Type Systems – Part 2, and Abstract Syntax Trees

Lecture 12

Outline of today's lecture

- Type systems continued

- Abstract-syntax tree (AST)

  - All subsequent phases of a compiler operate on ASTs

  - How is it different from a parse tree?

Last class

- Introduction to semantic analysis
- Dynamic and static scope
- Symbol Table
- Introduction to types and typing rules

Why a Separate Semantic Analysis?

- Parsing alone cannot catch many errors
- Some language constructs are not context-free
- Separation of concerns: less-complicated parsers

What Does Semantic Analysis Do?

- Checks of many kinds . . . Checks:

    1. All identifiers are declared
    2. Types
    3. Inheritance relationships
    4. Classes defined only once
    5. Methods in a class defined only once
    6. Reserved identifiers are not misused

    And others . . .

What's Wrong?

- Example 1

  Let y: String ← "abc" in y + 3

- Example 2

  Let y: Int in x + 3

Note: An example property that is not context free.

Types

- What is a type?
  - The notion varies from language to language

- Consensus
  - A set of values
  - A set of operations on those values

- Classes are one instantiation of the modern notion of type

Why Do We Need Type Systems?

Consider the assembly language fragment

add $r1, $r2, $r3

What are the types of $r1, $r2, $r3?

Types and Operations

- Certain operations are legal for values of each type
  - It doesn't make sense to add a function pointer and an integer in C++
  - It does make sense to add two integers
  - But both have the same assembly language implementation!

Type Systems

- A language's type system specifies which operations are valid for which types

- The goal of type checking is to ensure that operations are used with the correct types
  - Enforces intended interpretation of values, because nothing else will!

Type Checking Overview

- Three kinds of languages:
  - *Statically typed*: All or almost all checking of types is done as part of compilation (C, Java, Cool)
  - *Dynamically typed*: Almost all checking of types is done as part of program execution (Scheme)
  - *Untyped*: No type checking (machine code)

Type Checking and Type Inference

- *Type Checking* is the process of verifying fully typed programs

- *Type Inference* is the process of filling in missing type information

- The two are different, but the terms are often used interchangeably

Rules of Inference

- We have seen two examples of formal notation specifying parts of a compiler
  - Regular expressions
  - Context-free grammars

- The appropriate formalism for type checking is logical rules of inference

Why Rules of Inference?

- Inference rules have the form

  If Hypothesis is true, then Conclusion is true

- Type checking computes via reasoning

  If E_1 and E_2 have certain types, then E_3 has a certain type

- Rules of inference are a compact notation for "If-Then" statements

From English to an Inference Rule

- The notation is easy to read with practice

- Start with a simplified system and gradually add features

- Building blocks
  - Symbol $\wedge$ is "and"
  - Symbol $\Rightarrow$ is "if-then"
  - $x:T$ is "x has type T"

From English to an Inference Rule (2)

If e_1 has type Int and e_2 has type Int,
then $e_1 + e_2$ has type Int

$(e_1$ has type Int $\wedge$ e_2 has type Int$)$ $\Rightarrow$
$e_1 + e_2$ has type Int

$(e_1\text{: Int} \wedge e_2\text{: Int}) \Rightarrow e_1 + e_2\text{: Int}$

From English to an Inference Rule (3)

The statement

$$(e_1: \text{Int} \wedge e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}$$

is a special case of

$$\text{Hypothesis}_1 \wedge \ldots \wedge \text{Hypothesis}_n \Rightarrow \text{Conclusion}$$

This is an inference rule.

Notation for Inference Rules

- By tradition inference rules are written

$$\frac{\text{` Hypothesis … ` Hypothesis}}{\text{` Conclusion}}$$

- Type rules have hypotheses and conclusions

` e:T

- ` means “it is provable that . . .”

Two Rules

$$\frac{i \text{ is an integer literal}}{\text{`}\ i : \text{Int}} \quad [\text{Int}]$$

$$\frac{\text{`}\ e_1 : \text{Int} \qquad \text{`}\ e_2 : \text{Int}}{\text{`}\ e_1 + e_2 : \text{Int}} \quad [\text{Add}]$$

Two Rules (Cont.)

- These rules give templates describing how to type integers and + expressions

- By filling in the templates, we can produce complete typings for expressions

Example: 1 + 2

$$\frac{\dfrac{1 \text{ is an int literal}}{\vdash 1 : \text{Int}} \qquad \dfrac{2 \text{ is an int literal}}{\vdash 2 : \text{Int}}}{\vdash 1 + 2 : \text{Int}}$$

Soundness

- A type system is *sound* if
  - Whenever ` e: T
  - Then e evaluates to a value of type T

- We only want sound rules

Abstract Syntax Trees

- So far a parser traces the derivation of a sequence of tokens

- The rest of the compiler needs a structural representation of the program

- <u>Abstract syntax trees</u>
  - Similar to parse trees but ignore some details
  - Abbreviated as AST

Abstract Syntax Tree. (Cont.)

- Consider the grammar

  $E \rightarrow \text{int} \mid (E) \mid E + E$

- And the string

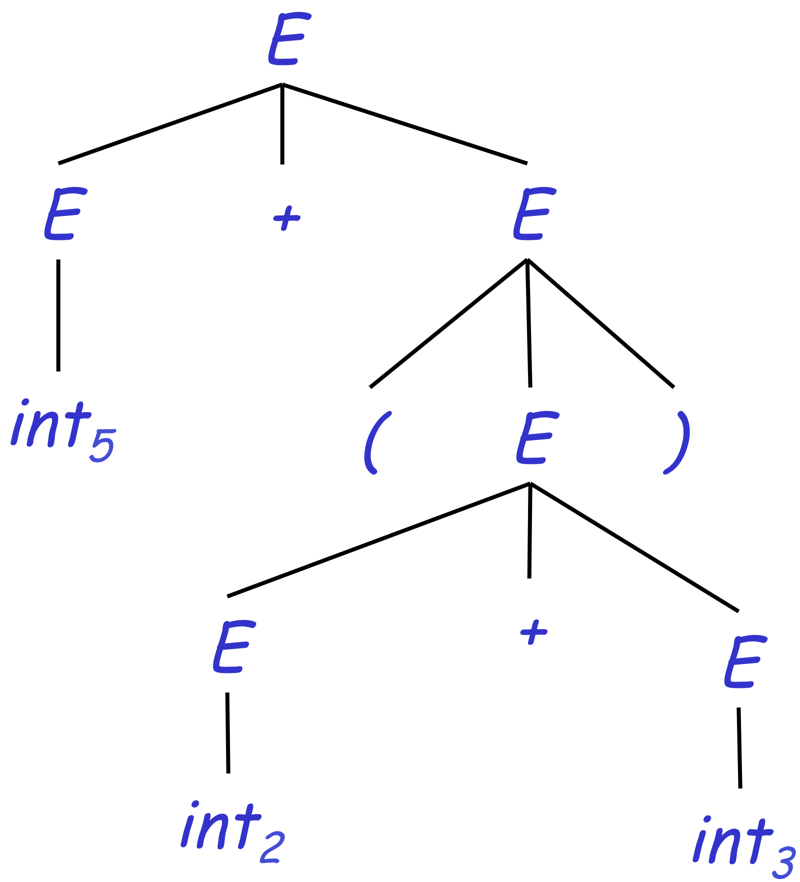
  $5 + (2 + 3)$

- After lexical analysis (a list of tokens)

  int_5 '+' '(' int_2 '+' int_3 ')'

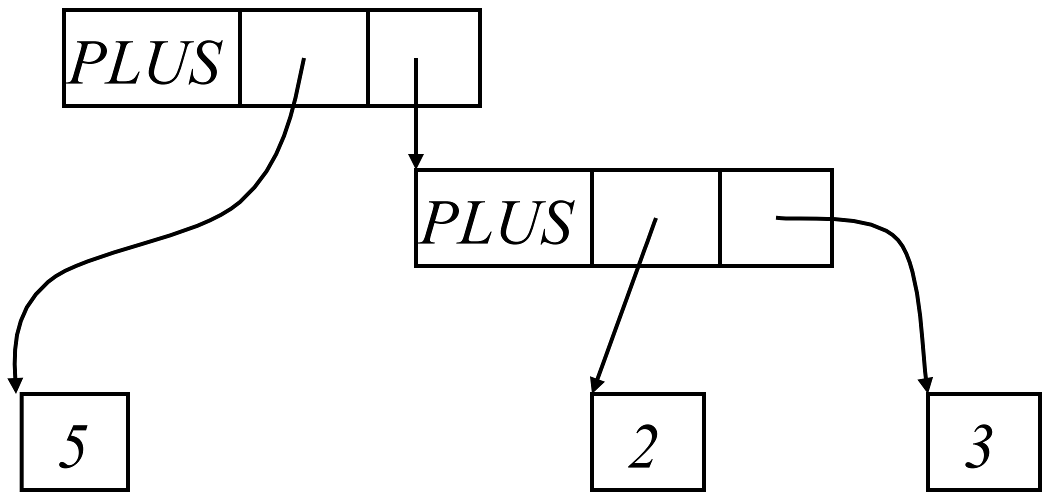
- During parsing we build a parse tree …

Example of Parse Tree



- Traces the operation of the parser
- Does capture the nesting structure
- But too much info
  - Parentheses
  - Single-successor nodes

Example of Abstract Syntax Tree



- Also captures the nesting structure
- But <u>abstracts</u> from the concrete syntax
  => more compact and easier to use
- An important data structure in a compiler

Semantic Actions

- This is what we'll use to construct ASTs
- Each grammar symbol may have <u>attributes</u>
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer
- Each production may have an <u>action</u>
  - Written as: $X \to Y_1 \ldots\ Y_n \quad \{\text{ action }\}$
  - That can refer to or compute symbol attributes

Semantic Actions: An Example

- Consider the grammar

$$E \rightarrow \text{int} \mid E + E \mid (E)$$

- For each symbol X define an attribute X.val
  - For terminals, val is the associated lexeme
  - For non-terminals, val is the expression's value (and is computed from values of subexpressions)

- We annotate the grammar with actions:

$$
\begin{array}{ll}
E \rightarrow \text{int} & \{ E.val = \text{int}.val \} \\
\quad \mid E_1 + E_2 & \{ E.val = E_1.val + E_2.val \} \\
\quad \mid (E_1) & \{ E.val = E_1.val \}
\end{array}
$$

Semantic Actions: An Example (Cont.)

- *String:* $5 + (2 + 3)$
- *Tokens:* int_5 '+' '(' int_2 '+' int_3 ')'

Productions	Equations
$E \rightarrow E_1 + E_2$	$E.val = E_1.val + E_2.val$
$E_1 \rightarrow int_5$	$E_1.val = int_5.val = 5$
$E_2 \rightarrow (E_3)$	$E_2.val = E_3.val$
$E_3 \rightarrow E_4 + E_5$	$E_3.val = E_4.val + E_5.val$
$E_4 \rightarrow int_2$	$E_4.val = int_2.val = 2$
$E_5 \rightarrow int_3$	$E_5.val = int_3.val = 3$

Semantic Actions: Notes

- Semantic actions specify a system of equations
  - Order of resolution is not specified

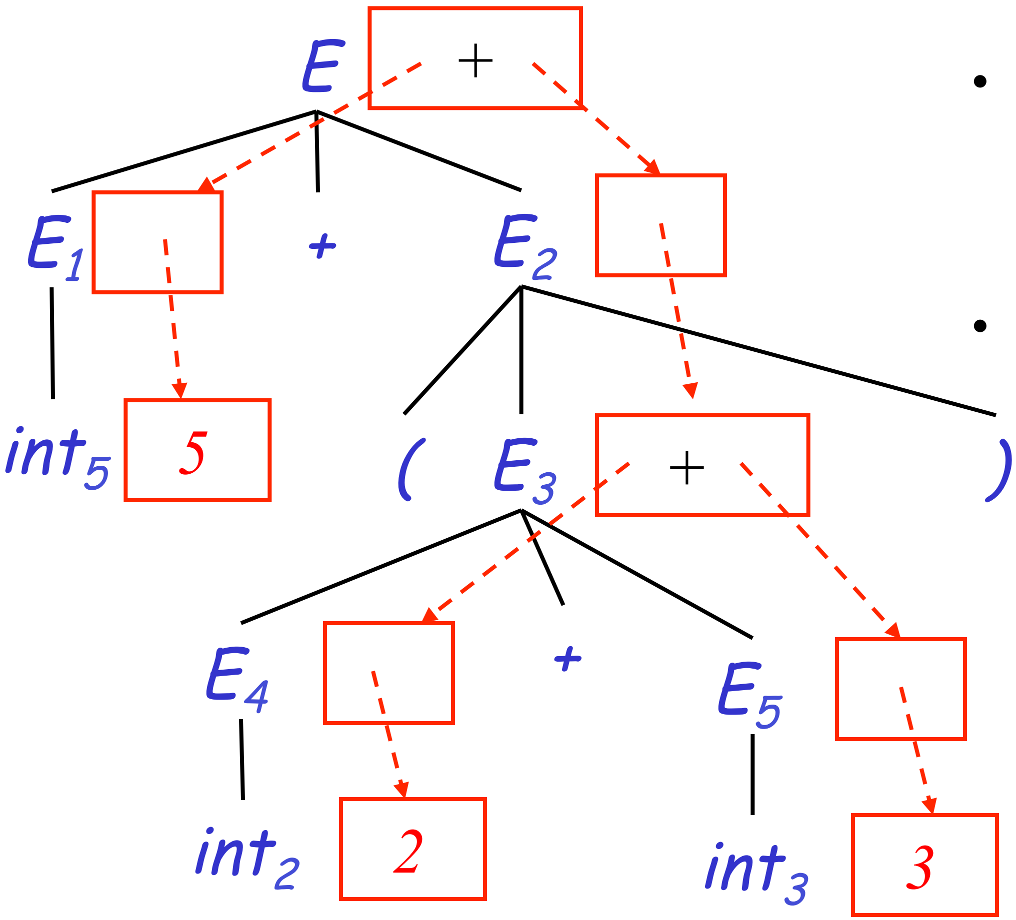
- Example:

  $E_3.val = E_4.val + E_5.val$

  - Must compute $E_4.val$ and $E_5.val$ before $E_3.val$
  - We say that $E_3.val$ depends on $E_4.val$ and $E_5.val$

- The parser must find the order of evaluation

Dependency Graph



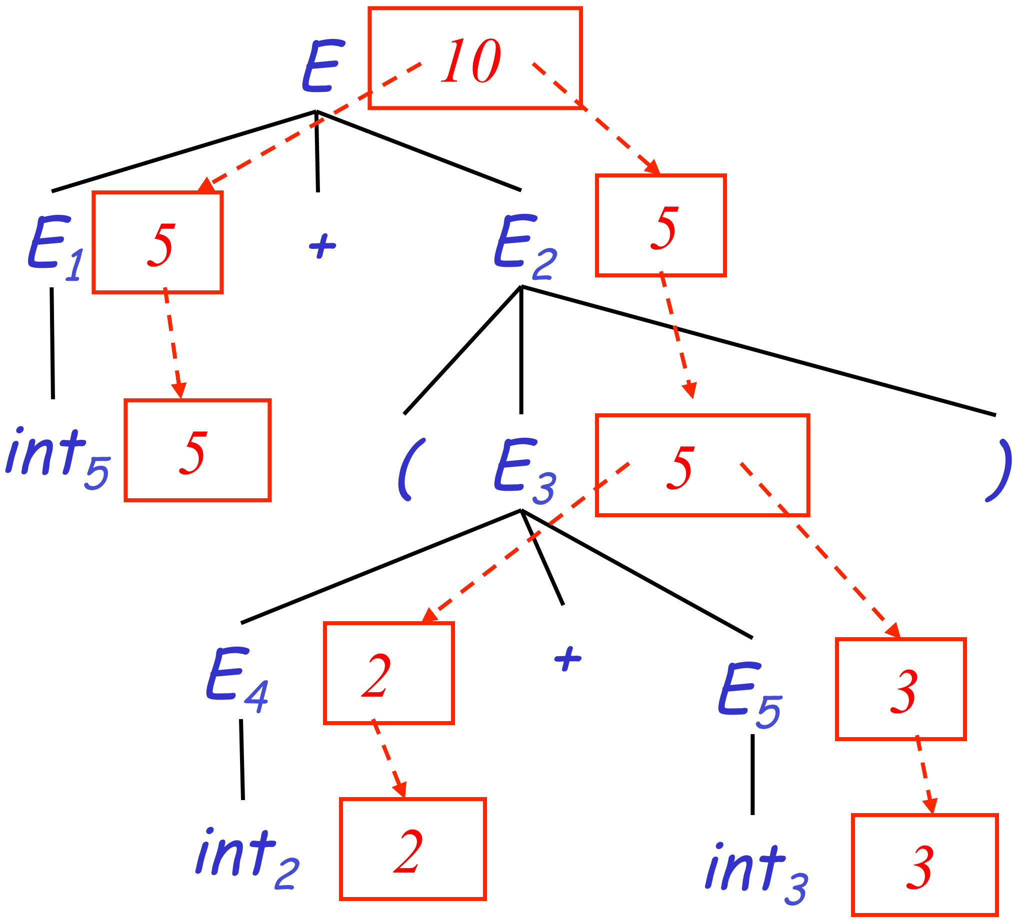
- Each node labeled E has one slot for the val attribute
- Note the dependencies

Evaluating Attributes

- An attribute must be computed after all its successors in the dependency graph have been computed
  - In previous example attributes can be computed bottom-up

- Such an order exists when there are no cycles
  - Cyclically defined attributes are not legal

Dependency Graph



Semantic Actions: Notes (Cont.)

- Synthesized attributes
  - Calculated from attributes of descendents in the parse tree
  - E.val is a synthesized attribute
  - Can always be calculated in a bottom-up order

- Grammars with only synthesized attributes are called S-attributed grammars
  - Most common case

Semantic Actions: Notes (Cont.)

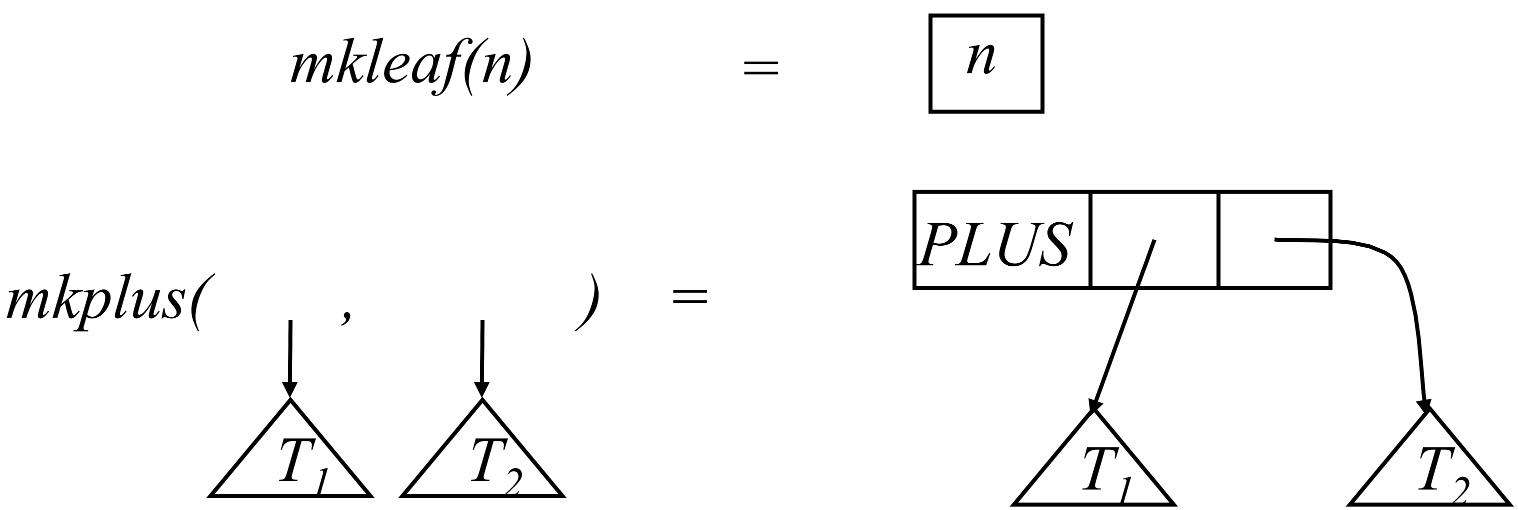
- Semantic actions can be used to build ASTs

- And many other things as well
  - Also used for type checking, code generation, …

- Process is called <u>syntax-directed translation</u>
  - Substantial generalization over CFGs

Constructing An AST

- We first define the AST data type
- Consider an abstract tree type with two constructors:



Constructing a Parse Tree

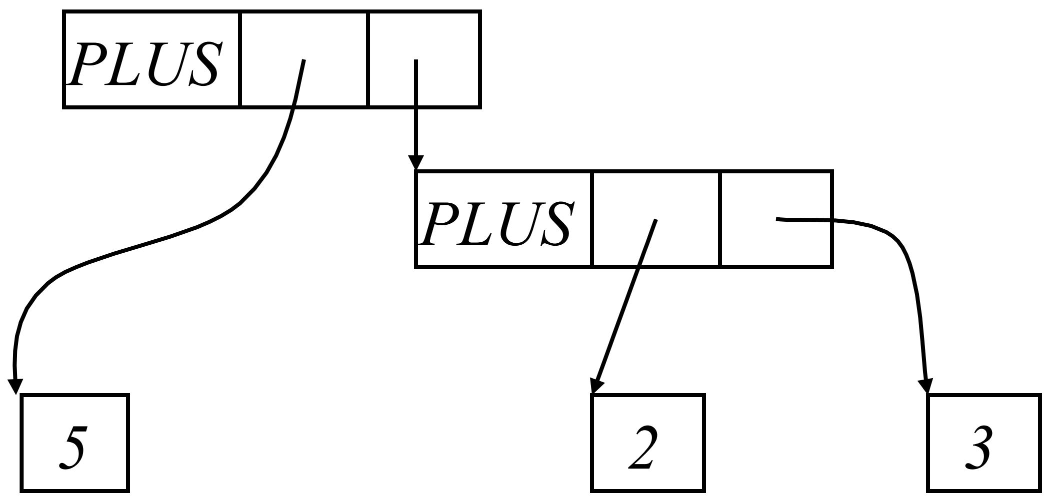
- We define a synthesized attribute ast
  - Values of ast values are ASTs
  - We assume that int.lexval is the value of the integer lexeme
  - Computed using semantic actions

$E \rightarrow$ int	E.ast = mkleaf(int.lexval)
$\mid E_1 + E_2$	E.ast = mkplus(E_1.ast, E_2.ast)
$\mid (E_1)$	E.ast = E_1.ast

Parse Tree Example

- Consider the string int_5 '+' '(' int_2 '+' int_3 ')'
- A bottom-up evaluation of the ast attribute:

  E.ast = mkplus(mkleaf(5),
  mkplus(mkleaf(2), mkleaf(3))



Summary

- We can specify language syntax using CFG

- A parser will answer whether $s \in L(G)$
  - … and will build a parse tree
  - … which we convert to an AST
  - … and pass on to the rest of the compiler