Context Analysis

Mooly Sagiv

html://www.cs.tau.ac.il/~msagiv/courses/wcc11-12.html

Short Decaf Program

Interface not declared

```
class MyClass implements MyInterface {
string myInteger;
void doSomething() {
   int[] x = new string;
   x[5] = myInteger * y   ;
   }
void doSomething() {
   }
int fibonacci(int n) {
    return doSomething() + fibonacci(n – 1);
   }
}
```

Type mismatch

y is undefined

Can’t multiple integers

Can’t redefine functions

Can’t add void

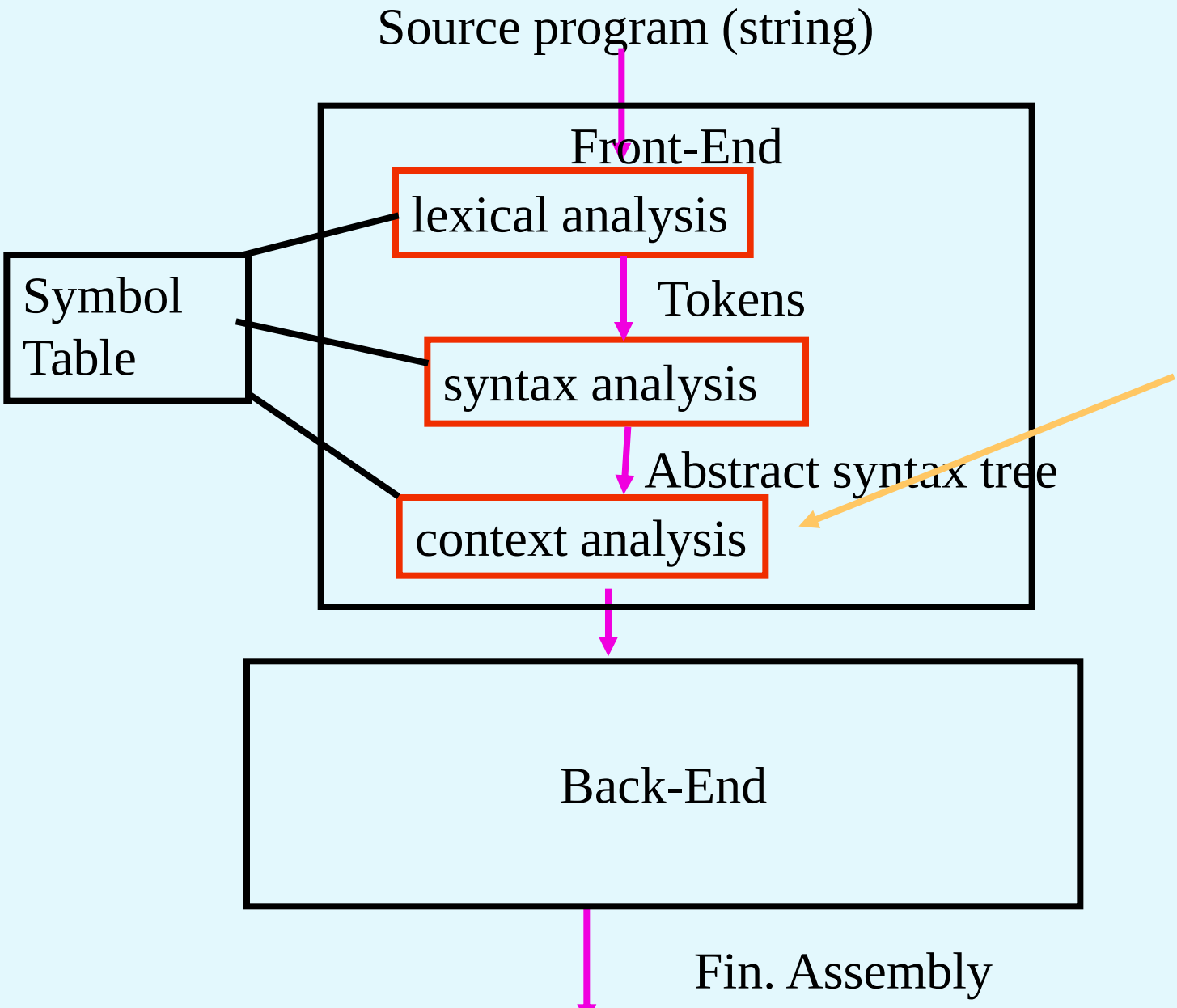
Outline

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

Context Analysis

- 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



Example Context Condition

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

Partial Grammar for C

Stm $\rightarrow$ Exp;

Stm $\rightarrow$ if (**Exp**) Stm

Stm $\rightarrow$ if (**Exp**) Stm else Stm

Stm $\rightarrow$ while (**Exp**) do Stm

Stm $\rightarrow$ break;

Stm$\rightarrow$ {StList }

StList $\rightarrow$ StList Stm

StList $\rightarrow \varepsilon$

Refined Grammar for C

Stm→Exp;

Stm → if (**Exp**) Stm

Stm → if (**Exp**) Stm else Stm

Stm→ while (**Exp**) do LStm

Stm→ {StList }

StList → StList Stm

StList → ε

LStm → Exp;

LStm → if (**Exp**) LStm

LStm→ if (**Exp**) LStm else LStm

LStm → while (**Exp**) do LStm

LStm → {LStList }

LStm → break;

LStList → LStList LStm

LStList → ε

A Possible Abstract Syntax for C

Stmt → Exp (Exp)
| Stmt Stmt (SeqStmt)
| Exp Stmt Stmt (IfStmt)
| Exp Stmt (WhileStmt)
| (BreakSt)

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 Context 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  BreakeSt) {
      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 Contexts
- Modularity is impossible to achieve
- Some programming languages allow forward declarations (Algol, ML and Java)

Example Context 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 Context Condition

- In Pascal
  Types in assignment must be “compatible”"

Partial Grammar for Pascal

Stm→ id Assign Exp

Exp → IntConst

Exp → RealConst

Exp→ Exp + Exp

Exp→ Exp -Exp

Exp→ (Exp)

Refined Grammar for Pascal

Stm→ RealId Assign RealExp

Stm→IntExpAssign IntExp

Stm→RealId Assign IntExp

RealExp → RealConst

RealIntExp → RealId

RealExp→ RealExp + RealExp

RealExp→ RealExp + IntExp

RealExp→ IntExp + RealExp

RealExp→ RealExp -RealExp

RealExp→ RealExp -RealExp

RealExp→ RealExp -IntExp

RealExp→ IntExp -RealExp

RealExp→ (RealExp)

IntExp → IntConst

IntExp → IntId

IntExp→ IntExp + IntExp

IntExp→ IntExp -IntExp

IntExp→ (IntExp)

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
    - Context 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

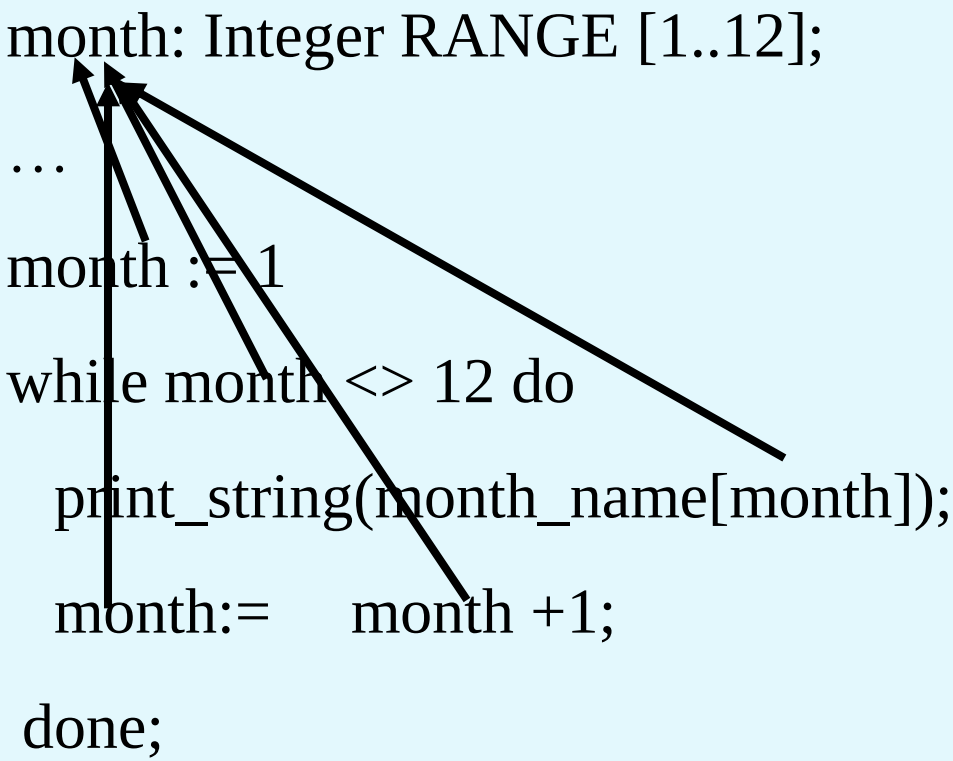
- For every expression e,
  - for every value v of e at runtime
    - v $\in$ val(type(e))
- The type may actually describe more values
- The rules can reject logically correct programs
- Becomes more complicated with subtyping (inheritance)

Issues in Context 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



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;
```

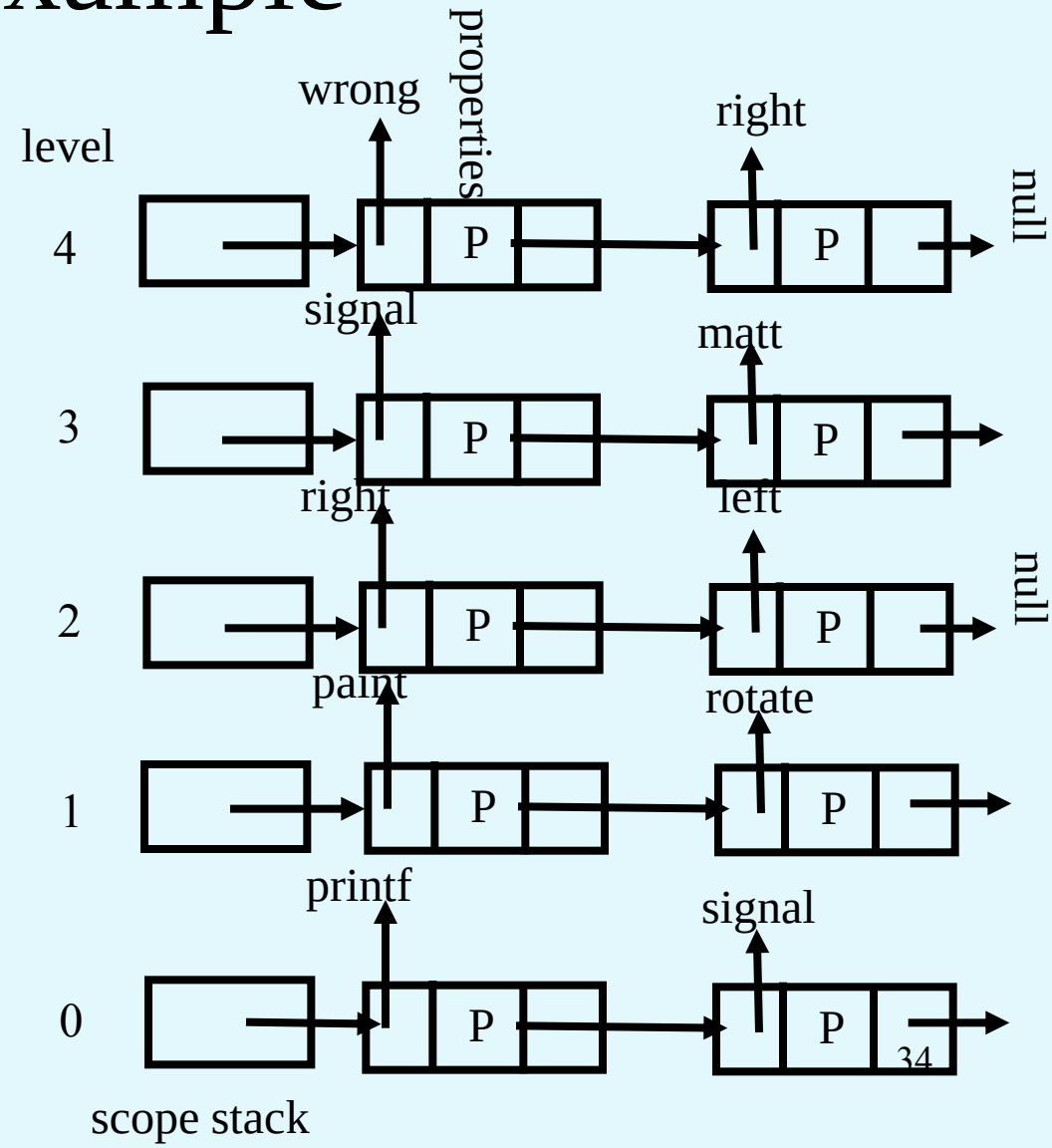
- Scope rules

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 ;
     …
    }
}
```

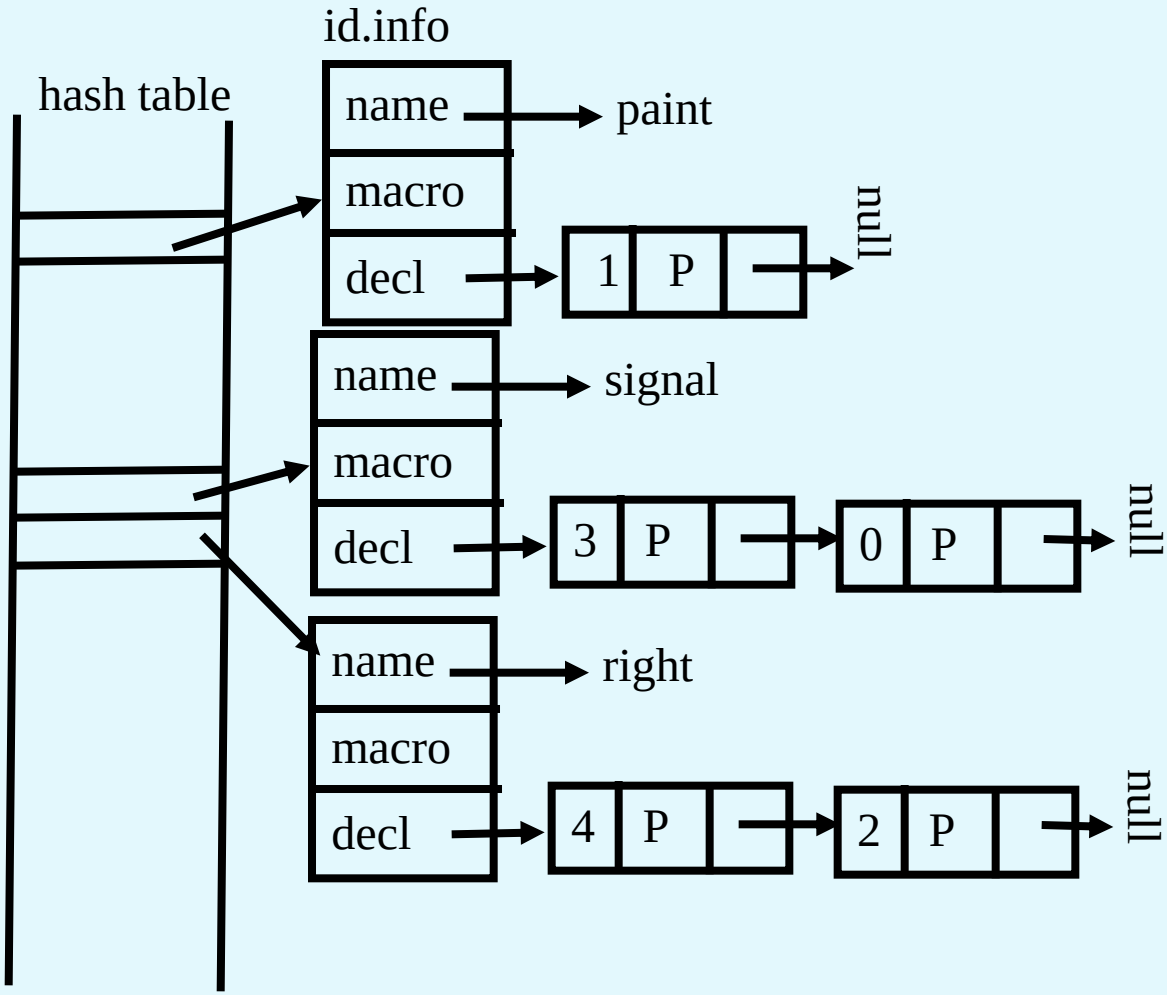


A Hash-Table Based Implementation

- 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

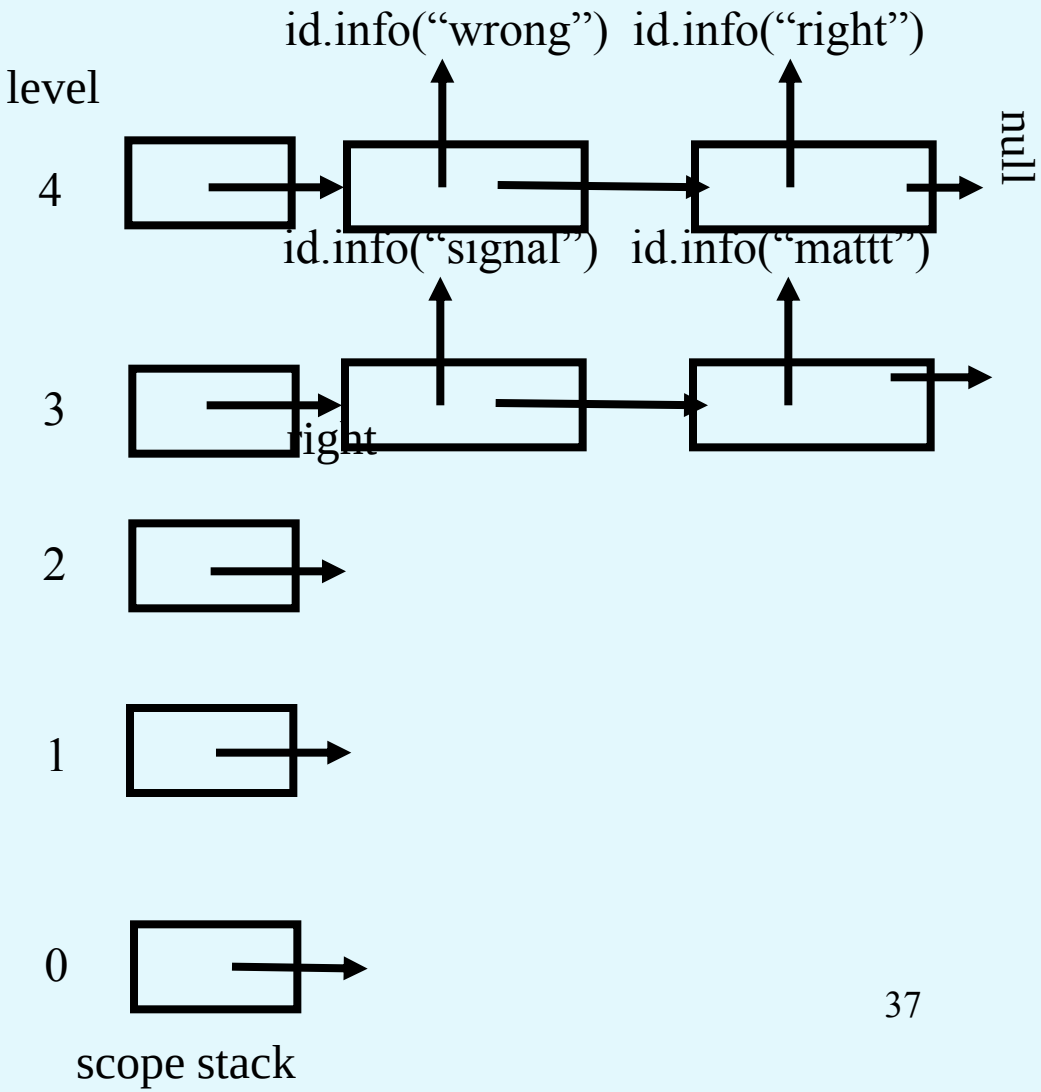
Example

```
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 e1 has type Integer and e2 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

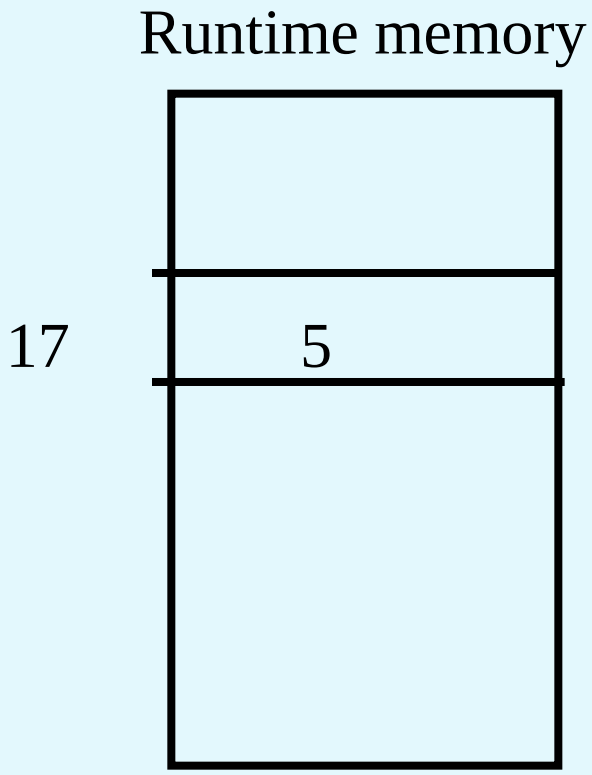
L-values vs. R-values

- Assignment x := exp is compiled into:
  - Compute the **address** of x
  - Compute the **value** of exp
  - Store the value of exp into the address of x
- Generalization
  - R-value
    - Maps program expressions into Context 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;
```



A Simple Example

int x = 5;

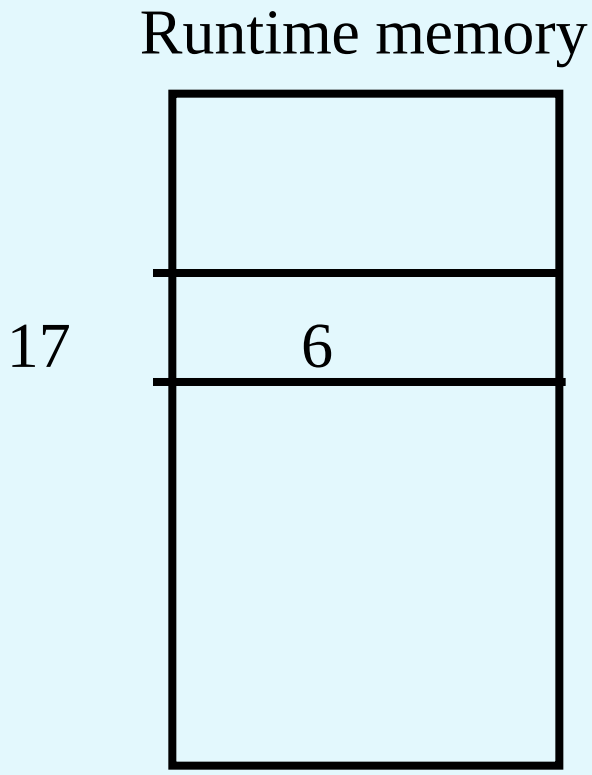
lvalue(x)=17, rvalue(x) =5

lvalue(5)=⊥, rvalue(5)=5

x = x + 1;

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

lvalue(5)=⊥, rvalue(5)=5



Partial rules for Lvalue in C

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

exp	**lvalue**	**rvalue**
id	location(id)	content(location(id))
const	$\perp$	value(const)
***e**	rvalue(e)	content(rvalue(e))
&e2	$\perp$	lvalue(e2)
e + e1	$\perp$	rvalue(e)+sizeof(T)*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

- Context 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

Summary

- Several ways to enforce Context 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 Context analyzer from specification
  (Based on attribute grammars)