CSE431 – Translation of Computer Languages

- Abstract Syntax Trees

Doug Shook

Roadmap / Review

- So far we have seen:
  - Scanners
  - Parsers
- What are tokens used for?
  - What is a parse tree used for?

Roadmap / Review

- What is the ultimate goal of the compiler?

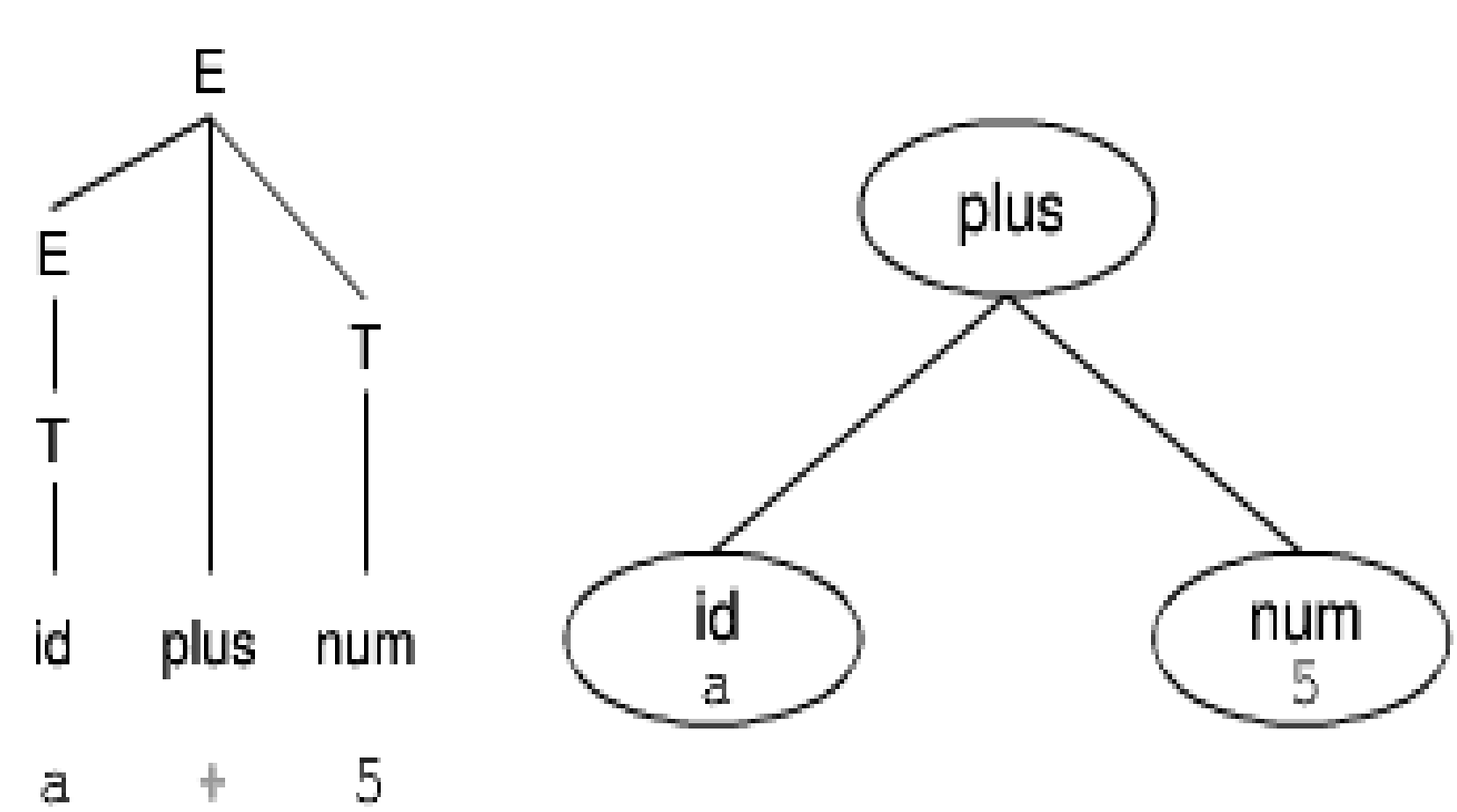
- Still several steps left:
  - Type Checking
  - Semantic Analysis
  - Code Generation
  - Optimization

- Is the parse tree the best tool for this job?
  - Why or why not?

Abstract Syntax Tree

- The parse tree is concrete
  - Is this a syntactically valid string?
- An AST is used for determining the meaning
  - "Scaffolding" removed
  - Redundant information is removed
  - Tries to capture the "essence" of the string
  - Serves as the basis for semantic analysis and code generation

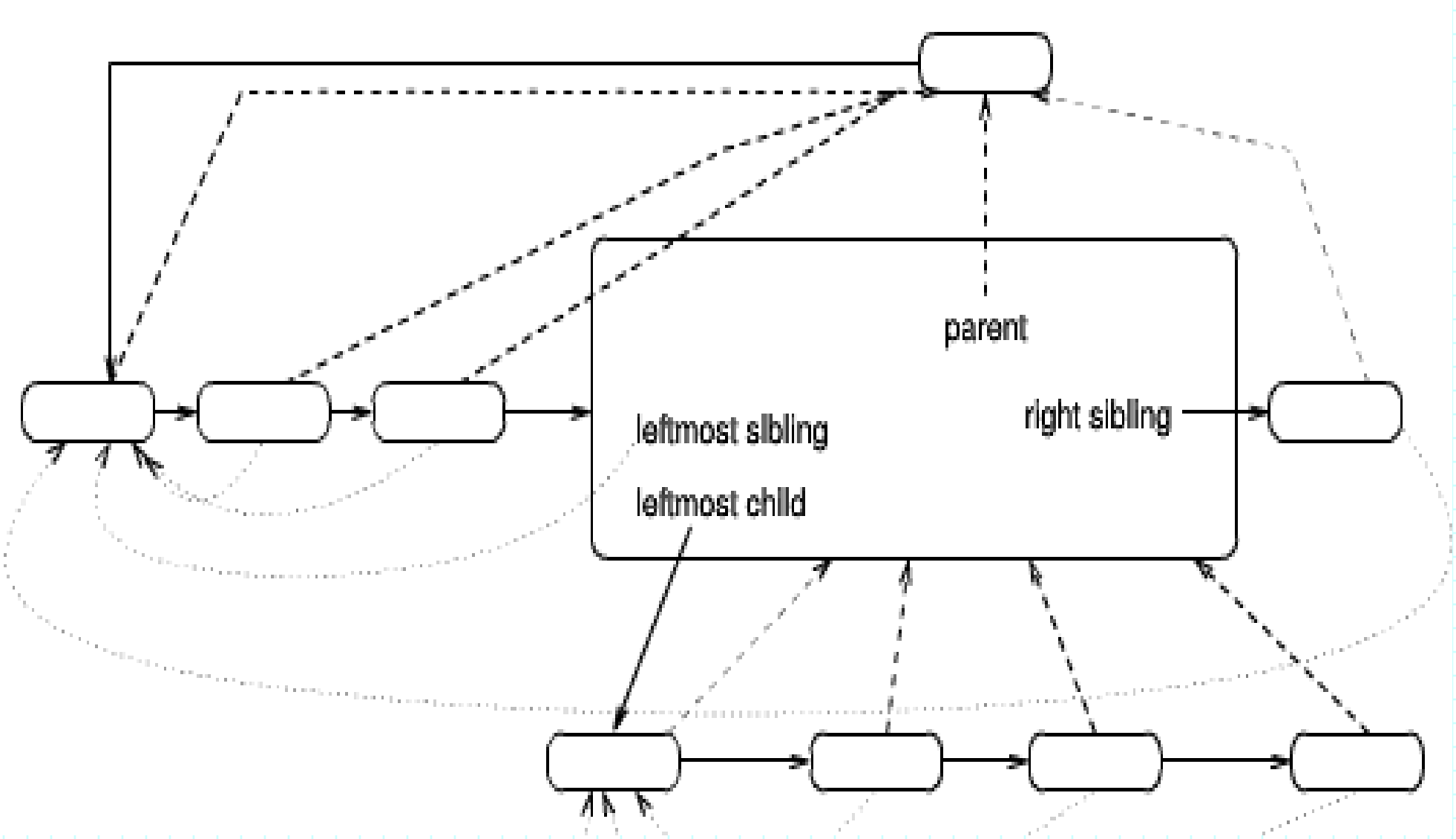
Abstract Syntax Tree



Abstract Syntax Tree

- An AST design typically follows these rules:
  - Bottom up
    - Create list of siblings, which are later adopted
  - Siblings are generated recursively
  - Support arbitrary numbers of child nodes
    - Arithmetic operations might have two nodes
    - Others might vary in size
      - I.E: method arguments

Abstract Syntax Tree



Abstract Syntax Tree

- Important properties regarding AST design:
  - It should be possible to unparse an AST
    - We want to know that node X corresponds to multiplication, etc.
  - The AST implementation is decoupled from content
    - Provides for consistent traversal
  - No single class hierarchy
    - Phase-specific interfaces exist

Abstract Syntax Tree

- Important methods:
  - `MakeNode(t)`
    - Creates an AST node that differs based on type *t*
  - `x.MakeSiblings(y)`
    - *y* becomes *x*'s sibling
  - `x.AdoptChildren(y)`
    - *x* adopts *y* and all of *y*'s siblings
  - `MakeFamily(op, kid1, kid2, ... , kidn)`
    - Convenience method
      - Create node *op*, adopt *kids*

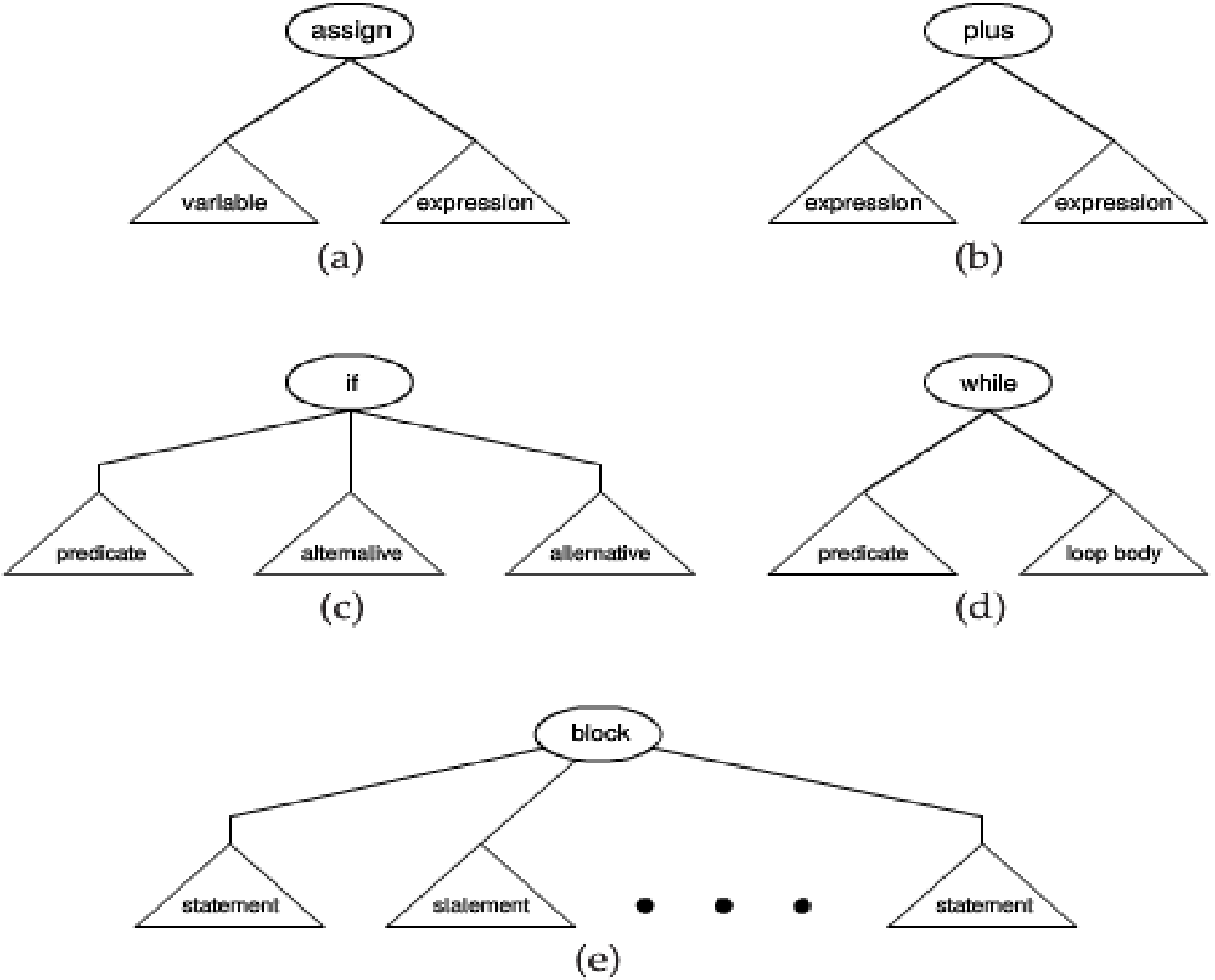
AST Construction

- Given a programming language, L
  - Design an unambiguous grammar for L
  - Design an AST for L
    - Omits details regarding disambiguation
    - Omits semantically useless symbols
  - Put semantic actions into the grammar
    - Utilize method from previous slide
  - Design compiler passes
    - 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
```

AST Structures



Semantic Actions

```
1  Start          → Stmt_ast $
                    return (ast)                                          (13)
2  Stmt_result    → id_var assign E_expr
                    result ← MAKEFAMILY(assign, var, expr)                (14)
3                 | if lparen E_p rparen Stmt_s fi
                    result ← MAKEFAMILY(if, p, s, MAKENODE( ))            (15)
4                 | if lparen E_p rparen Stmt_s1 else Stmt_s2 fi
                    result ← MAKEFAMILY(if, p, s1, s2)                    (16)
5                 | while lparen E_p rparen do Stmt_s od
                    result ← MAKEFAMILY(while, p, s)                      (17)
6                 | begin Stmts_list end
                    result ← MAKEFAMILY(block, list)                      (18)
7  Stmts_result   → Stmts_sofar semi Stmt_next
                    result ← sofar.MAKESIBLINGS(next)                     (19)
8                 | Stmt_first
                    result ← first                                        (20)
9  E_result       → E_e1 plus T_e2
                    result ← MAKEFAMILY(plus, e1, e2)                     (21)
10                | T_e
                    result ← e                                            (22)
11 T_result       → id_var
                    result ← MAKENODE(var)                                (23)
12                | num_val
                    result ← MAKENODE(val)                                (24)
```

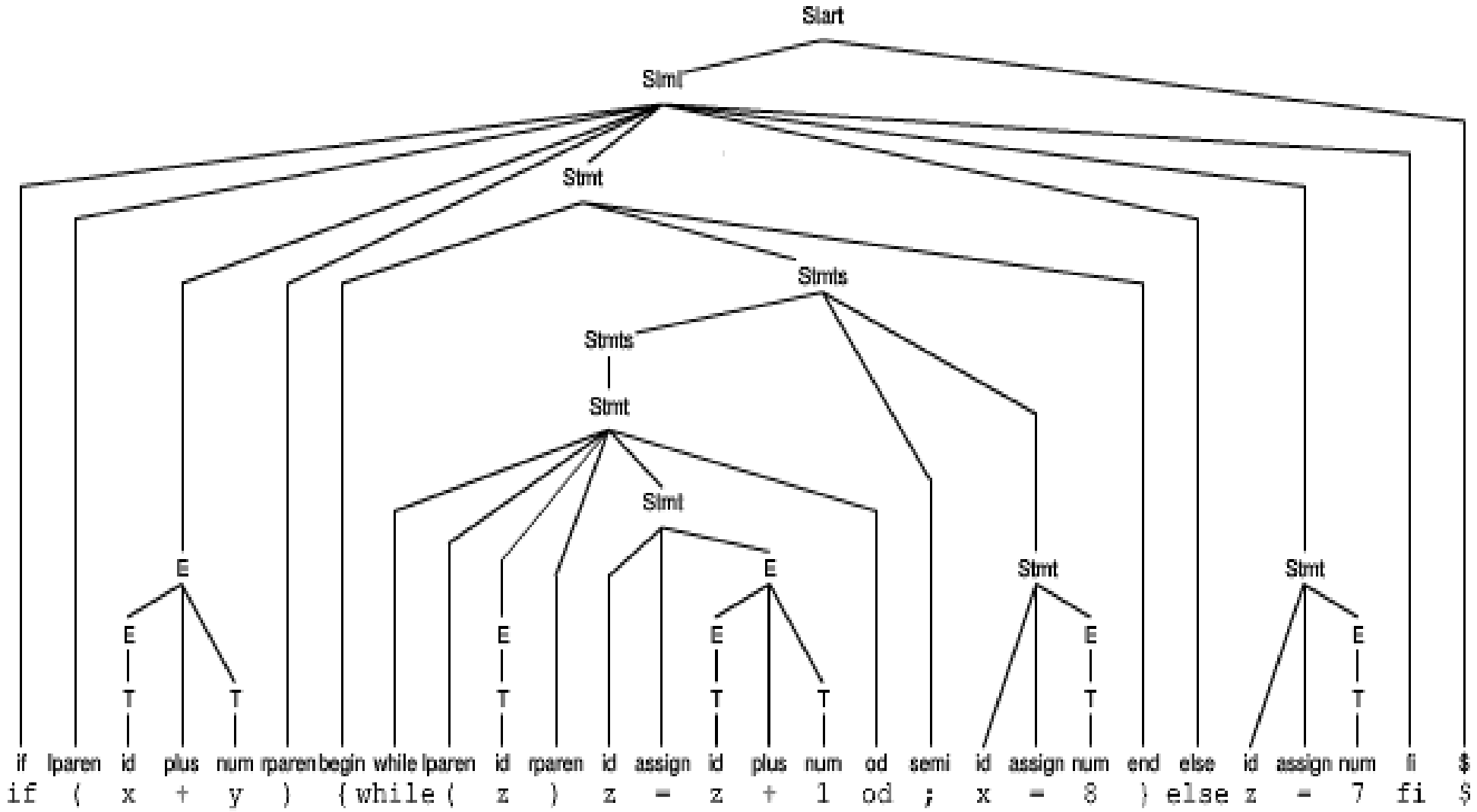
Semantic Actions - CUP

```
Stmt ::= id assign E
      |   if lparen E rparen Stmt fi
      |   if lparen E rparen Stmt else Stmt fi

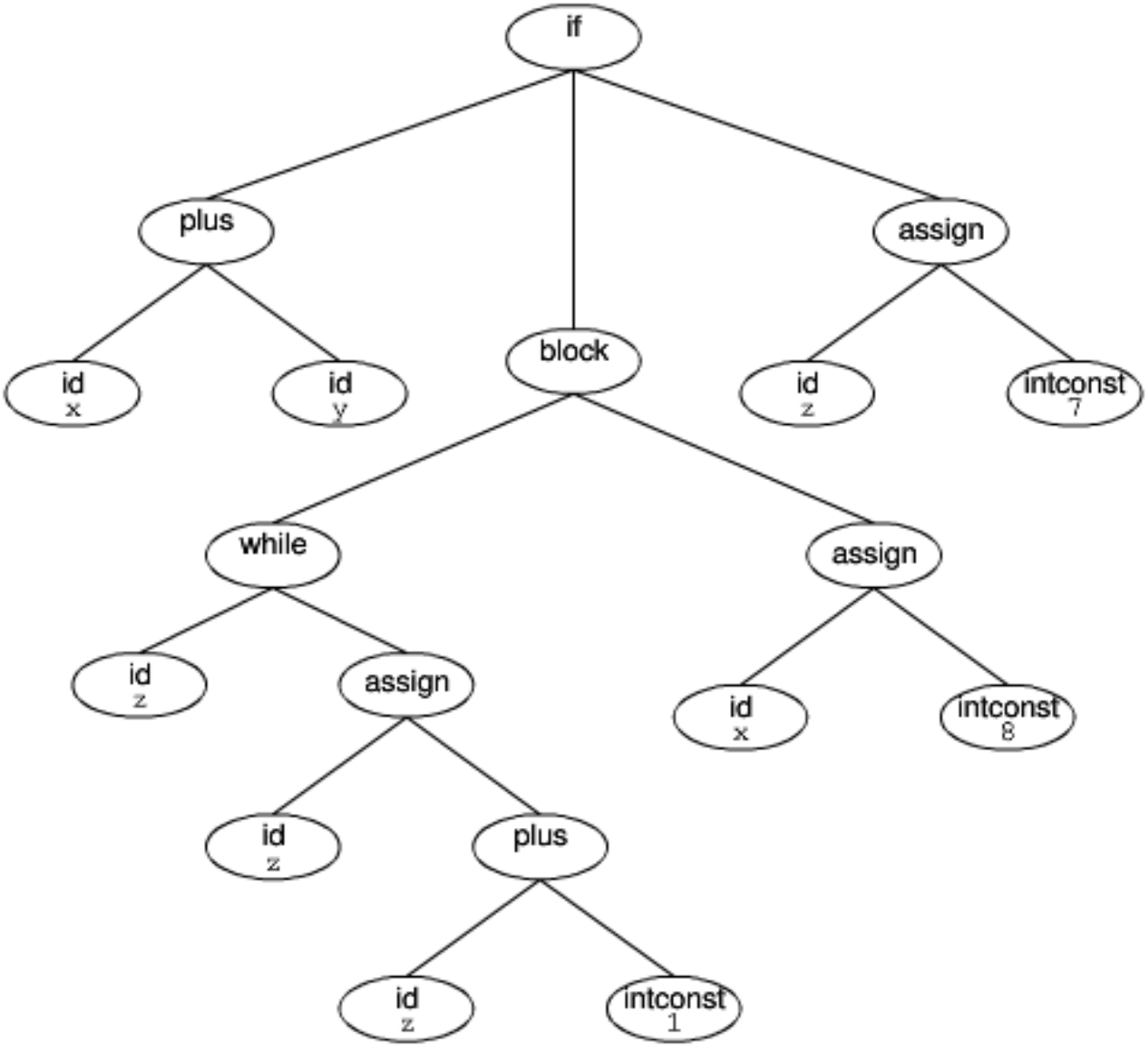
                        becomes

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); :}
```

Concrete Syntax Tree



Abstract Syntax Tree



Left and Right Values

- Identifiers can mean one of two things:
  - The value of the id (R-value)
  - The location of the id (L-value)

- For example:

```
x = y
```

- Do we care about the location or value of x?
  - Of y?

Left and Right Values

- Some languages provide operators for conversion
  - Classic example: C/C++
    - *: R-value -> L-value
    - &: L-value -> Rvalue

- What about Java?

Left and Right Values

```
1  Start          → Stmt_ast $
                    return (ast)
2  Stmt_result    → L_target assign R_source
                    result ← MAKEFAMILY(assign, target, source)
3  L_result       → id_name
                    result ← MAKENODE(name)
4                   | deref R_val
                    result ← val                                  (25)
5  R_result       → L_val
                    result ← MAKEFAMILY(deref, val)               (26)
6                   | num_val
                    result ← MAKENODE(val)
7                   | addr L_val
                    result ← val                                  (27)
```

- What is the AST for

  x = y

  x = &y

  *x = y

Practice

- Design an AST structure for a Java for loop
- Construct an AST for the following code snippet:

```
x = a + b;
y = a * b;
while (y > a) {
    a = a + 1;
    x = a + b;
}
```

Practice

- Design an AST and add semantic actions to this CUP specification, using the methods from slide 9.

```
Program :== List
        ;
List    :== List Line
        |   Line
        ;
Line    :== num Stmt
        ;
Stmt    :== if Expr then SimpleSt
        |   SimpleSt
        ;
SimpleSt:== var = Expr
        |   print Expr
        |   goto num
        ;
```