Abstract Syntax Tree (AST)

• Modularization of functionality avoids overcomplicating compilation phases
  – Semantic Analysis
  – Symbol table construction
  – Program optimization
  – Code generation

• AST is central data structure for all post-parsing activities (listed above)

• Goal of parsing is creation of AST
Syntax Trees

• Concrete
  – Actual parse tree

• Abstract
  – Eliminates unnecessary nodes
  – Structures the tree appropriately for evaluation
  – Serves as basis for code generation
Concrete vs. Abstract

Figure 7.16: (a) Derivation of $a + 5$ from $E$; (b) Abstract representation of $a + 5$. 

AST

- Constructed bottom-up (assuming bottom-up parsing).
- Support tree construction from leaves towards the root.
- Efficiently support nodes with arbitrary number of children.
Data Structure Implementation

• How would you implement this structure?
  – Tree with arbitrary children
• Java
• C/C++
Figure 7.12: Internal format of an AST node. A dashed line connects a node with its parent; a dotted line connects a node with its leftmost sibling. Each node also has a solid connection to its leftmost child and right sibling.
AST Design and Construction

• AST Nodes must hold sufficient information to recall the essential elements of the program fragments they represent
  – For example, want to know that node X of the AST corresponds directly to procedure Foo of the program

• Implementation of ASTs should be decoupled from what is represented
  – Accessors are used to hide a node’s internal representation

• No single class hierarchy
  – Each phase may view the nodes differently
AST Design and Construction

- Given a source language “L”, development of grammar and design of an AST is as follows:
  1. An unambiguous grammar for L is devised
     - May contain productions that disambiguate the grammar
  2. An AST for L is devised
     - Typically discards grammar details concerned with disambiguation
     - Omits semantically useless symbols
AST Design and Construction

3. Semantic actions are placed in grammar to construct AST
   – Code attached to CUP productions used to build AST nodes

4. Passes of the compiler are designed
   – Each phase may place new requirements on AST
Sample Grammar

1  Start → Stmt $
2  Stmt → id assign E
3   | if lparen E rparen Stmt else Stmt fi
4   | if lparen E rparen Stmt fi
5   | while lparen E rparen do Stmt od
6   | begin Stmts end
7  Stmts → Stmts semi Stmt
8   | Stmt
9  E → E plus T
10  | T
11  T → id
12  | num

Figure 7.14: Grammar for a simple language.
AST Structures

(a) assign
   \[\text{variable} \quad \text{expression}\]

(b) plus
   \[\text{expression} \quad \text{expression}\]

(c) if
   \[\text{predicate} \quad \text{alternative} \quad \text{alternative}\]

(d) while
   \[\text{predicate} \quad \text{loop body}\]

(e) block
   \[\text{statement} \quad \text{statement} \quad \text{statement} \quad \text{statement}\]
Construction

• CUP allows the assignment to Java variables of values associated with nonterminals.
• CUP allows attaching Java code to productions which are run when a reduction using that production is performed.
  – Usually results in setting a value for the left-hand nonterminal.
Construction

• Base grammar

Stmt ::= id assign E

| if lparen E rparen Stmt fi

| if lparen E rparen Stmt else Stmt fi

...

;
Construction

• Variable names attached to nonterminals

Stmt ::= id:id assign E:e

  | if lparen E:pr rparen Stmt:s fi

  | if lparen E:pr rparen Stmt:s1 else Stmt:s2 fi

...
Construction

• Java code added to productions
  – Most common action is to build a new tree node and assign to RESULT, which attaches it to the left-hand nonterminal
  • Values for the nonterminals on the right-hand side are usually child tree nodes

Stmt ::= id:id assign E:e
   { : RESULT = new AssignmentNode(id, e); :}
   | if lparen E:pr rparen Stmt:s fi
   { : RESULT = new IfNode(pr, s); :}
   | if lparen E:pr rparen Stmt:s1 else Stmt:s2 fi
   { : RESULT = new IfNode(pr, s1, s2); :}
...
;
Construction

Stmt ::= begin Stmt:block end

  {: RESULT = block; :}

;

Stmts ::= Stmt:stmt

  {: Result = block; :}

| Stmt:stmt

  {: RESULT = new BlockNode(s); :}

;
Construction

• Alternate construction of BlockNode

Stmt ::= begin Stmts:list end
   {: RESULT = new BlockNode(list); :}
;
Stmts ::= Stmts:list semi Stmt:stmt
   {:
     list.add(stmt);
     RESULT = list;
   :}
   | Stmt:s
   {: RESULT = new ArrayList();
     RESULT.add(s);
   :}
;
Concrete Syntax Tree

Figure 7.18: Concrete syntax tree.

Note that there is an error in this figure. There be a “do” token in here to match the grammar shown on the earlier slide.
Abstract Syntax Tree
Left and Right Values

\[ x = y \]

- “x” is the L-value
  - Refers to the location of “x”, not its value
- “y” is the R-value
  - Refers to the value of “y”, not its location
Implementation

• Java
  – Limits L-values for safety reasons

• C/C++
  – Provides operators for conversions
    ```
    int *p, *x, y;
    *p = 0;
    x = &y;
    ```
Example

Figure 7.22: ASTs illustrating left and right values for the assignments:

(a) \( x = y \)
(b) \( x = \& y \)
(c) \( \star x = y \)

Note that there is an error in this figure. The deref in the tree for example b should not be there.