

# Semantic Analysis

Mooly Sagiv

<http://www.cs.tau.ac.il/~msagiv/courses/wcc10.html>

# Outline

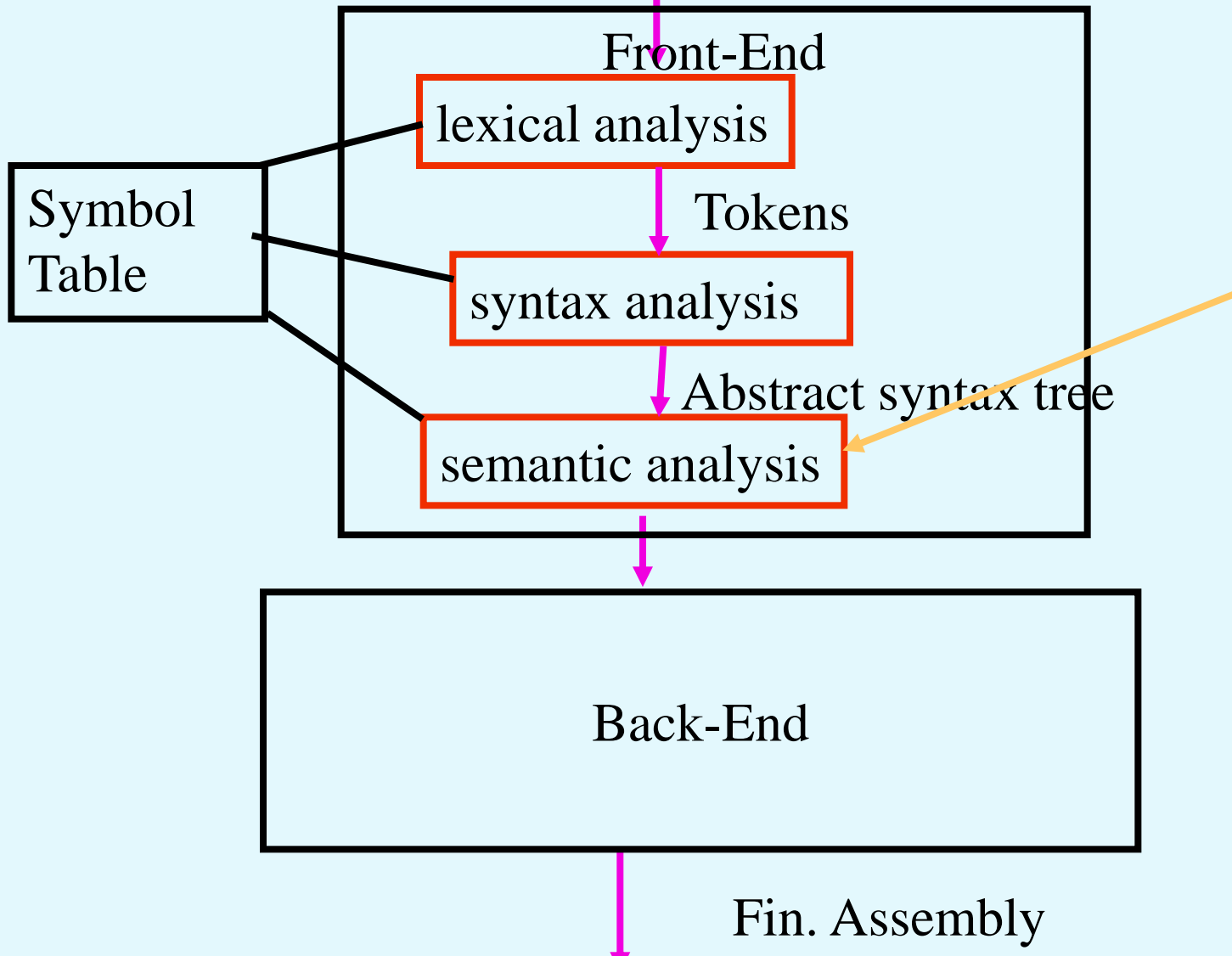
- What is Semantic Analysis
- Why is it needed?
- Scopes and type checking for imperative languages (Chapter 6)
- Attribute grammars (Chapter 3)

# Semantic Analysis

- The “meaning of the program”
- Requirements related to the “context” in which a construct occurs
- Context sensitive requirements - cannot be specified using a context free grammar (Context handling)
- Requires complicated and unnatural context free grammars
- Guides subsequent phases

# Basic Compiler Phases

Source program (string)



# Example Semantic Condition

- In C
  - **break** statements can only occur inside **switch** or loop statements

# Partial Grammar for C

$\text{Stm} \rightarrow \text{Exp};$

$\text{Stm} \rightarrow \text{if } (\mathbf{Exp}) \text{ Stm}$

$\text{StList} \rightarrow \text{StList Stm}$

$\text{Stm} \rightarrow \text{if } (\mathbf{Exp}) \text{ Stm else Stm}$

$\text{StList} \rightarrow \varepsilon$

$\text{Stm} \rightarrow \text{while } (\mathbf{Exp}) \text{ do Stm}$

$\text{Stm} \rightarrow \text{break};$

$\text{Stm} \rightarrow \{ \text{StList} \}$

# Refined Grammar for C

$\text{Stm} \rightarrow \text{Exp};$

$\text{Stm} \rightarrow \text{if } (\mathbf{Exp}) \text{ Stm}$

$\text{Stm} \rightarrow \text{if } (\mathbf{Exp}) \text{ Stm else Stm}$

$\text{Stm} \rightarrow \text{while } (\mathbf{Exp}) \text{ do LStm}$

$\text{Stm} \rightarrow \{ \text{StList} \}$

$\text{StList} \rightarrow \text{StList Stm}$

$\text{StList} \rightarrow \varepsilon$

$\text{LStm} \rightarrow \text{Exp};$

$\text{LStm} \rightarrow \text{if } (\mathbf{Exp}) \text{ LStm}$

$\text{LStm} \rightarrow \text{if } (\mathbf{Exp}) \text{ LStm else LStm}$

$\text{LStm} \rightarrow \text{while } (\mathbf{Exp}) \text{ do LStm}$

$\text{LStm} \rightarrow \{ \text{LStList} \}$

$\text{LStm} \rightarrow \text{break};$

$\text{LStList} \rightarrow \text{LStList LStm}$

$\text{LStList} \rightarrow \varepsilon$

# A Possible Abstract Syntax for C

```
package Absyn;
abstract public class Absyn { public int pos ;}
class Exp extends Absyn { };
class Stmt extends Absyn { } ;
class SeqStmt extends Stmt { public Stmt fstSt; public Stmt secondSt;
    SeqStmt(Stmt s1, Stmt s2) {  fstSt = s1; secondSt s2 ; }
}
class IfStmt extends Stmt { public Exp exp; public Stmt thenSt; public Stmt elseSt;
    IfStmt(Exp e, Stmt s1, Stmt s2) {  exp = e; thenSt = s1; elseSt s2 ; }
}
class WhileStmt extends Stmt {public Exp exp; public Stmt body;
    WhileSt(Exp e; Stmt s) { exp =e ; body = s; }
}
class BreakSt extends Stmt { };
```



# Partial CUP Specification

```
...
%%
stm ::= IF '(' exp: e ')' stm: s { : RESULT = new IfStm(e, s, null) ; : }
      | IF '(' exp: e ')' stm: s1 ELSE stm: s2 { : RESULT = new IfStm(e, s1, s2) ; : }
      | WHILE '(' exp: e ')' stm: s { : RESULT = new WhileStm(e, s) ; : }
      | '{ s: stmList }' { : RESULT = s ; : }
      | BREAK ';' { : RESULT = new BreakStm() ; : }
      ;
stmList ::= stmList: s1 stmt: s2 { : RESULT = new SeqStm(s1, s2) ; : }
          | /* empty */ { : RESULT = null ; : }
```

# A Semantic Check

(on the abstract syntax tree)

```
static void checkBreak(Stmt st)
{
    if (st instanceof SeqSt) {
        SeqSt seqst = (SeqSt) st;
        checkBreak(seqst.fstSt); checkBreak(seqst.secondSt);
    }
    else if (st instanceof IfSt) {
        IfSt ifst = (IfSt) st;
        checkBreak(ifst.thenSt); checkBreak(ifst.elseSt);
    }
    else if (st instanceof WhileSt) ; // skip
    else if (st instanceof BreakSt) {
        System.error.println("Break must be enclosed within a loop".
            st.pos); }
}
```

# Syntax Directed Solution

```
parser code {:  
public int loop_count = 0 ;  
:}  
stm      : := exp ';'   
          | IF '(' exp ')' stm   
          | IF '(' exp ')' stm ELSE stm   
          | WHILE '(' exp ')' m stm { : loop_count--; : }   
          | '{' stmList '}'   
          | BREAK ';' { : if (loop_count == 0)   
            system.error.println("Break must be enclosed within a loop");   
            : }   
          ;  
stmList ::= stmList st   
          | /* empty */   
          ;  
m       ::= /* empty */ { : loop_count++; ; : }   
          ;
```

# Problems with Syntax Directed Translations

- Grammar specification may be tedious (e.g., to achieve LALR(1))
- May need to rewrite the grammar to incorporate different semantics
- Modularity is impossible to achieve
- Some programming languages allow forward declarations (Algol, ML and Java)

## Example Semantic Condition: Scope Rules

- Variables must be defined within scope
- Dynamic vs. Static Scope rules
- Cannot be coded using a context free grammar

# Dynamic vs. Static Scope Rules

```
procedure p;  
    var x: integer  
    procedure q;  
        begin { q }  
            ...  
            x  
            ...  
        end { q };  
    procedure r;  
        var x: integer  
        begin { r }  
            q;  
        end; { r }  
begin { p }  
    q;  
    r;  
end { p }
```

# Example Semantic Condition

- In Pascal  
Types in assignment must be “compatible”

# Partial Grammar for Pascal

$\text{Stm} \rightarrow \text{id Assign Exp}$

$\text{Exp} \rightarrow \text{IntConst}$

$\text{Exp} \rightarrow \text{RealConst}$

$\text{Exp} \rightarrow \text{Exp} + \text{Exp}$

$\text{Exp} \rightarrow \text{Exp} - \text{Exp}$

$\text{Exp} \rightarrow ( \text{Exp} )$



# Refined Grammar for Pascal

$\text{Stm} \rightarrow \text{RealId Assign RealExp}$

$\text{Stm} \rightarrow \text{IntExp Assign IntExp}$

$\text{Stm} \rightarrow \text{RealId Assign IntExp}$

$\text{RealExp} \rightarrow \text{RealConst}$

$\text{IntExp} \rightarrow \text{IntConst}$

$\text{RealIntExp} \rightarrow \text{RealId}$

$\text{IntExp} \rightarrow \text{IntId}$

$\text{RealExp} \rightarrow \text{RealExp} + \text{RealExp}$

$\text{IntExp} \rightarrow \text{IntExp} + \text{IntExp}$

$\text{RealExp} \rightarrow \text{RealExp} + \text{IntExp}$

$\text{RealExp} \rightarrow \text{IntExp} + \text{RealExp}$

$\text{IntExp} \rightarrow \text{IntExp} - \text{IntExp}$

$\text{RealExp} \rightarrow \text{RealExp} - \text{RealExp}$

$\text{RealExp} \rightarrow \text{RealExp} - \text{IntExp}$

$\text{IntExp} \rightarrow ( \text{IntExp} )$

$\text{RealExp} \rightarrow \text{IntExp} - \text{RealExp}$

$\text{RealExp} \rightarrow ( \text{RealExp} )$

# Syntax Directed Solution

```
%%  
...  
stm ::= id:i Assign exp:e { : compatAss(lookup(i), e) ; :}  
;  
exp ::= exp:e1 PLUS exp:e2 { : compatOp(Op.PLUS, e1, e2);  
                               RESULT = opType(Op.PLUS, e1, e2); :}  
| exp:e1 MINUS exp:e2 { : compatOp(Op.MINUS, e1, e2);  
                               RESULT = opType(Op.MINUS, e1, e2); :}  
| ID: i { : RESULT = lookup(i); :}  
| INCONST { : RESULT = new TyInt() ; :}  
| REALCONST { : RESULT = new TyReal(); :}  
| ‘(‘ exp: e ‘)’ { : RESULT = e ; :}  
;
```

# Type Checking

## (Imperative languages)

- Identify the type of every expression
- Usually one or two passes over the syntax tree
- Handle scope rules

# Types

- What is a type
  - Varies from language to language
- Consensus
  - A set of values
  - A set of operations
- Classes
  - One instantiation of the modern notion of types

# Why do we need type systems?

- Consider assembly code
  - 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 does not 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!
- The goal of **type inference** is to infer a unique type for every “valid expression”

# Type Checking Overview

- Three kinds of languages
  - Statically typed: (Almost) all checking of types is done as part of compilation
    - Semantic Analysis
    - C, Java, ML
  - Dynamically typed: Almost all checking of types is done as part of program execution
    - Code generation
    - Scheme, Python
  - Untyped
    - No type checking (Machine Code)



# Type Wars

- Competing views on static vs. dynamic typing
- Static typing proponents say:
  - Static checking catches many programming errors
  - Prove properties of your code
  - Avoids the overhead of runtime type checks
- Dynamic typing proponents say
  - Static type systems are restrictive
  - Rapid prototyping difficult with type systems
  - Complicates the programming language and the compiler
  - Compiler optimizations can hide costs

# Type Wars (cont.)

- In practice, most code is written in statically typed languages with escape mechanisms
  - Unsafe casts in C Java
  - union in C
- It is debatable whether this compromise represents the best or worst of both worlds

# Soundness of type systems

- For every expression  $e$ ,
  - for every value  $v$  of  $e$  at runtime
    - $v \in \text{val}(\text{type}(e))$
- The type may actually describe more values
- The rules can reject logically correct programs
- Becomes more complicated with subtyping (inheritance)

# Issues in Semantic Analysis Implementation

- Name Resolution
- Type Checking
  - Type Equivalence
  - Type Coercions
  - Casts
  - Polymorphism
  - Type Constructors

# Name Resolution (Identification)

- Connect **applied occurrences** of an identifier/operator to its **defining occurrence**

```
month: Integer RANGE [1..12];  
...  
month := 1  
while month <> 12 do  
    print_string(month_name[month]);  
    month := month + 1;  
done;
```

The diagram illustrates the resolution of the variable 'month'. It shows a defining declaration at the top: 'month: Integer RANGE [1..12];'. Below it, several occurrences of 'month' are shown: 'month := 1', 'month <> 12', 'month := month + 1', and 'month\_name[month]'. Arrows point from each of these occurrences back to the defining declaration, indicating that they all refer to the same variable.

# Name Resolution (Identification)

- Connect **applied occurrences** of an identifier/operator to its **defining occurrence**
- Forward declarations
- Separate name spaces

```
struct one_int {  
    int i ;  
} i;  
i.i = 3;
```

- Scope rules

# A Simple Implementation

- A separate table per scope/name space
- Record properties of identifiers
- Create entries for defining occurrences
- Search for entries for applied occurrences
- Create table per scope enter
- Remove table per scope enter
- Expensive search

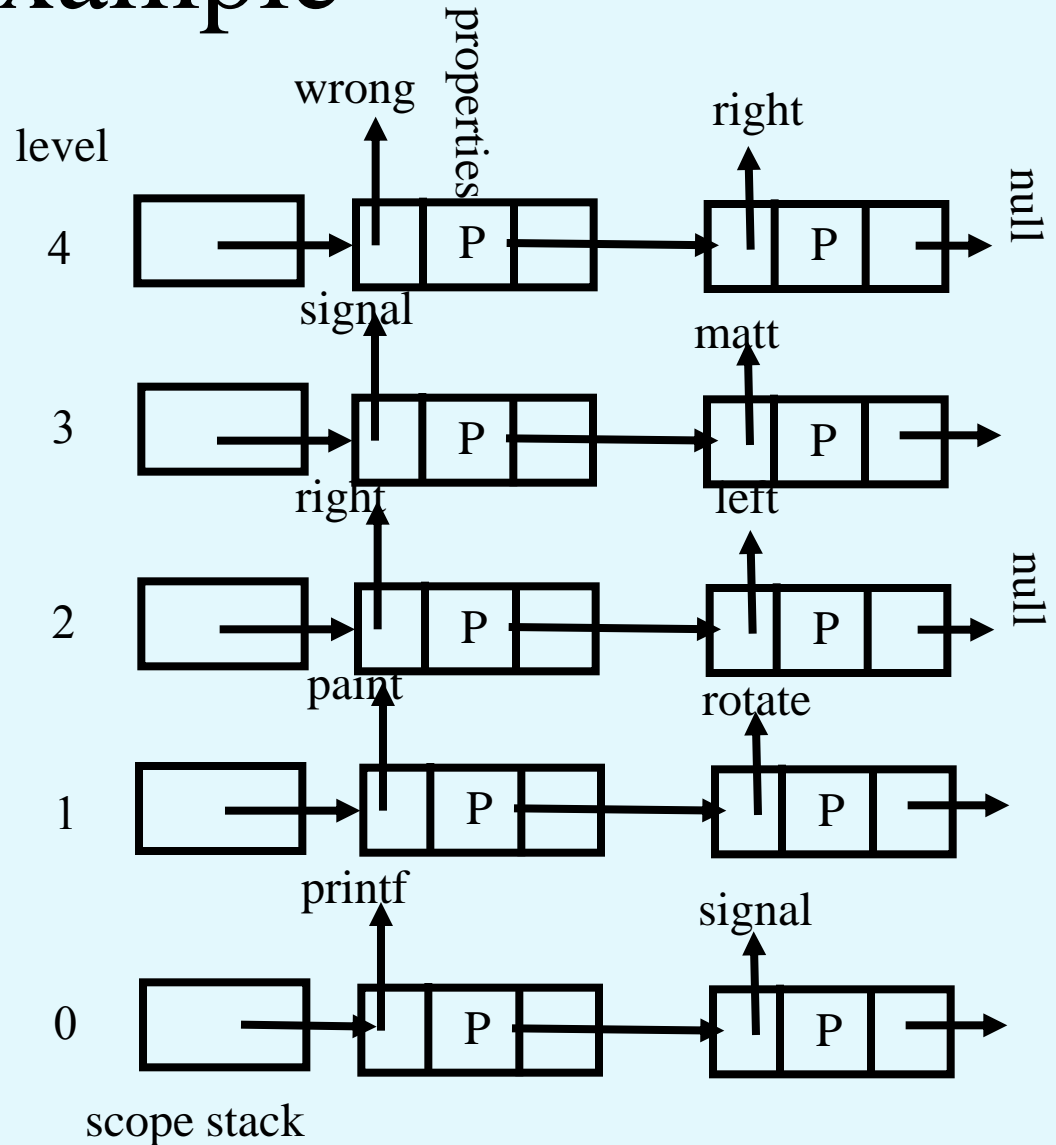
# Example

```

void roate(double angle) {
...
}

void paint(int left, int right) {
  Shade matt, signal;
...
{
  Counter right; wrong ;
...
}
}

```



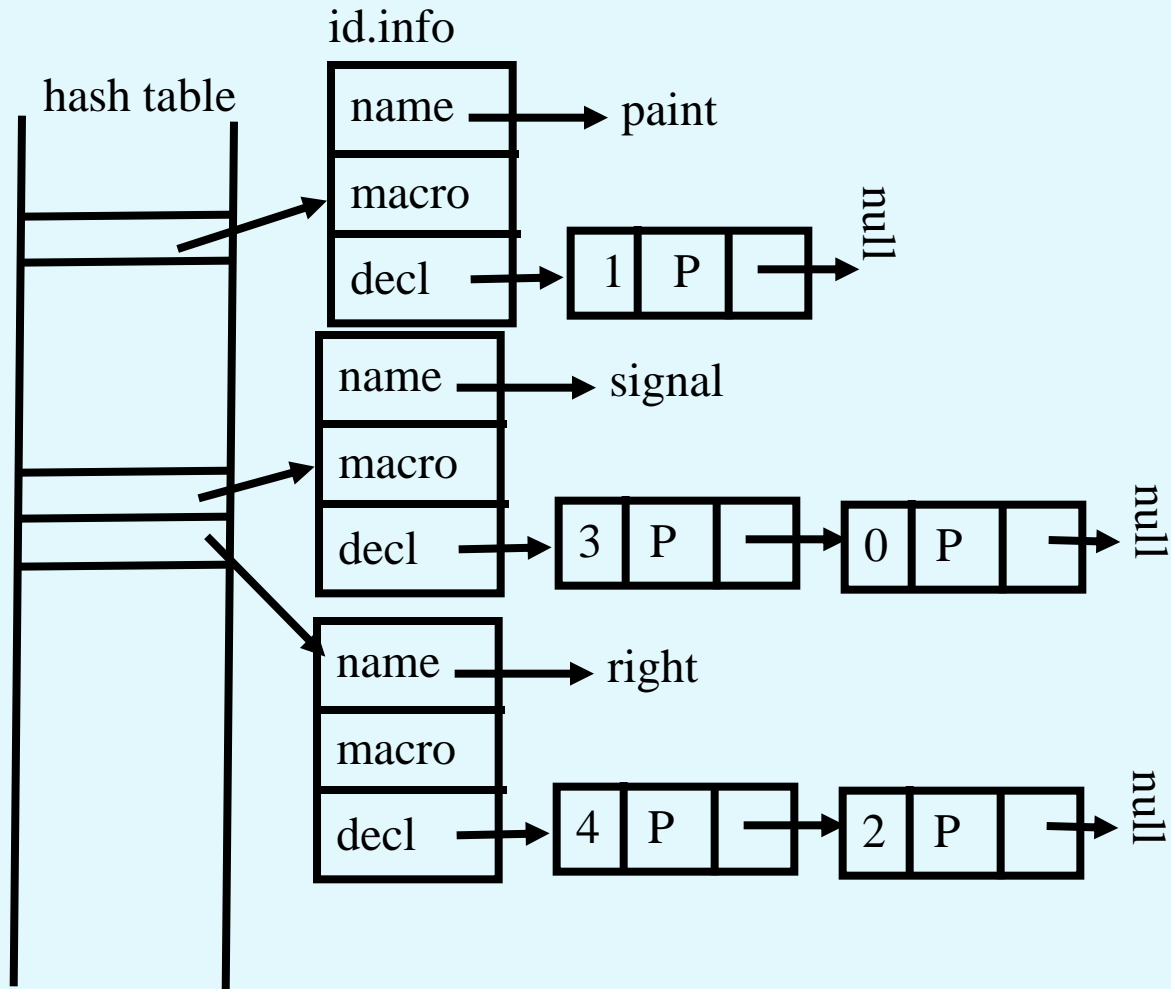


# A Hash-Table Based Implementation

- A unified hashing table for all occurrences
- Separate entries for every identifier
- Ordered lists for different scopes
- Separate table maps scopes to the entries in the hash
  - Used for ending scopes

# Example

```
void roate(double angle) {  
...  
}  
void paint(int left, int right) {  
  Shade matt, signal;  
...  
{  
  Counter right; wrong ;  
...  
}  
}
```



# Example(cont.)

```
void roate(double angle) {
```

```
...
```

```
}
```

```
void paint(int left, int right) {
```

```
  Shade matt, signal;
```

```
...
```

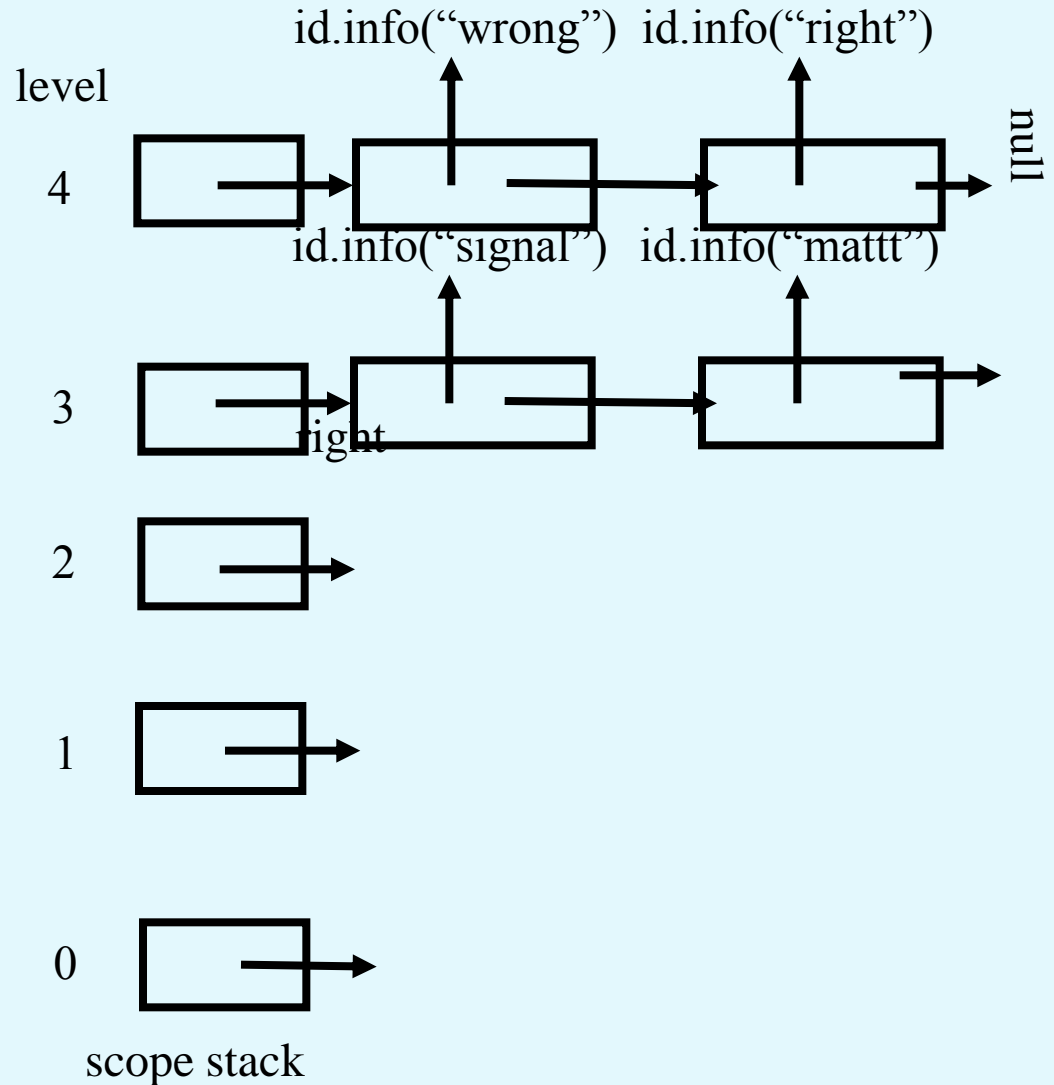
```
{
```

```
  Counter right; wrong ;
```

```
...
```

```
}
```

```
}
```



# Overloading

- Some programming languages allow to resolve identifiers based on the context
  - $3 + 5$  is different than  $3.1 + 5.1$
- Overloading user defined functions  
PUT(s: STRING) PUT(i: INTEGER)
- Type checking and name resolution interact
- May need several passes

# Type Checking

- Non-trivial
- Construct a type table (separate name space)
- May require several passes

# Type Equivalence

- Name equivalence

- TYPE t1 = ARRAY[Integer] of Integer;
- TYPE t2 = ARRAY[Integer] of Integer;
- TYPE t3 = ARRAY[Integer] of Integer;
- TYPE t4 = t3;

- Structural equivalence

- TYPE t5= RECORD {c: Integer ; p: Pointer to t5;}
- TYPE t6= RECORD {c: Integer ; p: Pointer to t6 ;}
- TYPE t7 = RECORD {c: Integer ; p : Pointer to  
RECORD {c: Integer ; p: Pointer to t5;}}

# Simple Inference

- The type of an expression depends on the type of the arguments and the required result
- If  $e_1$  has type Integer and  $e_2$  has type Integer then the result has type Integer

# Corner Cases

- What about power operator



# Casts and Coercions

- The compiler may need to insert implicit conversions between types  
float x = 5;
- The programmer may need to insert explicit conversions between types

# L-values vs. R-values

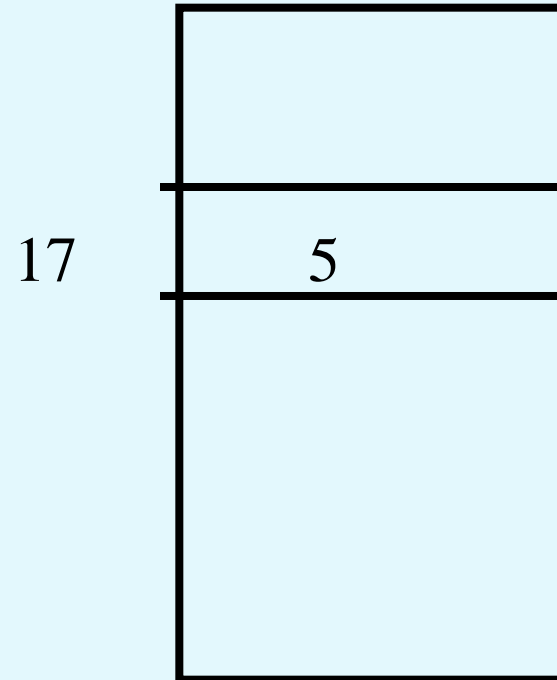
- Assignment  $x := \text{exp}$  is compiled into:
  - Compute the **address** of  $x$
  - Compute the **value** of  $\text{exp}$
  - Store the value of  $\text{exp}$  into the address of  $x$
- Generalization
  - R-value
    - Maps program expressions into semantic values
  - L-value
    - Maps program expressions into locations
    - Not always defined
  - Java has no small L-values

# A Simple Example

```
int x = 5;
```

```
x = x + 1;
```

Runtime memory



# A Simple Example

`int x = 5;`

`lvalue(x)=17, rvalue(x) =5`

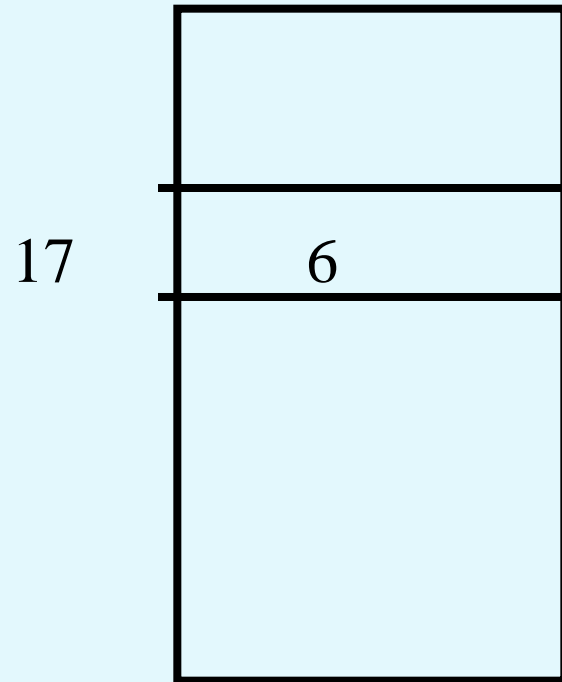
`lvalue(5)=⊥, rvalue(5)=5`

`x = x + 1;`

`lvalue(x)=17, rvalue(x) =5`

`lvalue(5)=⊥, rvalue(5)=5`

Runtime memory



# Partial rules for Lvalue in C

- Type of  $e$  is pointer to  $T$
- Type of  $e1$  is integer
- $\text{lvalue}(e2) \neq \perp$

<b>exp</b>	<b>lvalue</b>	<b>rvalue</b>
<b>id</b>	$\text{location}(\text{id})$	$\text{content}(\text{location}(\text{id}))$
<b>const</b>	$\perp$	$\text{value}(\text{const})$
<b>*e</b>	$\text{rvalue}(e)$	$\text{content}(\text{rvalue}(e))$
<b>&amp;e2</b>	$\perp$	$\text{lvalue}(e2)$
<b>e + e1</b>	$\perp$	$\text{rvalue}(e) + \text{sizeof}(T) * \text{rvalue}(e1)$

# Kind Checking

Defined L-values in assignments

expected

found

	lvalue	rvalue
lvalue	-	deref
rvalue	error	-

# Type Constructors

- Record types
- Union Types
- Arrays

# Routine Types

- Usually not considered as data
- The data can be a pointer to the generated code



# Dynamic Checks

- Certain consistencies need to be checked at runtime in general
- But can be statically checked in many cases
- Examples
  - Overflow
  - Bad pointers

# Summary

- Semantic analysis requires multiple traversals of the AST
- Is there a generalization?

# Attribute Grammars [Knuth 68]

- Generalize syntax directed translations
- Every grammar symbol can have several attributes
- Every production is associated with evaluation rules
  - Context rules
- The order of evaluation is automatically determined
  - Dependency order
  - Acyclicity
- Multiple visits of the abstract syntax tree

# Attribute Grammar for Types

stm  $\rightarrow$  id Assign exp  
{ compat\_ass(id.type, exp.type) }

exp  $\rightarrow$  exp PLUS exp  
{ compat\_op(PLUS, exp[1].type, exp[2].type)  
exp[0].type = op\_type(PLUS, exp[1].type, exp[2].type) }

exp  $\rightarrow$  exp MINUS exp  
{ compat\_op(MINUS, exp[1].type, exp[2].type)  
exp[0].type = op\_type(MINUS, exp[1].type, exp[2].type) }

exp  $\rightarrow$  ID { exp.type = lookup(id.repr) }

exp  $\rightarrow$  INCONST { exp.type = ty\_int ; }

exp  $\rightarrow$  REALCONST { exp.type = ty\_real ; }

exp  $\rightarrow$  '(' exp ')' { exp[0].type = exp[1].type ; }

# Example Binary Numbers

$$Z \rightarrow L$$

$$Z \rightarrow L.L$$

$$L \rightarrow L B$$

$$L \rightarrow B$$

$$B \rightarrow 0$$

$$B \rightarrow 1$$

Compute the numeric value of Z

$Z \rightarrow L$

$\{ Z.v = L.v \}$

$Z \rightarrow L.L$

$\{ Z.v = L[1].v + L[2].v \}$

$L \rightarrow L B$

$\{ L[0].v = L[1].v + B.v \}$

$L \rightarrow B$

$\{ L.v = B.v \}$

$B \rightarrow 0$

$\{ B.v = 0 \}$

$B \rightarrow 1$

$\{ B.v = ? \}$

$Z \rightarrow L$

$$\{ Z.v = L.v \}$$

$Z \rightarrow L.L$

$$\{ Z.v = L[1].v + L[2].v \}$$

$L \rightarrow L B$

$$\{ L[0].v = L[1].v + B.v \}$$

$L \rightarrow B$

$$\{ L.v = B.v \}$$

$B \rightarrow 0$

$$\{ B.v = 0 \}$$

$B \rightarrow 1$

$$\{ B.v = 2^{B.s} \}$$

$Z \rightarrow L$

$$\{ Z.v = L.v \}$$

$Z \rightarrow L.L$

$$\{ Z.v = L[1].v + L[2].v \}$$

$L \rightarrow L B$

$$\{ L[0].v = L[1].v + B.v$$

$$B.s = L[0].s$$

$$L[1].s = L[0].s + 1 \}$$

}

$L \rightarrow B$

$$\{ L.v = B.v$$

$$B.s = L.s$$

}

$B \rightarrow 0$

$$\{ B.v = 0 \}$$

$B \rightarrow 1$

$$\{ B.v = 2^{B.s} \}$$



$Z \rightarrow L$

$$\{ Z.v = L.v$$

$$L.s = 0 \}$$

$Z \rightarrow L.L$

$$\{ Z.v = L[1].v + L[2].v$$

$$L[1].s = 0$$

$$L[2].s = ? \}$$

$L \rightarrow L B$

$$\{ L[0].v = L[1].v + B.v$$

$$B.s = L[0].s$$

$$L[1].s = L[0].s + 1 \}$$

}

$L \rightarrow B$

$$\{ L.v = B.v$$

$$B.s = L.s$$

}

$B \rightarrow 0$

$$\{ B.v = 0 \}$$

$B \rightarrow 1$

$$\{ B.v = 2^{B.s} \}$$

$Z \rightarrow L$

$$\{ Z.v = L.v \\ L.s = 0 \}$$

$Z \rightarrow L.L$

$$\{ Z.v = L[1].v + L[2].v \\ L[1].s = 0 \\ L[2].s = -L[2].l \}$$

$B \rightarrow 0$

$$\{ B.v = 0 \}$$

$B \rightarrow 1$

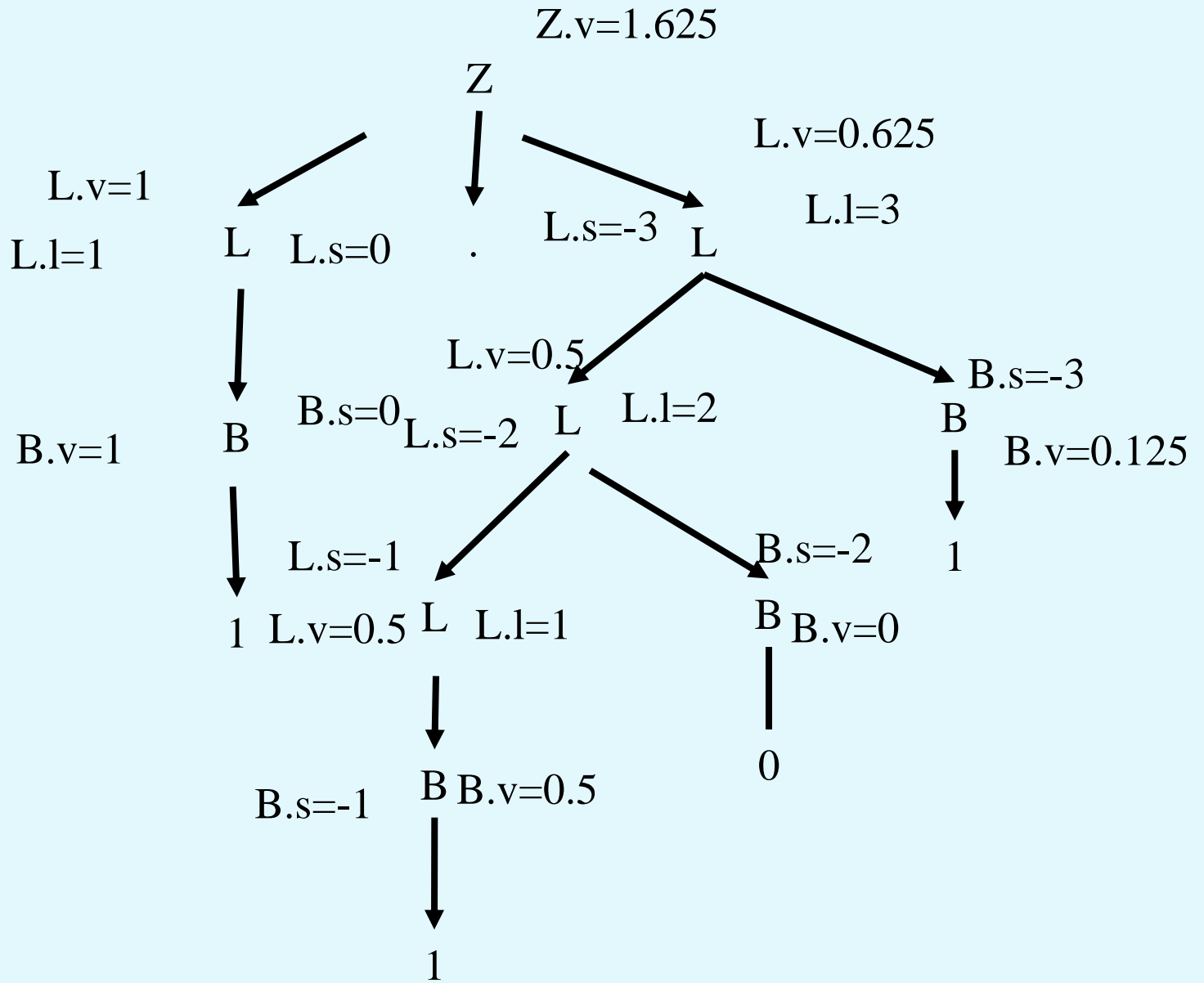
$$\{ B.v = 2^{B.s} \}$$

$L \rightarrow L B$

$$\{ L[0].v = L[1].v + B.v \\ B.s = L[0].s \\ L[1].s = L[0].s + 1 \\ L[0].l = L[1].l + 1 \}$$

$L \rightarrow B$

$$\{ L.v = B.v \\ B.s = L.s \\ L.l = 1 \}$$



# Summary

- Several ways to enforce semantic correctness conditions
  - syntax
    - Regular expressions
    - Context free grammars
  - syntax directed
  - traversals on the abstract syntax tree
  - later compiler phases?
  - Runtime?
- There are tools that automatically generate semantic analyzer from specification  
(Based on attribute grammars)