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 $\land$ 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 $\textbf{Int}$ and $e_2$ has type $\textbf{Int}$, then $e_1 + e_2$ has type $\textbf{Int}$

$(e_1 \text{ has type } \textbf{Int} \land e_2 \text{ has type } \textbf{Int}) \Rightarrow e_1 + e_2 \text{ has type } \textbf{Int}$

$(e_1 : \textbf{Int} \land e_2 : \textbf{Int}) \Rightarrow e_1 + e_2 : \textbf{Int}$
From English to an Inference Rule (3)

The statement

\[(e_1: \text{Int} \land e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}\]

is a special case of

\[\text{Hypothesis}_1 \land \ldots \land \text{Hypothesis}_n \Rightarrow \text{Conclusion}\]

This is an inference rule.
Notation for Inference Rules

- By tradition inference rules are written
  ```
  Hypothesis ... Hypothesis
  Conclusion
  ```
- Type rules have hypotheses and conclusions
  ```
  e : T
  ```
- `` means “it is provable that . . .”
Two Rules

\( \text{i is an integer literal} \quad \text{\`i : Int} \quad \text{[Int]} \)

\( \text{\`e_1 : Int} \quad \text{\`e_2 : Int} \quad \text{\`e_1 + e_2 : Int} \quad \text{[Add]} \)

Original Slides by Prof. Alex Aiken
(Modified by Prof. Vijay Ganesh) Lecture 12
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

1 is an int literal
\`
1 : Int
\`

2 is an int literal
\`
2 : Int
\`

\`
1 + 2 : 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

- **Abstract syntax trees**
  - 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)
  \[ \text{int}_5 \ '+' \ '(\ ' \text{int}_2 \ '+' \ \text{int}_3 \ ')' \]

- 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 abstracts 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 attributes
  - For terminal symbols (lexical tokens) attributes can be calculated by the lexer

- Each production may have an action
  - Written as: \( X \rightarrow Y_1 \ldots Y_n \) \{ 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, \( \text{val} \) is the associated lexeme
  - For non-terminals, \( \text{val} \) is the expression’s value (and is computed from values of subexpressions)

• We annotate the grammar with actions:
  \[
  E \rightarrow \text{int} \quad \{ \text{E.val} = \text{int.val} \} \\
  \mid E_1 + E_2 \quad \{ \text{E.val} = E_1.val + E_2.val \} \\
  \mid ( E_1 ) \quad \{ \text{E.val} = E_1.val \}
  \]
Semantic Actions: An Example (Cont.)

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

<table>
<thead>
<tr>
<th>Productions</th>
<th>Equations</th>
</tr>
</thead>
<tbody>
<tr>
<td>E → E₁ + E₂</td>
<td>E.val = E₁.val + E₂.val</td>
</tr>
<tr>
<td>E₁ → int₅</td>
<td>E₁.val = int₅.val = 5</td>
</tr>
<tr>
<td>E₂ → ( E₃ )</td>
<td>E₂.val = E₃.val</td>
</tr>
<tr>
<td>E₃ → E₄ + E₅</td>
<td>E₃.val = E₄.val + E₅.val</td>
</tr>
<tr>
<td>E₄ → int₂</td>
<td>E₄.val = int₂.val = 2</td>
</tr>
<tr>
<td>E₅ → int₃</td>
<td>E₅.val = int₃.val = 3</td>
</tr>
</tbody>
</table>
Semantic Actions: Notes

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

• Example:
  \[ E_3.\text{val} = E_4.\text{val} + E_5.\text{val} \]
  - Must compute \( E_4.\text{val} \) and \( E_5.\text{val} \) before \( E_3.\text{val} \)
  - We say that \( E_3.\text{val} \) depends on \( E_4.\text{val} \) and \( E_5.\text{val} \)

• 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

\[
\begin{align*}
E & \leftarrow 10 \\
E_1 & + E_2 \\
( E_3 & + E_4 + E_5 ) \\
E_4 & + E_5 \\
int_5 & 5 \\
int_2 & 2 \\
int_3 & 3
\end{align*}
\]
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 syntax-directed translation
  – Substantial generalization over CFGs
Constructing An AST

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

\[
mkleaf(n) = \begin{cases} \n \end{cases}
\]

\[
mkplus(T_1, T_2) = \begin{cases} \text{PLUS} \end{cases}
\]
Constructing a Parse Tree

• We define a synthesized attribute \( \text{ast} \)
  - Values of \( \text{ast} \) values are ASTs
  - We assume that \( \text{int.lexval} \) is the value of the integer lexeme
  - Computed using semantic actions

\[
E \rightarrow \text{int} \quad \text{E.ast} = \text{mkleaf} \left( \text{int.lexval} \right) \\
| E_1 + E_2 \quad \text{E.ast} = \text{mkplus} \left( E_1.\text{ast}, E_2.\text{ast} \right) \\
| ( E_1 ) \quad \text{E.ast} = E_1.\text{ast}
\]
Parse Tree Example

- Consider the string $\text{int}_5 ' + ' ( ' \text{int}_2 ' + ' \text{int}_3 ' )$
- A bottom-up evaluation of the ast attribute:
  $$E.\text{ast} = \text{mkplus}(\text{mkleaf}(5),$$
  $$\text{mkplus}(\text{mkleaf}(2), \text{mkleaf}(3)))$$
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