# CMPT 379
# Compilers

Anoop Sarkar

http://www.cs.sfu.ca/~anoop



# Goal of Semantic Analysis

- Ensure that program obeys certain kinds of sanity checks
  - all used variables are defined
  - types are used correctly
  - method calls have correct number and types of parameters and return value

## Symbol Tables

- Symbol tables map **identifiers** (strings) to **descriptors** (information about identifiers)
- Basic Operation: Lookup
  - Given a string, find a descriptor
  - Typical Implementation: hash table
- Examples
  - Given a class name, find class descriptor
  - Given variable name, find descriptor
  - local descriptor, parameter descriptor, field descriptor



## Parameter Descriptors

- When build parameter descriptor, have
  - name of type
  - name of parameter
- What is the check? Must make sure name of type identifies a valid type
  - look up use of identifier (in context) in the symbol table
  - if not there, fails semantic check

# Local Symbol Table

- When building a local symbol table, have a list of local descriptors
- What to check for?
  - duplicate variable names
  - shadowed variable names
- When to check?
  - when descriptor is inserted into the local symbol table
- Parameter and field symbol tables are similar



# Symbol Tables

- Compilers use symbol tables to produce:
  - Object layout in memory
  - Code to
    - Access Object Fields
    - Access Local Variables
    - Access Parameters
    - Invoke methods

# Hierarchy In Symbol Tables

- Hierarchy Comes From
  - Nested Scopes: Local scope inside field scope
  - Inheritance: Child class inside parent class
- Nested scopes are annotations on the parse tree
- Symbol table hierarchy reflects the hierarchy
- Lookup proceeds up hierarchy until descriptor is found



# Blocks

```
main ()
{
    /* B0 */ int a = 0;  int b = 0;
    {
        /* B1 */ int b = 1;
        { /* B2 */ int a = 2; }
        { /* B3 */ int b = 3; }
    /* back to B1 */ }
/* back to B0 */ }
```

| |
|---|
| B0: a, b |
| B1: b |
| B2: a  B3: b |
| |

Symbol Table
Storage for Names

# Scoping Analysis
## symbol "liveness"

- Hierarchy in symbol tables can be implemented in various ways:

1. Using the nodes in the parse tree as part of the descriptor, and using bottom-up traversal from the variable use to detect valid use



# Scoping Analysis

2. Based on the local scoping binding for identifiers can be inserted and then after they go out of scope, the binding is deleted from the symbol table
3. Use the parse stack to store symbol tables:
   - Each block pushes a new symbol table onto the stack.
   - Symbols are searched from top of the stack down.
   - As the symbol goes out of scope, the symbol table is popped out of the stack

## Load Instruction

- Check instructions that store values into variables
- Source contains identifier with variable name
- Look up variable name:
  - If in local symbol table, reference local descriptor
  - If in parameter symbol table, reference parameter descriptor
  - If in field symbol table, reference field descriptor
  - If not found, semantic error



## Load Array Instruction

- Check instructions that load array variables
  - Variable name
  - Array index expression
- Semantic check:
  - Look up variable name (if not there, semantic error)
  - Check type of expression (if not integer, semantic error)

# Binary operators

- Check instructions that combine two expressions with a binary operator like + or *
- What can go wrong?
  - expressions have wrong type
  - both must be integers (for example)
- So compiler checks type of expressions
  - load instructions record type of accessed variable
  - operations record type of produced expression
  - so just check types, if wrong, semantic error



# Type Inference for Bin-op

- Most languages let you add floats, ints, doubles
- What are issues?
  - Types of result of add operation
  - Coercions on operands of add operation
- Standard rules usually apply
  - If add an int and a float, coerce the int to a float, do the add with the floats, and the result is a float.
  - If add a float and a double, coerce the float to a double, do the add with the doubles, result is double

# Summary of Semantic Checks

- Do semantic checks when build IR
- Many correspond to making sure entities are there to build correct IR
- Others correspond to simple sanity checks
- Each language has a list that must be checked
- Can flag many potential errors at compile time



# Equality of types

- Main semantic tasks involve liveness analysis and checking equality
- Equality checking of types (basic types) is crucial in ensuring that code generation can target the correct instructions
- Coercions also rely on equality checking of types
- But what about those objects in PLs (records, functions, etc) that are not basic types?
- Can we perform any semantic checks on these as well?

# Type Systems

- So far we have seen simple cases of type checking and coercion
- Basic types for data types: *boolean, char, integer, real*
- A basic type for lack of a type: *void*
- A basic type for a type error: *type_error*
- Based on these basic types we can build new types using type constructors



# Type Constructors

- Arrays: int p[10];
  - type: *array(10, integer)*
  - multi-dim arrays: int p[3][2]: *array(3, array(2, integer))*
- Products/tuples: pair<int, char> p(10,'a');
  - type: *integer* $\times$ *char*
- Records: struct { int p; char q; } data;
  - Type: *record((p* $\times$ *integer)* $\times$ *(q* $\times$ *char))*
- Pointers: int *p;
  - Type: *pointer(integer)*

# Type Constructors

- Functions: int foo (int p, char q) { return 2; }
  - Type: $integer \times char \rightarrow integer$
  - A function maps elements from the domain to the range
  - Function types map a domain type D to a range type R
  - A type for a function is denoted by $D \rightarrow R$
- In addition, type expressions can contain type variables
  - Example: $\alpha \times \beta \rightarrow \alpha$



# Equivalence of Type Exprs

- Check equivalence of type exprs: $s$ and $t$
- If $s$ and $t$ are basic types, then return true
- If $s = array(s_1, s_2)$ and $t = array(t_1, t_2)$ then return true if equal($s_1$, $t_1$) and equal($s_2$, $t_2$)
- If s = $s_1 \times s_2$ and $t = t_1 \times t_2$ then return true if equal($s_1$, $t_1$) and equal($s_2$, $t_2$)
- If $s = pointer(s_1)$ and $t = pointer(t_1)$ then return true if equal($s_1$, $t_1$)

# Polymorphic Functions

- Consider the following ML program:

  ***fun*** *null [] =* ***true***

  *| null (_::_) =* ***false****;*

  ***fun*** *tl (_::xs) = xs;*

  ***fun*** *length (alist) =*

  ***if*** *null(alist)* ***then*** *0*

  ***else*** *length(tl(alist)) + 1;*
- *null* tests if a list is empty
- *tl* removes first element and returns rest



# Polymorphic Functions

- *length* is a polymorphic function (different from polymorphism in object inheritance)
- The function *length* accepts lists with elements of any basic type:

  *length([‘a’, ‘b’, ‘c’])*

  *length([1, 2, 3])*

  *length([ [1,2,3], [4,5,6] ])*
- The type for *length* is *list*($\alpha$) $\rightarrow$ *integer*
- $\alpha$ can stand for any basic type: *integer* or *char*

## Polymorphic Functions

- Consider the following ML program:

  ***fun*** *map f [] = []*

  *| map f (x::xs) = (f(x)) :: map f xs;*
- *map* takes two arguments: a function *f* and a list
- It applies *f* to each element of the list and creates a new list with the range of *f*
- Type of *map*: $(\alpha \rightarrow \beta) \rightarrow list(\alpha) \rightarrow list(\beta)$



## Type Inference

- *Type inference* is the problem of determining the type of a statement from its body
- Similar to type checking and coercion
- But inference can be much more expressive when type variables can be used
- For example, the type of the *map* function on previous page uses type variables

# Type Variable Substitution

- We can take a type variable in a type expression and substitute a value
- In *list($\alpha$)* we can substitute the type *integer* for the variable $\alpha$ to get *list(integer)*
- *list(integer)* < *list($\alpha$)* means *list(integer)* is an instance of *list($\alpha$)*
- *S(t)* is a substitution for type expr *t*
- Replacing *integer* for $\alpha$ is a substitution



# Type Variable Substitution

- $s < t$ means $s$ is an instance of $t$
- Or $s$ is more specific than $t$
- Or $t$ is more general than $s$
- Some more examples:
  - $integer \rightarrow integer < \alpha \rightarrow \alpha$
  - $(integer \rightarrow integer) \rightarrow (integer \rightarrow integer) < \alpha \rightarrow \alpha$
  - $list(\alpha) < \beta$
  - $\alpha < \beta$

# Type Expr Unification

- Incorrect type variable substitutions:
  - $\textit{integer} < \textit{boolean}$
  - $\textit{integer} \rightarrow \textit{boolean} < \alpha \rightarrow \alpha$
  - $\textit{integer} \rightarrow \alpha < \alpha \rightarrow \alpha$
- In general, there are many possible substitutions
- Type exprs $s$ and $t$ unify if there is a substitution $S$ that is most general such that $S(s) = S(t)$
- Such a substitution $S$ is the *most general unifier* which imposes the fewest constraints on variables



# Example of Type Inference

- Example:

  ***fun*** *length (alist) =*

      ***if*** *null(alist)* ***then*** *0*

      ***else*** *length(tl(alist)) + 1;*
- *length* : $\alpha_1$
- *null : list*($\alpha_2$) $\rightarrow$ *boolean*
- *alist : list*($\alpha_2$)
- *null(alist) : boolean*

# Example (cont'd)

*__fun__ length (alist) =*
*__if__ null(alist) __then__ 0*
*__else__ length(tl(alist)) + 1;*

- $0 : integer$
- $tl : list(\alpha_3) \rightarrow list(\alpha_3)$
- $tl(alist) : list(\alpha_2)$
- $length : list(\alpha_2) \rightarrow \alpha_4$
- $length(tl(alist)) : \alpha_4$
- $1 : integer$
- $+ : integer \times integer \rightarrow integer$
- $if : boolean \times \alpha_5 \times \alpha_5 \rightarrow \alpha_5$
- $length : list(\alpha_2) \rightarrow integer$

$list(\alpha_2) \rightarrow \alpha_4 < \alpha_1$

$integer < \alpha_5$

$integer < \alpha_4$



# Unification

- Algorithm for finding the ***most general substitution*** *S* such that *S(s) = S(t)*
- Also called the ***most general unifier***
- *unify(m, n)* unifies two type exprs *m* and *n* and returns true/false if they can be unified
- Side effect is to keep track of the *mgu* substitution for unification to succeed

# Unification Algorithm

- We will explain the algorithm using an example:
  - E: $((\alpha 1 \rightarrow \alpha 2) \rightarrow list(\alpha 3)) \rightarrow list(\alpha 2)$
  - F: $((\alpha 3 \rightarrow \alpha 4) \rightarrow list(\alpha 3)) \rightarrow \alpha 5$
- What is the most general unifier?
  - $S_1(E) = S_1(F)$ $((\alpha 1 \rightarrow \alpha 1) \rightarrow list(\alpha 1)) \rightarrow list(\alpha 1)$

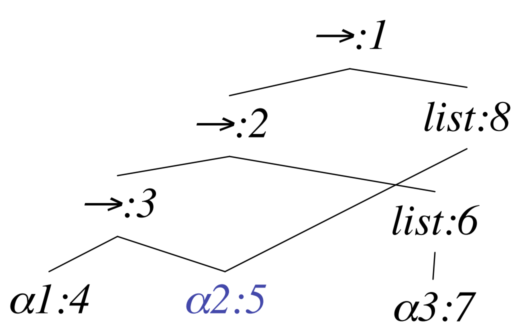
√ – $S_2(E) = S_2(F)$ $((\alpha 1 \rightarrow \alpha 2) \rightarrow list(\alpha 1)) \rightarrow list(\alpha 2)$

√ – $S_3(E) = S_3(F)$ $((\alpha 3 \rightarrow \alpha 2) \rightarrow list(\alpha 3)) \rightarrow list(\alpha 2)$



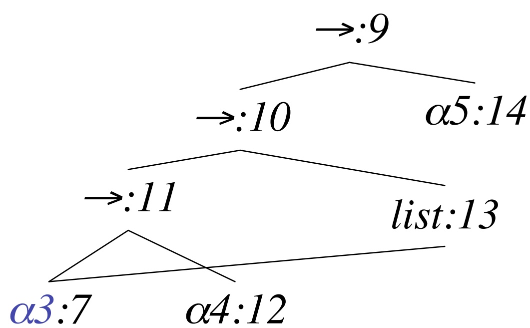
# Unification Algorithm

E: $((\alpha 1 \rightarrow \alpha 2) \rightarrow list(\alpha 3)) \rightarrow list(\alpha 2)$



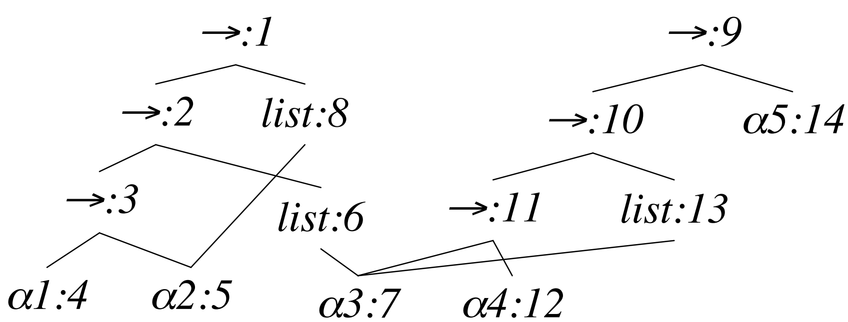
# Unification Algorithm

F: $((\alpha 3 \rightarrow \alpha 4) \rightarrow list(\alpha 3)) \rightarrow \alpha 5$

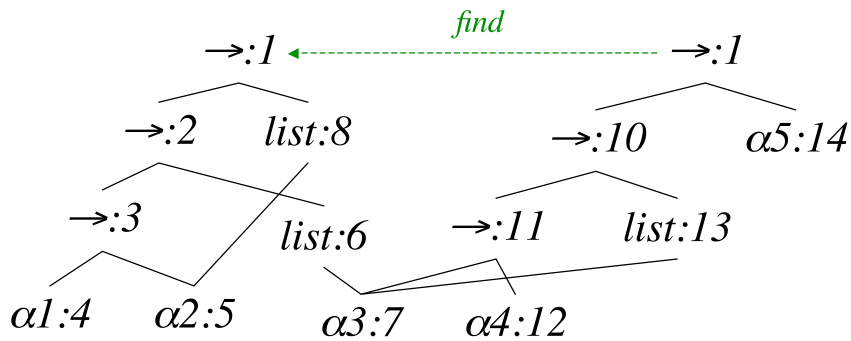





# Unify(1,9)

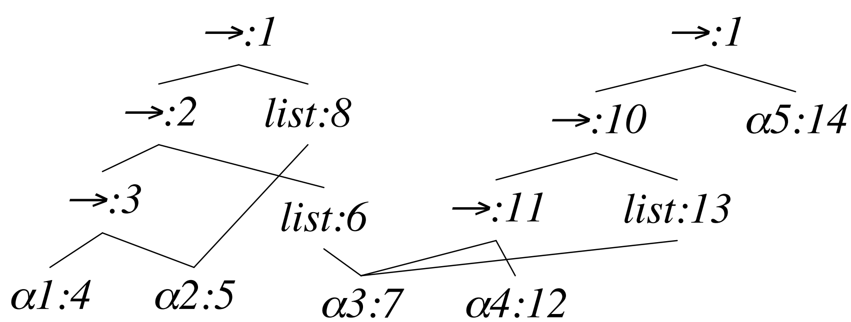


# Unify(1,9)

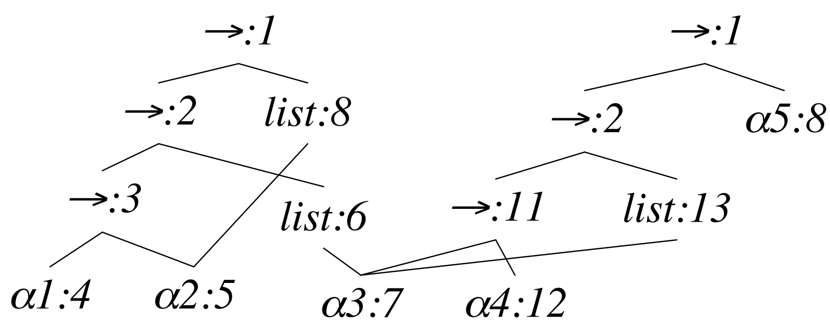





# Unify(2,10) *and* Unify(8,14)

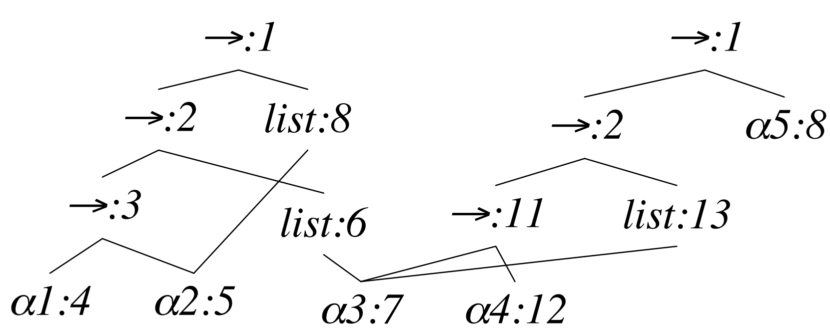


# Unify(2,10) *and* Unify(8,14)

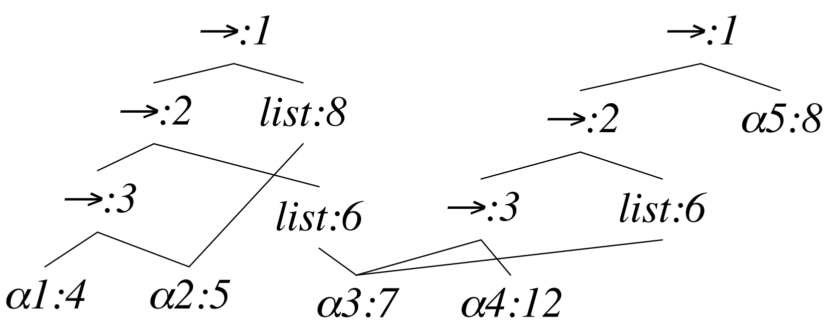





# Unify(3,11) *and* Unify(6,13)

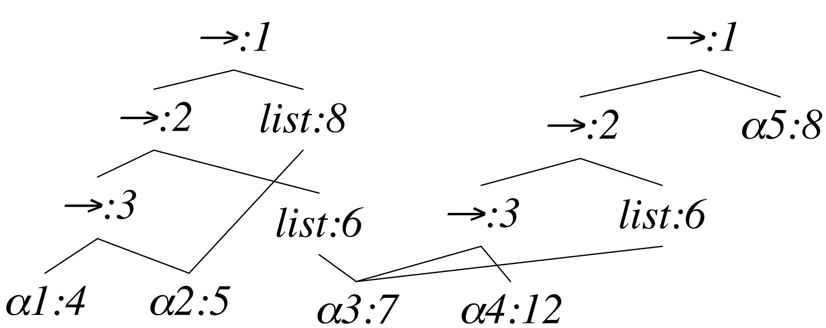


# Unify(3,11) *and* Unify(6,13)

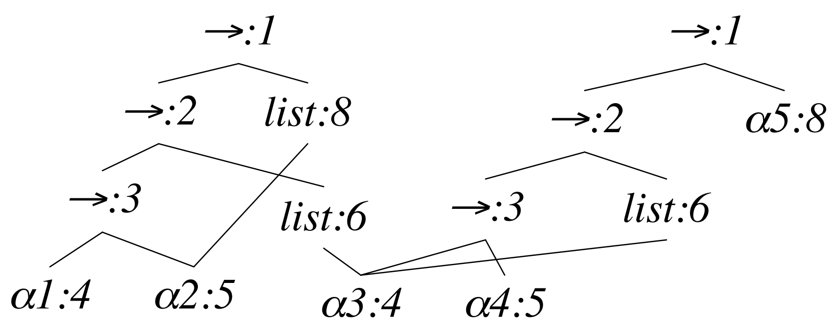





# Unify(4,7) *and* Unify(5,12)

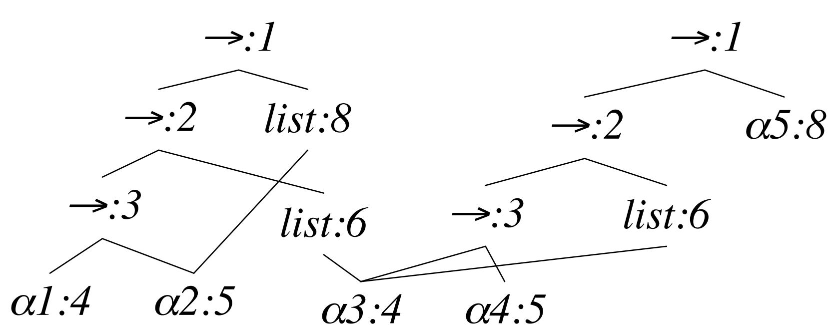


# Unify(4,7) *and* Unify(5,12)






# Unification success

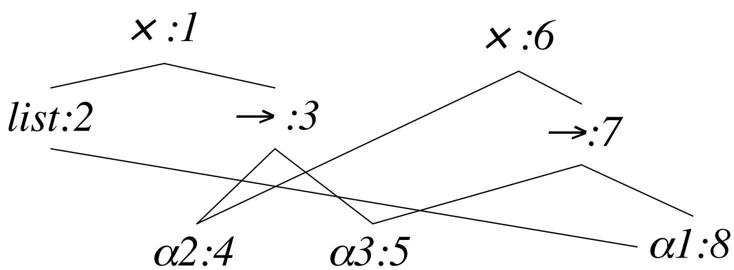



$((\alpha 1 \rightarrow \alpha 2) \rightarrow list(\alpha 1)) \rightarrow list(\alpha 2)$

# Unification: Occur Check

$list(\alpha 1) \times (\alpha 2 \rightarrow \alpha 3)$

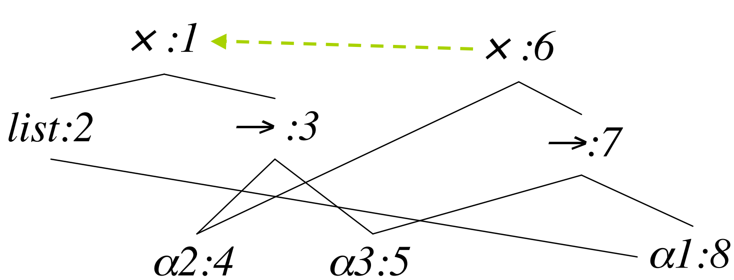
$\alpha 2 \times (\alpha 3 \rightarrow \alpha 1)$





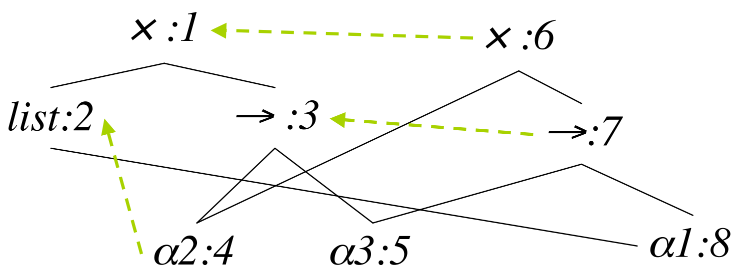

# Unify(1,6)

6--1



# Unify(2,4) and Unify(3,7)

6--1, 4--2, 7--3

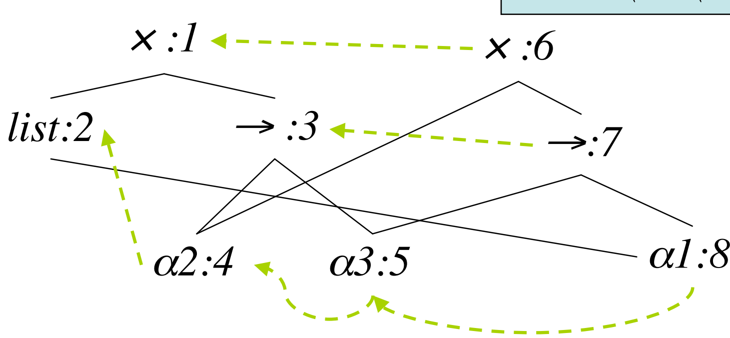





# Unify(4,5) and Unify(5,8)

6--1, 4--2, 7--3, 5--4, 8--5

- $list(\alpha 1)$
- $= list(\alpha 2)$
- $= list(list(\alpha 1)$



# Occur Check

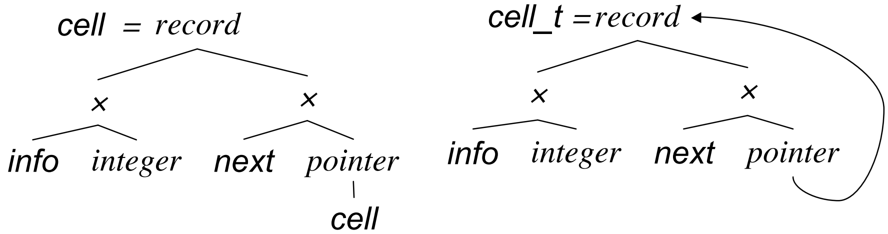
- Our unification algorithm creates a cycle in *find* for some inputs
- The cycle leads to an infinite loop. Note that Algorithm 6.32 in the Purple Dragon book has this bug
- A solution to this is to unify only if no cycles are created: the *occur check*
- Makes unification slower but correct



# Recursive types

- Recursive types arise naturally in PLs
- For example, in pseudo-C:

  *struct cell { int info; cell_t *next; } cell_t;*

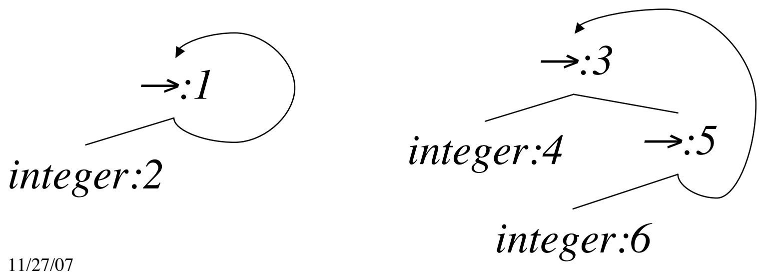


# Recursive type equivalence

- Are these recursive type expressions equivalent:

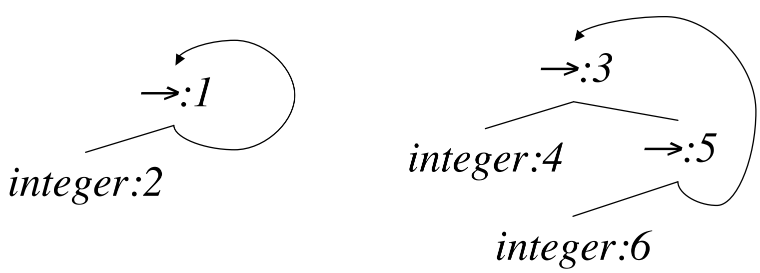
  $\alpha 1 = integer \rightarrow \alpha 1$

  $\alpha 2 = integer \rightarrow (integer \rightarrow \alpha 2)$

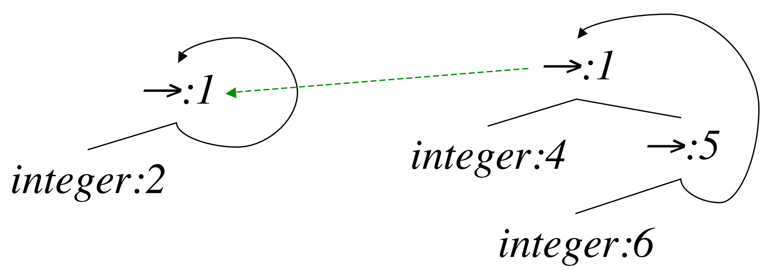





# Unify(1,3)

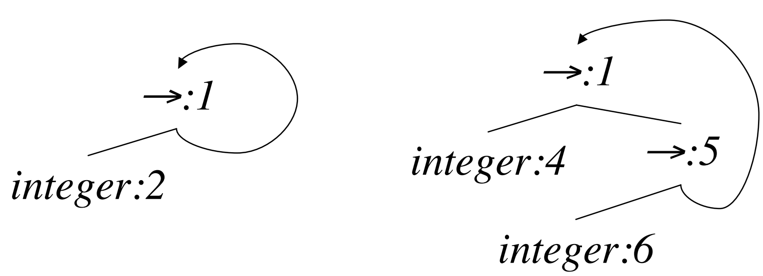


# Unify(1,3)

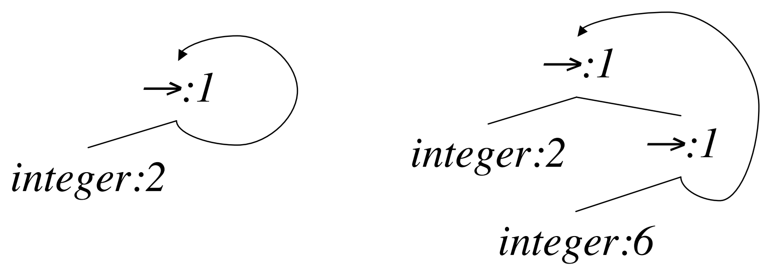





# Unify(2,4) *and* Unify(1,5)

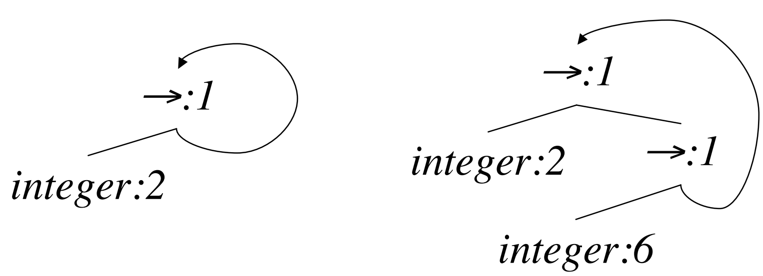


# Unify(2,4) *and* Unify(1,5)

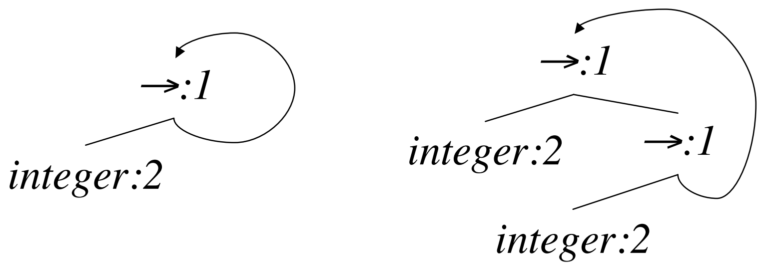





# Unify(2,6) *and* Unify(1,1)



# Unify(2,6) *and* Unify(1,1)






# Summary

- Semantic analysis: checking various well-formedness conditions
- Most common semantic conditions involve types of variables
- Symbol tables
- Discovering types for variables and functions using inference (unification)