$
$T \rightarrow (E)$	$T.node = E.node$
$T \rightarrow \textbf{id}$	$T.node = \textbf{new}\ Leaf(\textbf{id}, entry_{\textbf{id}})$
$T \rightarrow \textbf{num}$	$T.node = \textbf{new}\ Leaf(\textbf{num}, \textbf{num}.val)$

Dependency Graph for $a - 4 + c$



Implementing SDTs

- Any SDT can be implemented by
  (i) building a parse tree, and
  (ii) performing the actions in a left-to-right depth-first order (i.e., preorder traversal)
- Need to make all attribute values available when the semantic action is executed
- 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
- When semantic action involves **only synthesized** attributes, the action can be put at the **end** of the production
- SDT with all **actions at the right end** of a production is called **postfix SDT**

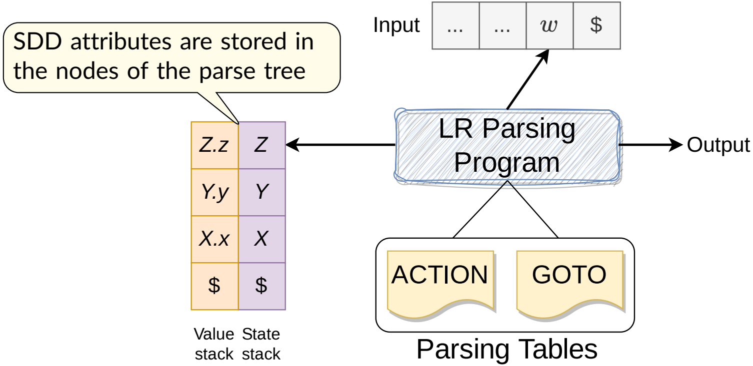
Postfix SDT for the Desk Calculator

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

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

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

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

SDT with Actions Inside Productions

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

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

Infix-to-Prefix SDT Problematic for Translating During Parsing

$L \rightarrow E\$$
$E \rightarrow \{\texttt{print}(\text{“}+\text{”})\} E_1 + T$
$E \rightarrow T$
$T \rightarrow \{\texttt{print}(\text{“}*\text{”})\} T_1 * F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow \textbf{digit}\{\texttt{print}(\textbf{digit}.\mathit{lexval})\}$

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

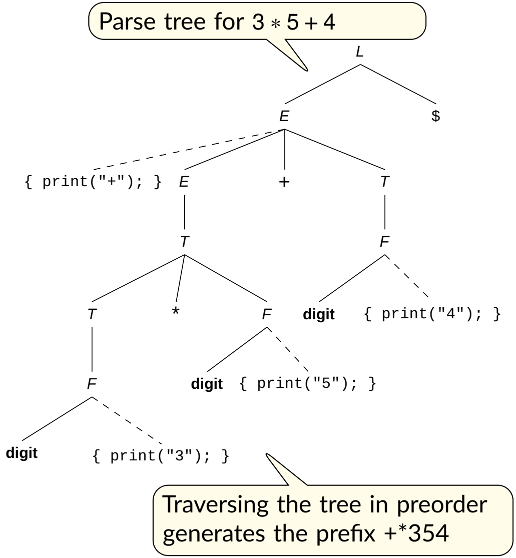
Implementing SDTs with Embedded Actions

(i) Parse the input and produce a parse tree (ignore semantic actions)

(ii) 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

(iii) Perform a **preorder traversal** of the tree and execute the action when a node labeled by an action is visited



Converting L-attributed SDDs to SDTs

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

$$S \rightarrow \textbf{while}\ (C)\ S_1$$

Symbol	Attributes
S	*next*, *code*
C	*true*, *false*, *code*

$S.code$ and $C.code$ are synthesized attributes, while $S.next$, $C.true$, and $C.false$ are inherited attributes

Example of Converting L-attributed SDDs to SDTs

SDD

Production	Semantic Rules
$S \rightarrow \textbf{while}\ (C)\ S_1$	$L1 = \textbf{new}\ \texttt{Label}(); L2 = \textbf{new}\ \texttt{Label}();$ $C.false = S.next; C.true = L2;$ $S_1.next = L1;$ $S.code = \textbf{label} \| L1 \| C.code \| \textbf{label} \| L2 \| S_1.code$

SDT

$S \rightarrow \textbf{while}\ ($	$\{L1 = \textbf{new}\ \texttt{Label}(); L2 = \textbf{new}\ \texttt{Label}();$ $C.false = S.next; C.true = L2; \}$
$C)$	$\{S_1.next = L1; \}$
S_1	$\{S.code = \textbf{label} \| L1 \| C.code \| \textbf{label} \| L2 \| S_1.code; \}$

References

- A. Aho et al. Compilers: Principles, Techniques, and Tools. Sections 2.3, 5.1–5.4, 2nd edition, Pearson Education.
- K. Cooper and L. Torczon. Engineering a Compiler. Sections 4.1, 4.3–4.4, 2nd edition, Morgan Kaufmann.