Semantic Processing

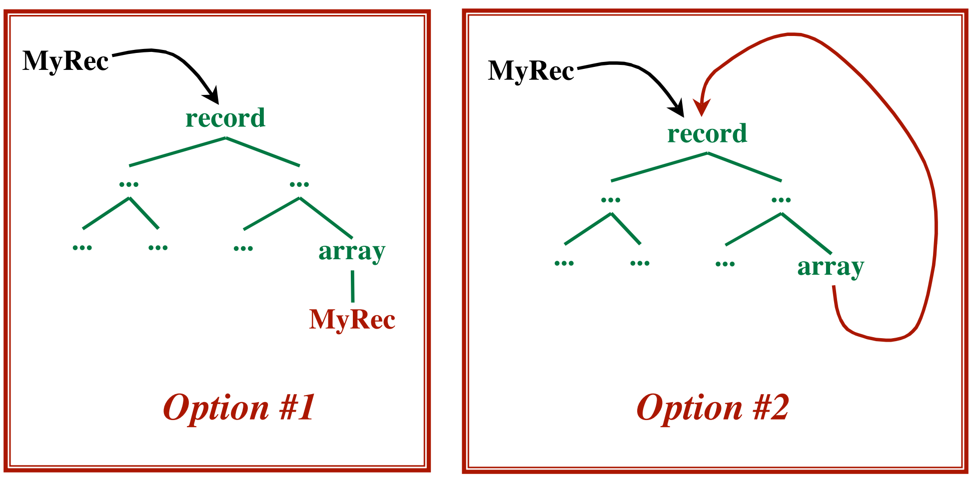
(Part 2)

All Projects Due: Friday 12-2-05, Noon

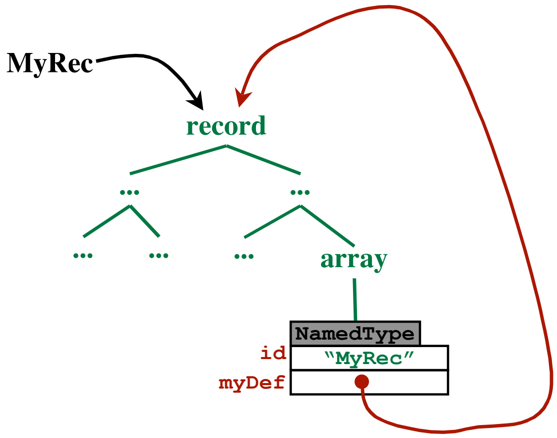
Final: Monday, December 5, 2005, 10:15-12:05
Comprehensive

Recursive Type Definitions

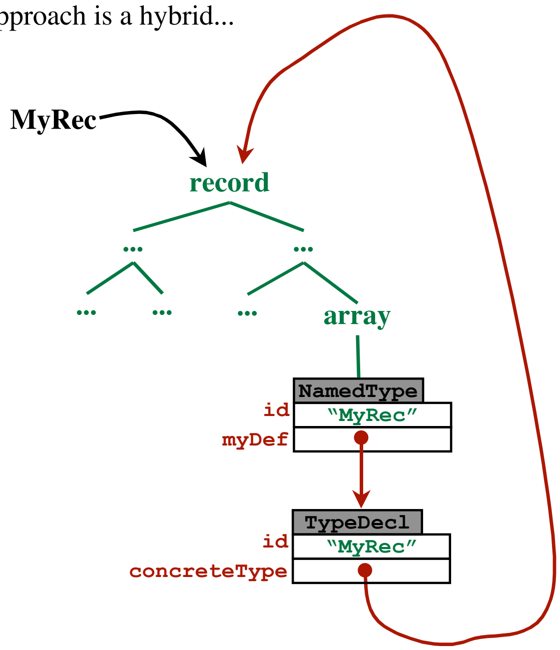
```
type MyRec is record
                f1: integer;
                f2: array of MyRec;
              end;
```



Our approach is a hybrid...



Our approach is a hybrid...



Testing Type Equivalence

Name Equivalence

- Stop when you get to a defined name
- Are the definitions the same (==)?

Structural Equivalence

- Test whether the type trees have the same shape.
- Graphs may contain cycles!
  The previous algorithm ("typeEquiv") will inifinite loop.
- Need an algorithm for testing "*Graph Isomorphism*"

PCAT

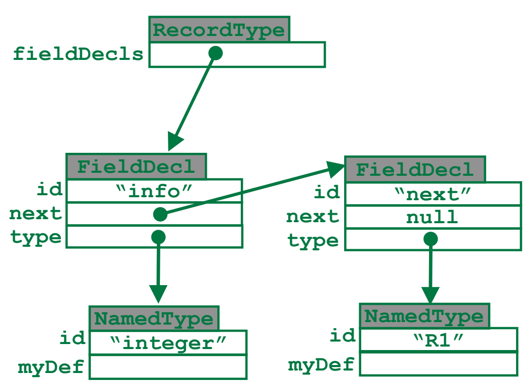
Recursion can occur in arrays and records.

```
type R is record
            info: integer;
            next: R;
          end;
type A is array of A;
```

PCAT uses Name Equivalence

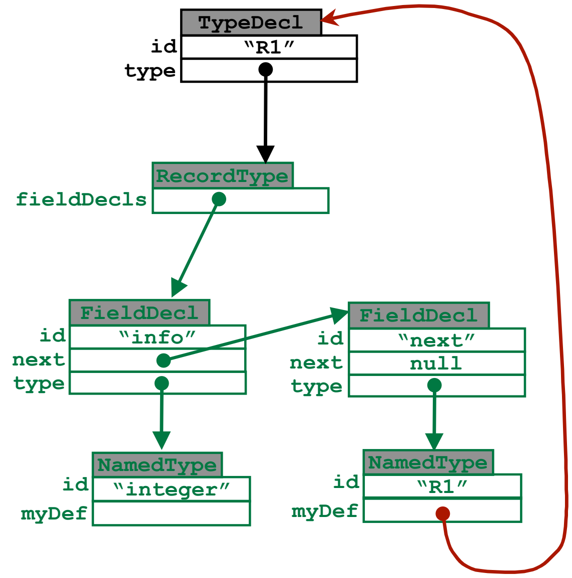
Representing Recursive Types in PCAT

```
type R1 is record
             info: integer;
             next: R1;
           end;
```



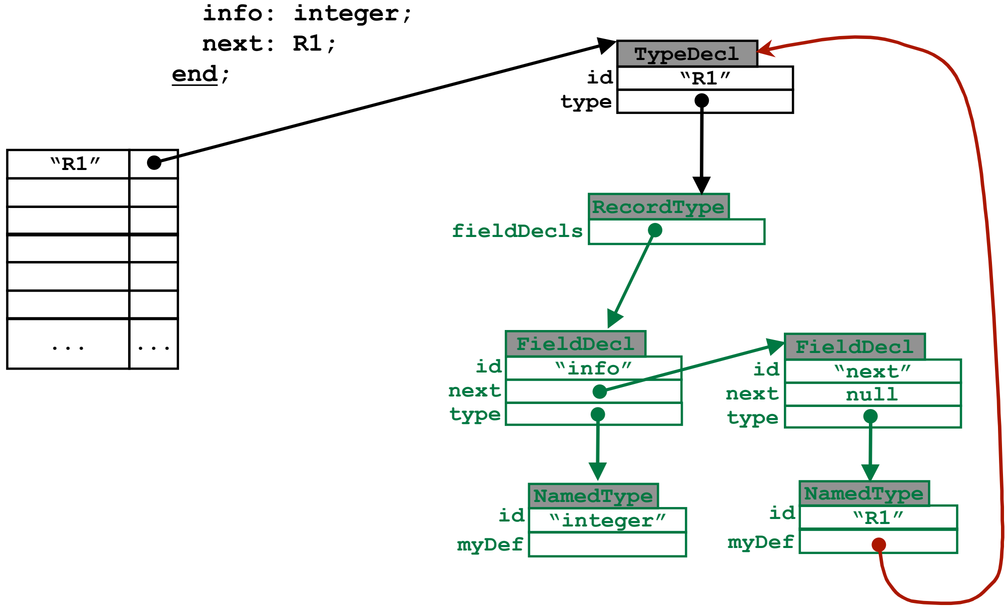
Representing Recursive Types in PCAT

```
type R1 is record
          info: integer;
          next: R1;
        end;
```



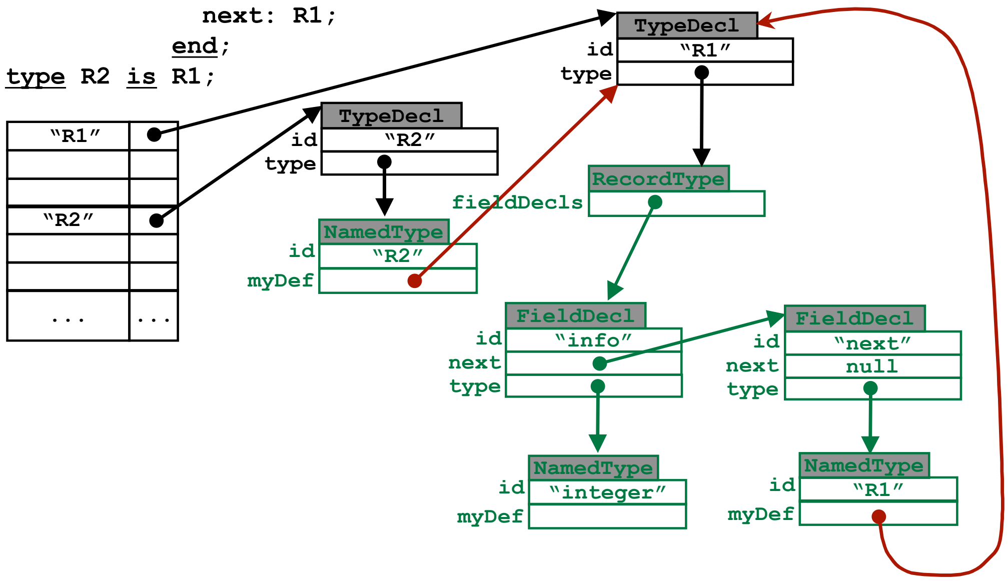
Representing Recursive Types in PCAT

```
type R1 is record
          info: integer;
          next: R1;
        end;
```



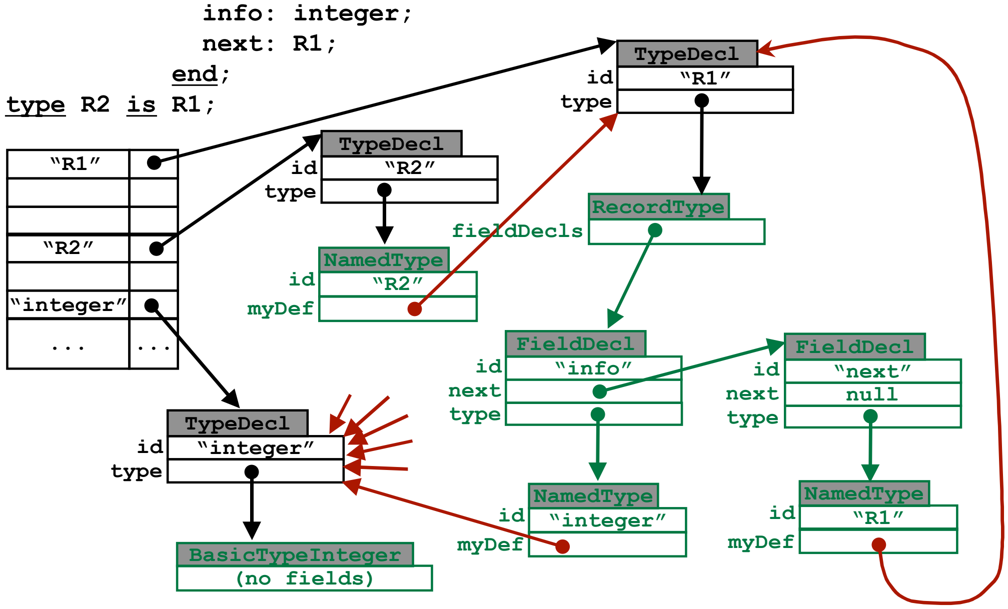
Representing Recursive Types in PCAT

```
type R1 is record
            info: integer;
            next: R1;
         end;
type R2 is R1;
```



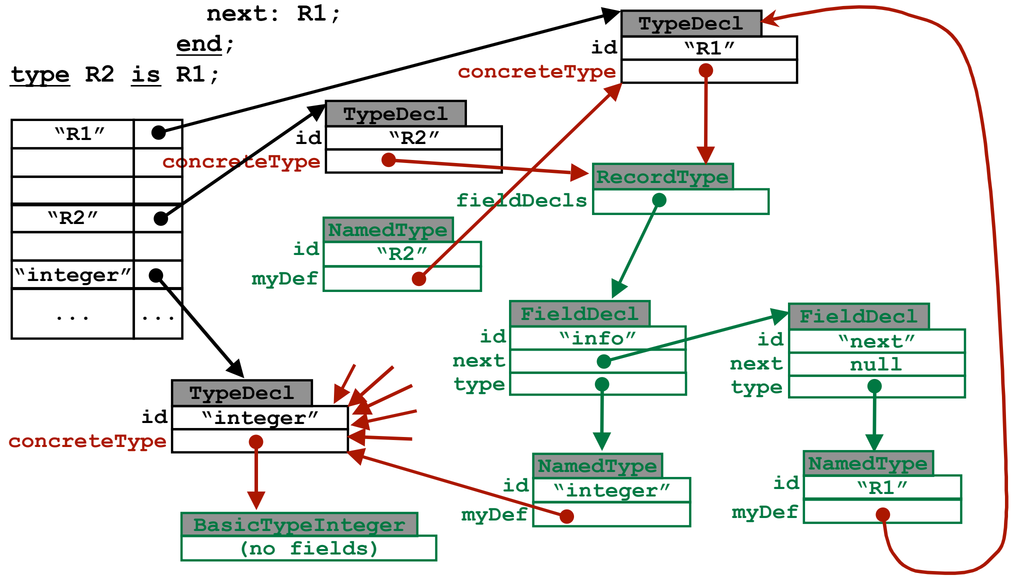
Representing Recursive Types in PCAT

```
type R1 is record
            info: integer;
            next: R1;
         end;
type R2 is R1;
```



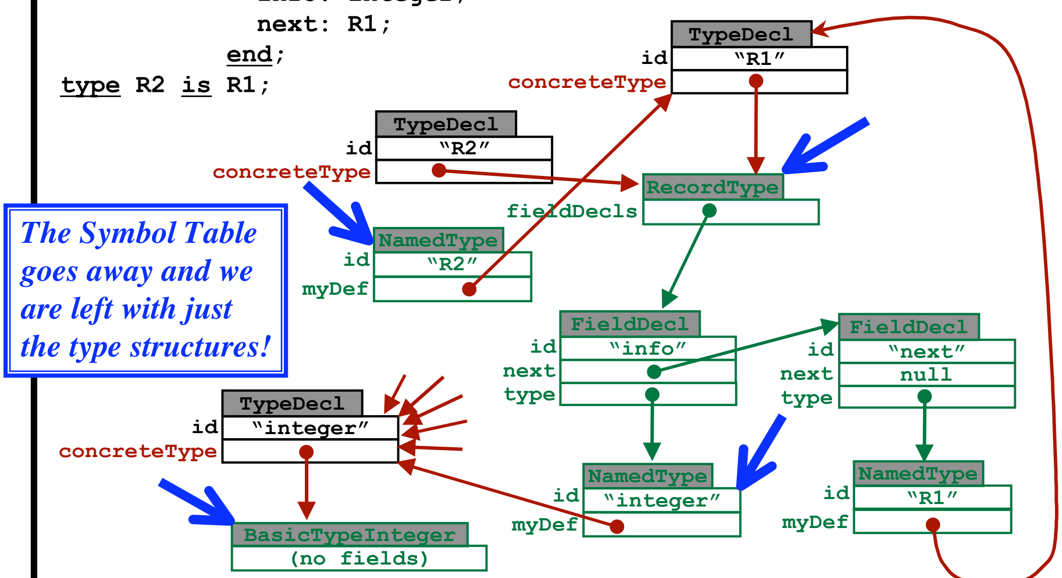
Representing Recursive Types in PCAT

```
type R1 is record
          info: integer;
          next: R1;
        end;
type R2 is R1;
```



Representing Recursive Types in PCAT

```
type R1 is record
          info: integer;
          next: R1;
        end;
type R2 is R1;
```



Type Conversions

```
var r: real;
    i: integer;
  ... r + i ...
```

During Type-checking...
- **Compiler discovers the problem**
- **Must insert "conversion" code**

Case 1:

No extra code needed.

```
i = p;      // e.g., pointer to integer conversion.
```

Case 2:

One (or a few) machine instructions

```
r = i;      // e.g., integer to real conversion.
```

Case 3:

Will need to call an external routine

```
System.out.print ("i=" + i);      // int to string
```

Perhaps written in the source language (an "*upcall*")

One compiler may use all 3 techniques.

Explicit Type Conversions

Example (Java):

```
i = r;
```

Type Error

Programmer must insert something to say "This is okay":

```
i = (int) r;
```

Language Design Approaches:

"C" casting notation

```
i = (int) r;
```

Function call notation

```
i = realToInt (r);
```

Keyword

```
i = realToInt r;
```

I like this:
- ***No additional syntax***
- ***Fits easily with other user-coded data transformations***

Compiler may insert:
- nothing
- machine instructions
- an upcall

Implicit Type Conversions ("Coercions")

Example (Java, PCAT):

```
r = i;
```

Compiler determines when a coercion must be inserted.
Rules can be complex.... Ugh!
Source of subtle errors.

My preference:
Minimize implicit coercions
Require explicit conversions

Java Philosophy:

Implicit coercions are okay
when no loss of numerical accuracy.

byte → short → int → long → float → double

Compiler may insert:
- nothing
- machine instructions
- an upcall

"Overloading" Functions and Operators

What does "+" mean?

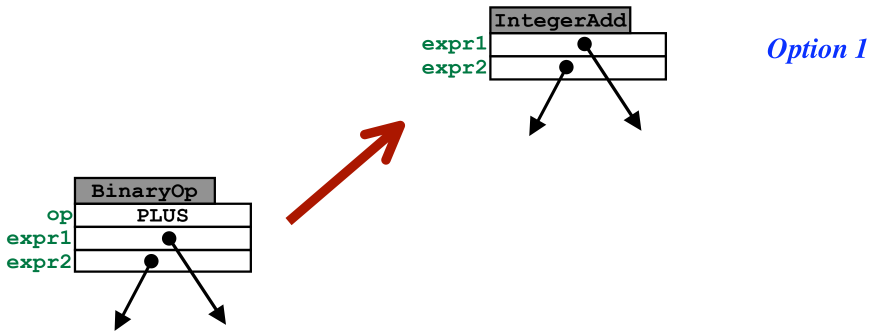
- integer addition
  16-bit? 32-bit?
- floating-point addition
  Single precision? Double precision?
- string concatenation
- user-defined meanings
  e.g., complex-number addition

Compiler must "resolve" the meaning of the symbols

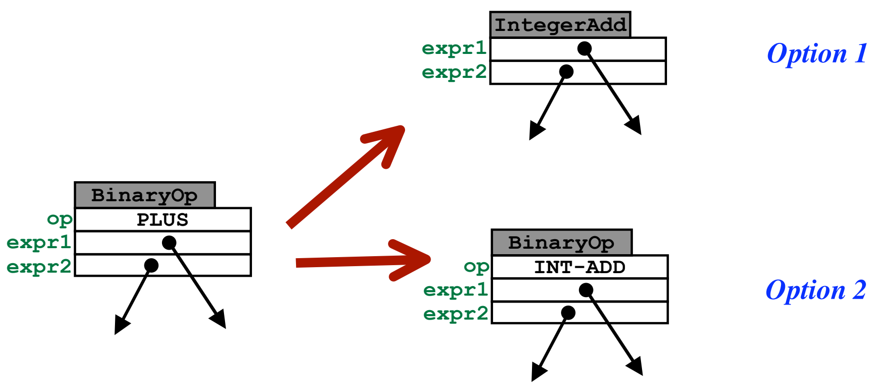
Will determine the operator from types of arguments

`i+i` → integer addition
`d+i` → floating-point addition (and double-to-int coercion)
`s+i` → string concatenation (and int-to-string coercion)

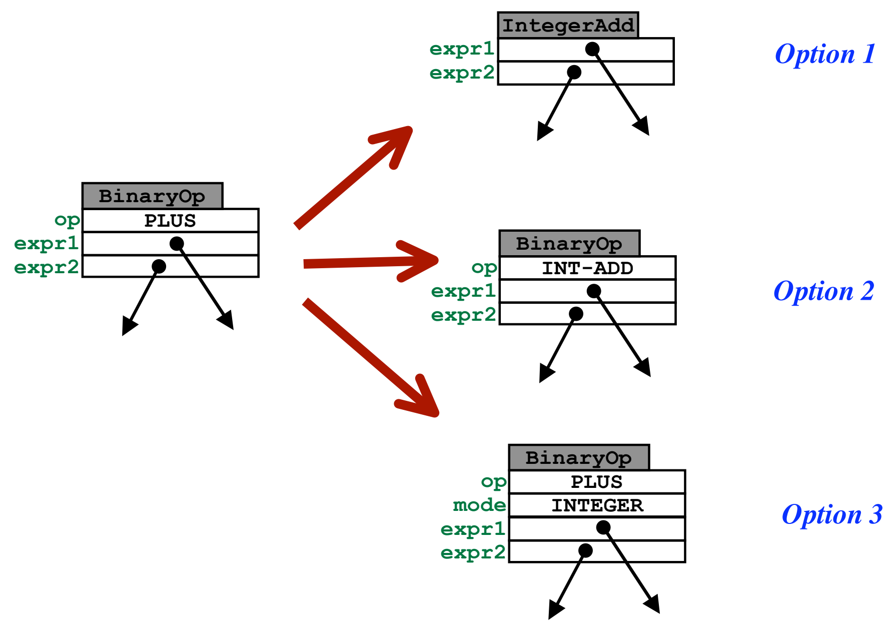
AST Design Options



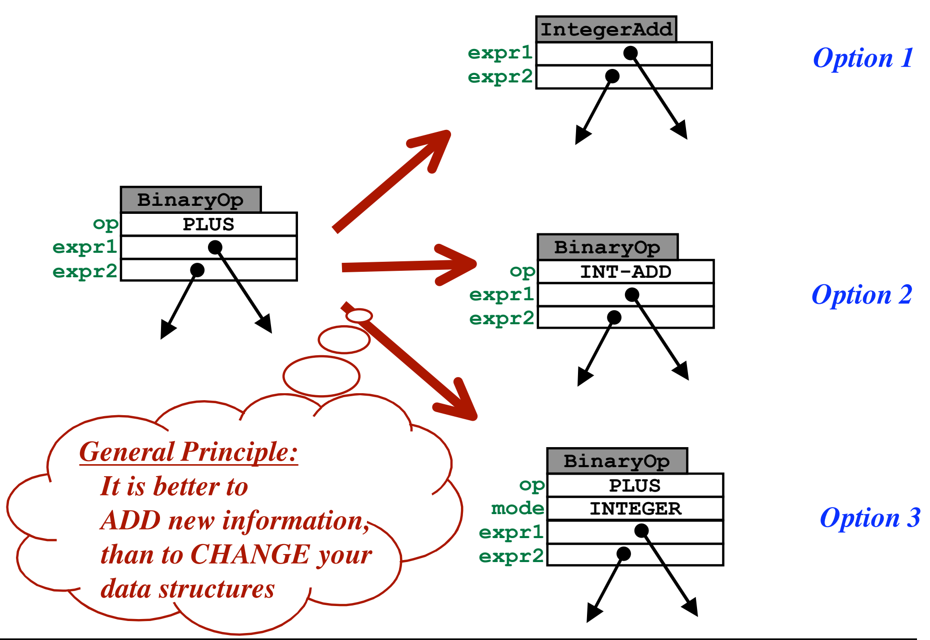
AST Design Options



AST Design Options



AST Design Options



Working with Functions

Want to say:

```
var f: int → real := ... ;
...
x := f(i);
```

Operators Syntax

E → E + E
→ E * E
→ E • E
→ ...

The "application" operator

Sometimes adjacency is used for function application

```
3N     ≡  3 * N
foo N  ≡  foo • N
```

Parsing Issues?

E → E E

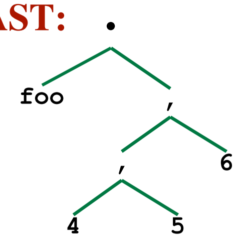
The programmer can always add parentheses:

```
foo 3 ≡ foo (3) ≡ (foo) 3
```

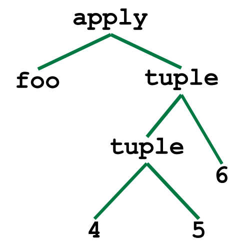
If the language also has tuples...

```
foo(4,5,6) ≡ (foo)(4,5,6)
```

AST:



AST:



Type Checking for Function Application

Syntax:

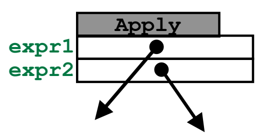
E → E • E

or:

E → E E

or:

E → E (E)



Type-Checking Code (e.g., in "checkApply")...

```
t1 = type of expr1;
t2 = type of expr2;
if t1 has the form "tDOMAIN → tRANGE" then
  if typeEquals(t2,tDOMAIN) then
    resultType = tRANGE;
  else
    error;
  endIf
else
  error
endIf
```

Curried Functions

Traditional ADD operator:

```
add: int × int → int
... add(3,4) ...
```

Curried ADD operator:

```
add: int → int → int
... add 3 4 ...
```

Recall: function application is Right-Associative

`≡ int → (int → int)`

Each argument is supplied individually, one at a time.

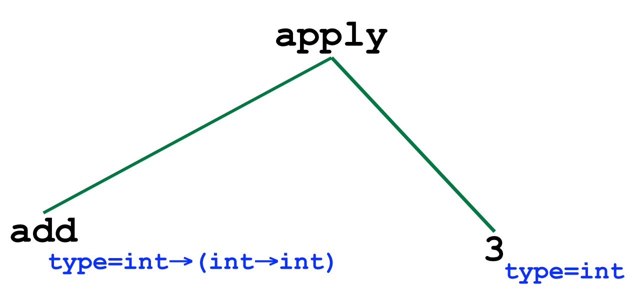
```
add 3 4 ≡ (add 3) 4
```

Can also say:

```
f: int → int
f = add 3;
... f 4 ...
```

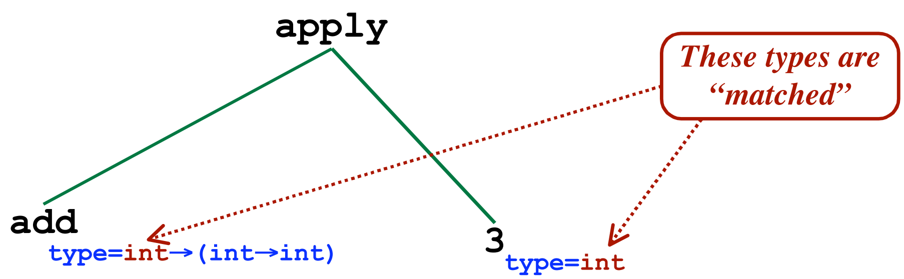

Type Checking "apply"

"type" is a synthesized attribute



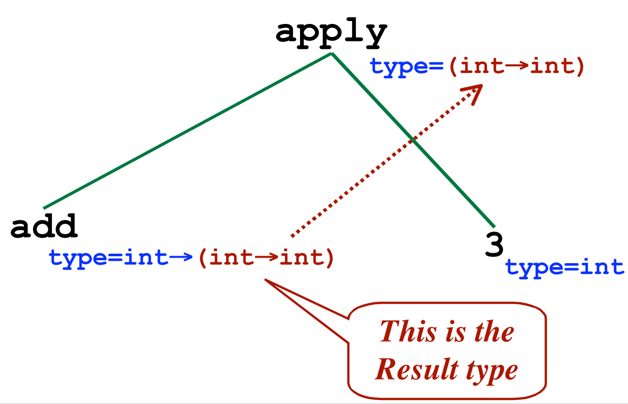
Type Checking “apply”

“type” is a synthesized attribute



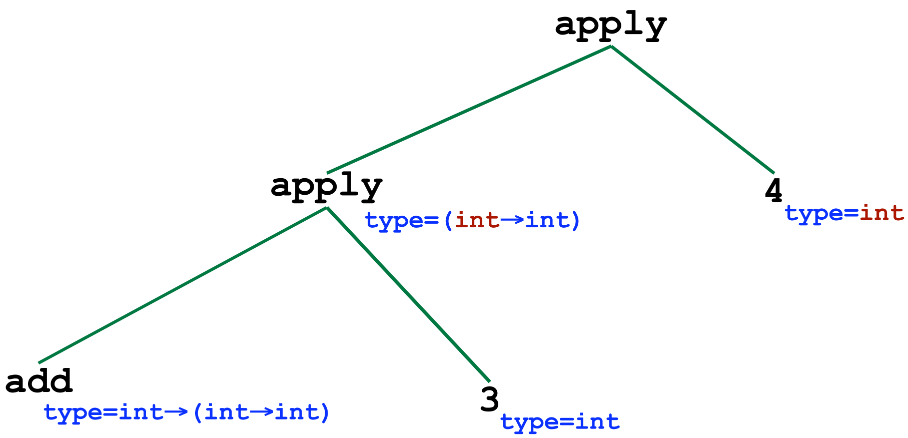
Type Checking “apply”

“type” is a synthesized attribute



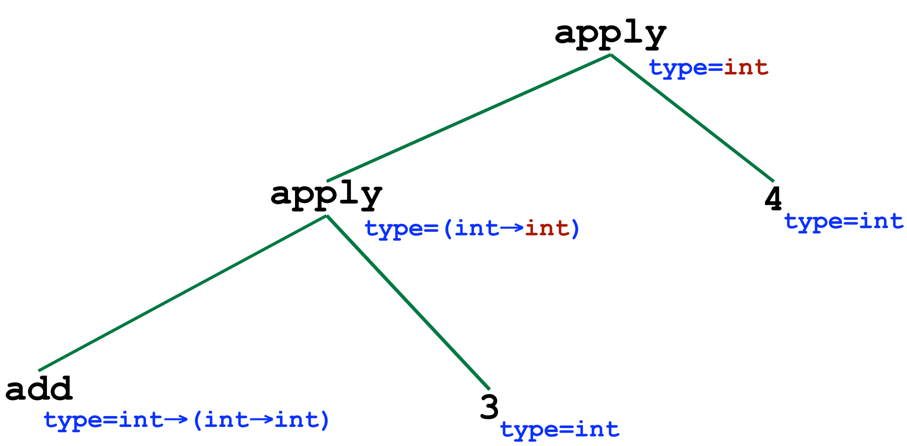
Type Checking "apply"

"type" is a synthesized attribute



Type Checking "apply"

"type" is a synthesized attribute



A Data Structure Example

Goal: *Write a function that finds the length of a list.*

```
type MyRec is record
                info: integer;
                next: MyRec;
              end;

procedure length (p:MyRec) : integer is
  var len: integer := 0;
  begin
    while (p <> nil) do
      len := len + 1;
      p := p.next;
    end;
    return len;
  end;
```

Traditional Languages: Each parameter must have a single, unique type.

A Data Structure Example

Goal: *Write a function that finds the length of a list.*

```
type MyRec is record
                info: integer;
                next: MyRec;
              end;

procedure length (p:MyRec) : integer is
  var len: integer := 0;
  begin
    while (p <> nil) do
      len := len + 1;
      p := p.next;
    end;
    return len;
  end;
```

Traditional Languages: Each parameter must have a single, unique type.

Problem: **Must write a new "length" function for every record type!!!**

... Even though we didn't access the fields particular to MyRec

Another Example: The “find” Function

Passed: • A list of T’s
• A function “test”, which has type T→boolean

Returns: • A list of all elements that passed the “test”
i.e., a list of all elements x, for which test(x) is true

```
procedure find (inList: array of T;
                test:   T→boolean)    : array of T is
  var result: array of T;
      i, j: integer := 1;
  begin
    result := ... new array ...;
    while i < sizeof(inList) do
      if test(inList[i]) then
        result[j] := inList[i];
        j := j + 1;
      endIf;
      i := i + 1;
    endWhile;
  return result;
end;
```


This function should work for any type T.

Goal: Write the function once and re-use.

This problem is typical...
- Data Structure Manipulation

Want to re-use code...
- Hash Table Lookup Algorithms
- Sorting Algorithms
- B-Tree Algorithms

etc.

...Regardless of the type of data being mainpulated.

The "ML" Version of "Length"

Background:

Data Types:

```
Int
Bool
List(...)
```

Lists:

```
[1,3,5,7,9]
[]
[[1,2], [5,4,3], [], [6]]
```

Type is: `List(Int)`

Type is: `List(List(Int))`

Operations on Lists:

```
head
   head([5,4,3]) ⇒ 5
   head: List(T)→T
tail
   tail([5,4,3]) ⇒ [4,3]
   tail: List(T)→List(T)
null
   null([5,4,3]) ⇒ false
   null: List(T)→Bool
```

Notation:
`x:T`
means: *"The type of x is T"*

Semantics - Part 2

The "ML" Version of "Length"

Operations on Integers:

```
+
   5 + 7 ≡ +(5,7) ⇒ 12
   +: Int×Int→Int
```

Constants:

```
0: Int
1: Int
2: Int
...
```

"Constant" Function:
`Int ≡ →Int`
(A function of zero arguments)

```
fun length (x) = if null(x)
                 then 0
                 else length(tail(x))+1
```

New symbols introduced here:

```
x: List(α)
length: List(α)→Int
```

No types are specified explicitly! No Declarations!
ML infers the types from the way the symbols are used!!!

Predicate Logic Refresher

Logical Operators (AND, OR, NOT, IMPLIES)

`&`, `|`, `~`, →

Predicate Symbols

`P`, `Q`, `R`, ...

Function and Constant Symbols

`f`, `g`, `h`, ... `a`, `b`, `c`, ...

Variables

`x`, `y`, `z`, ...

Quantifiers

∀, ∃

WFF: Well-Formed Formulas

`∀x.    ~P(f(x))  & Q(x)   → Q(x)`

Precedence and Associativity:

(Quantifiers bind most loosely)

`∀x.(((~P(f(x)) & Q(x)) → Q(x))`

A grammar of Predicate Logic Expressions? Sure!

Semantics - Part 2

Type Expressions

Basic Types

`Int`, `Bool`, etc.

Constructed Types

→, ×, `List()`, `Array()`, `Pointer()`, etc.

Type Expressions

`List(Int × Int) → List(Int → Bool)`

Type Variables

α, β, γ, α_1, α_2, α_3, ...

Universal Quantification: ∀

`∀α . List(α)→List(α)`

(Won't use existential quantifier, ∃)

Remember: ∀ binds loosely

`∀α .(List(α)→List(α))`

"For any type α, a function that maps lists of α's to lists of α's."

Type Expressions

Okay to change variables (as long as you do it consistently)...

```
   ∀α . Pointer(α)→Boolean
≡ ∀β . Pointer(β)→Boolean
```

What do we mean by that?

Same as for predicate logic...

- Can't change α to a variable name already in use elsewhere
- Must change all occurrences of α to the same variable

We will use only universal quantification ("for all", ∀)

Will not use ∃

Okay to just drop the ∀ quantifiers.

```
   ∀α . ∀β . (List(α) × (α → β)) → List(β)
≡           (List(α) × (α → β)) → List(β)
≡           (List(β) × (β → γ)) → List(γ)
```


Practice

Given:

```
x: Int
y: Int→Boolean
```

What is the type of `(x,y)`*?*

Practice

Given:

```
x: Int
y: Int→Boolean
```

What is the type of `(x,y)`*?*

```
(x,y): Int × (Int→Boolean)
```


Practice

Given:

```
x: Int
y: Int→Boolean
```

What is the type of `(x,y)`*?*

```
(x,y): Int × (Int→Boolean)
```

Given:

```
f: List(α)→List(α)
z: List(Int)
```

What is the type of `f(z)`*?*

Practice

Given:

```
x: Int
y: Int→Boolean
```

What is the type of `(x,y)`*?*

```
(x,y): Int × (Int→Boolean)
```

Given:

```
f: List(α)→List(α)
z: List(Int)
```

What is the type of `f(z)`*?*

```
f(z): List(Int)
```

Practice

Given:

```
x: Int
y: Int→Boolean
```

What is the type of `(x,y)`*?*

```
(x,y): Int × (Int→Boolean)
```

Given:

```
f: List(α)→List(α)
z: List(Int)
```

What is the type of `f(z)`*?*

```
f(z): List(Int)
```

What is going on here?

We "matched" α to `Int`

We used a "***Substitution***"

```
α = Int
```

What do we mean by "matched"???

Practice

Given:

```
x: Int
y: Int→Boolean
```

What is the type of `(x,y)` *?*

```
(x,y): Int × (Int→Boolean)
```

Given:

```
f: List(α)→List(α)
z: List(Int)
```

What is the type of `f(z)` *?*

```
f(z): List(Int)
```

What is going on here?

We "matched" α to `Int`

We used a "***Substitution***"

α = Int

What do we mean by "matched"???

UNIFICATION!

Unification

Given: Two [type] expressions

Goal: Try to make them equal

Using: Consistent substitutions for any [type] variables in them

Result:

- Success
  plus the variable substitution that was used
- Failure

A Language With Polymorphic Functions

```
P  → D ; E
D  → D ; D
   → id : Q
Q  → ∀ id . Q
   → T
T  → T "→" T
   → T × T
   → List ( T )
   → Int
   → Bool
   → id
   → ( T )
E  → id
   → int
   → E E
   → ( E , E )
   → ( E )
```

- Quantified Type Expressions (Q → ∀ id . Q, Q → T)
- Unquantified Type Expressions (T productions)
- Type Variables (T → id)
- Grouping (T → (T))
- Function Apply (E → E E)
- Tuple Construction (E → (E , E))
- Grouping (E → (E))

Semantics - Part 2

A Language With Polymorphic Functions

```
P  → D ; E
D  → D ; D
   → id : Q
Q  → ∀ id . Q
   → T
T  → T "→" T
   → T × T
   → List ( T )
   → Int
   → Bool
   → id
   → ( T )
E  → id
   → int
   → E E
   → ( E , E )
   → ( E )
```

Examples of Expressions:

```
123
(x)
foo(x)
find(test,myList)
add(3,4)
```

A Language With Polymorphic Functions

P → D ; E
D → D ; D
  → **id** : Q
Q → ∀ **id** . Q
  → T
T → T "→" T
  → T × T
  → `List` (T)
  → `Int`
  → `Bool`
  → **id**
  → (T)
E → **id**
  → **int**
  → E E
  → (E , E)
  → (E)

Examples of Types:

```
Int → Bool
Bool × (Int → Bool)
α
α × (α → Bool)
(((β→Bool) × List(β))→List(β))
```

A Type Variable (id)

A Language With Polymorphic Functions

P → D ; E
D → D ; D
  → **id** : Q
Q → ∀ **id** . Q
  → T
T → T "→" T
  → T × T
  → `List` (T)
  → `Int`
  → `Bool`
  → **id**
  → (T)
E → **id**
  → **int**
  → E E
  → (E , E)
  → (E)

Examples of Quatified Types:

```
Int → Bool
∀ α . (α → Bool)
∀ β .(((β→Bool) × List(β))→List(β))
```

A Language With Polymorphic Functions

```
P  → D ; E
D  → D ; D
   → id : Q
Q  → ∀ id . Q
   → T
T  → T "→" T
   → T × T
   → List ( T )
   → Int
   → Bool
   → id
   → ( T )
E  → id
   → int
   → E E
   → ( E , E )
   → ( E )
```

Examples of Declarations:

```
i: Int;
myList: List(Int);
test: ∀ α . (α → Bool);
find: ∀ β .((β→Bool) × List(β))→List(β))
```


A Language With Polymorphic Functions

```
P  → D ; E
D  → D ; D
   → id : Q
Q  → ∀ id . Q
   → T
T  → T "→" T
   → T × T
   → List ( T )
   → Int
   → Bool
   → id
   → ( T )
E  → id
   → int
   → E E
   → ( E , E )
   → ( E )
```

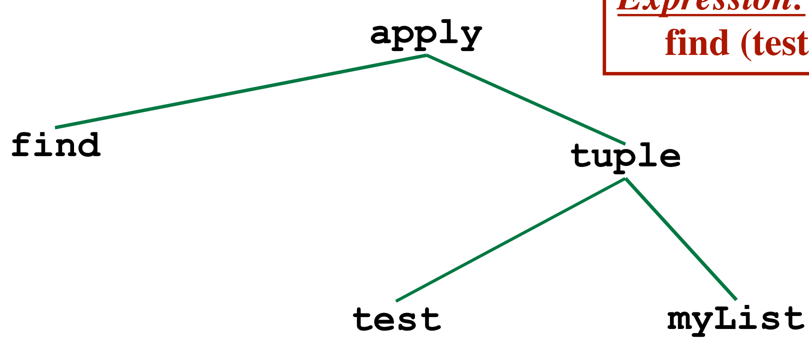
An Example Program:

```
myList: List(Int);
test: ∀ α . (α → Bool);
find: ∀ β .((β→Bool) × List(β))→List(β));
find (test, myList)
```

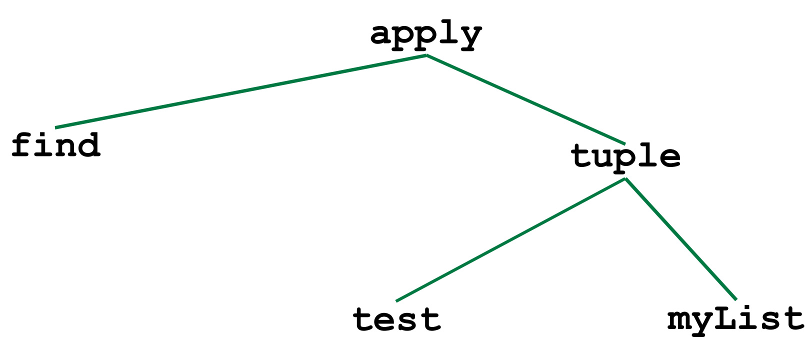
GOAL:
Type-check this expression given these typings!

Parse Tree (Annotated with Synthesized Types)

Expression:
find (test, myList)



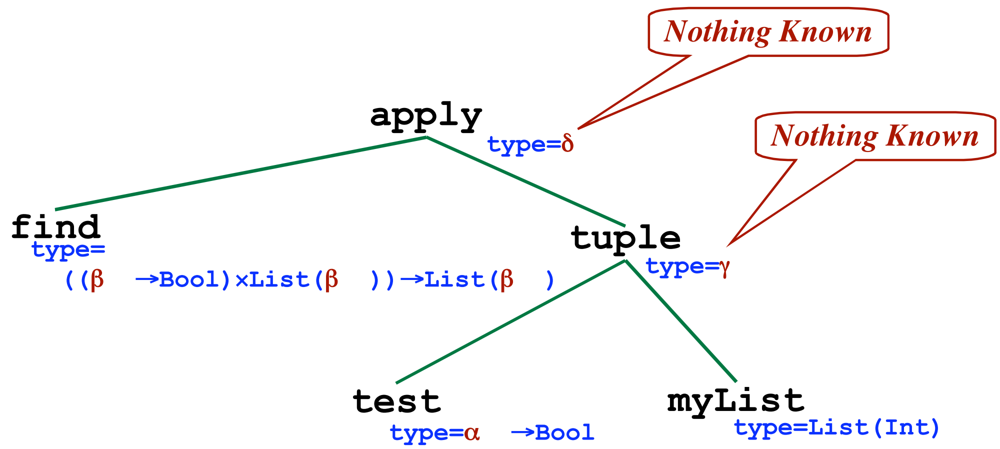
Parse Tree (Annotated with Synthesized Types)



Add known typing info:

```
myList: List(Int);
test: ∀ α .(α → Bool);
find: ∀ β .(((β→Bool) × List(β))→List(β));
```

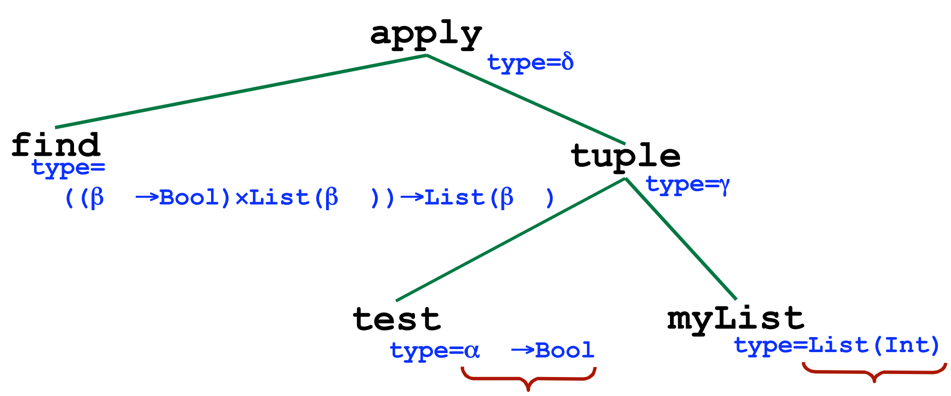
Parse Tree (Annotated with Synthesized Types)



Add known typing info:

```
myList: List(Int);
test: ∀ α .(α → Bool);
find: ∀ β .(((β→Bool) × List(β))→List(β));
```

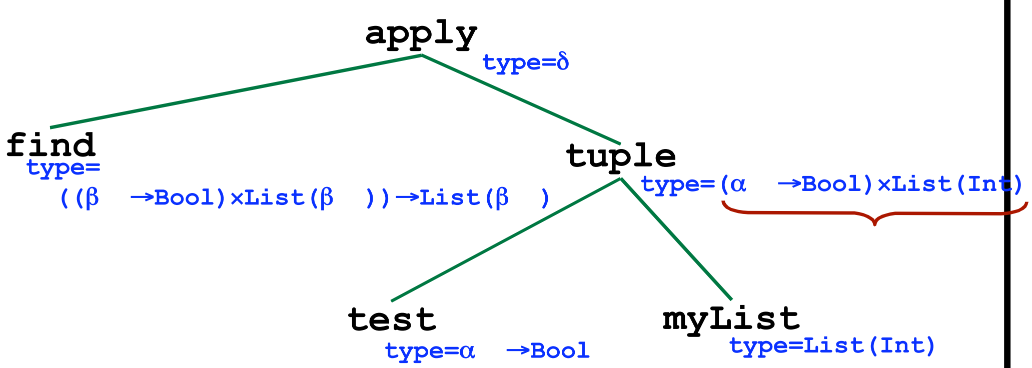

Parse Tree (Annotated with Synthesized Types)



Tuple Node:

Match γ to (α→Bool) × List(Int)

Parse Tree (Annotated with Synthesized Types)



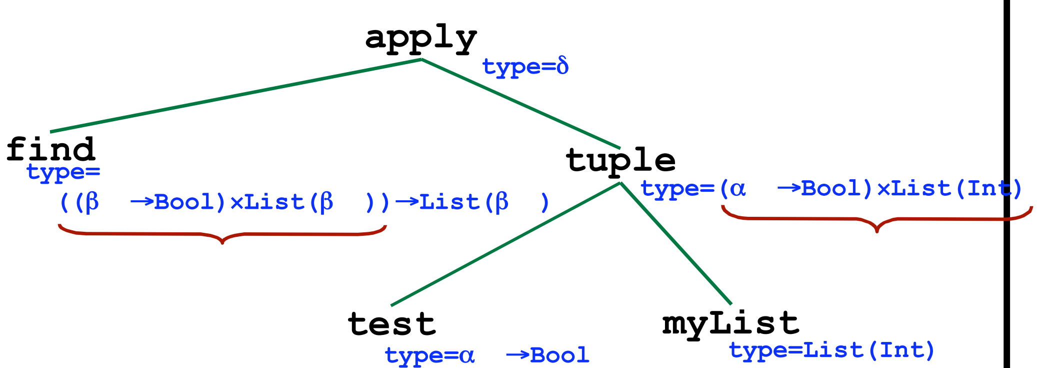
Tuple Node:

Match γ to (α→Bool) × List(Int)

Conclude:

γ = (α→Bool) × List(Int)

Parse Tree (Annotated with Synthesized Types)



Apply Node:

Match

(β →Bool) × List(β)

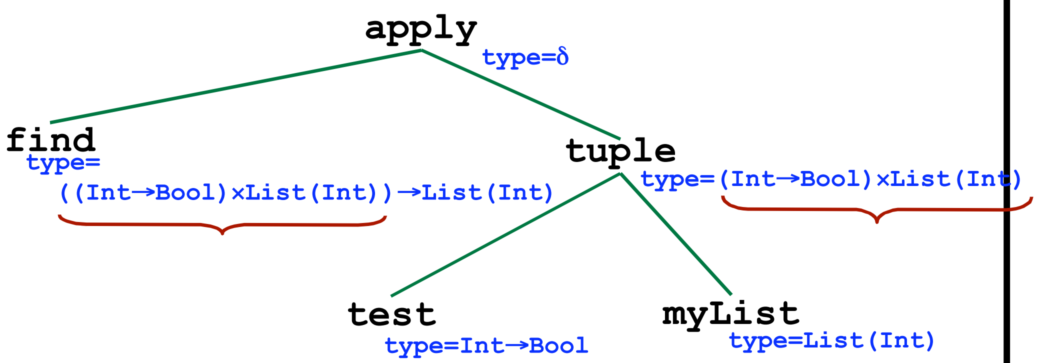
(α →Bool) × List(Int)

Conclude:

β = Int

α = β = Int

Parse Tree (Annotated with Synthesized Types)



Apply Node:

Match

(β →Bool) × List(β)

(α →Bool) × List(Int)

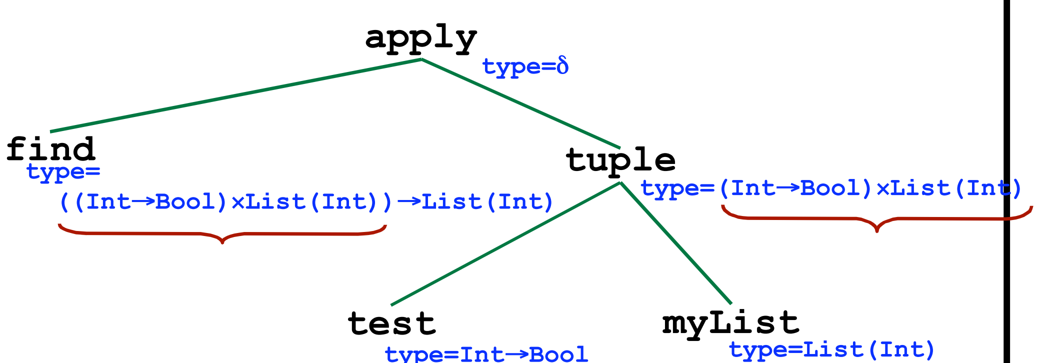
Conclude:

β = Int

α = β = Int

Semantics - Part 2

Parse Tree (Annotated with Synthesized Types)



Apply Node:

Match

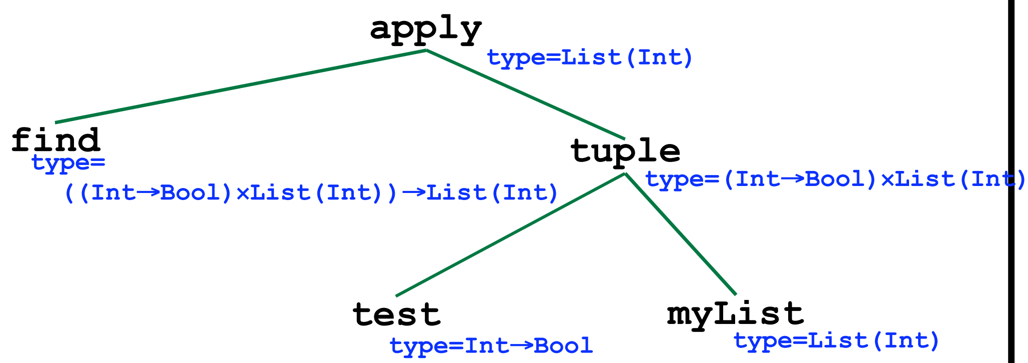
List(Int)

δ

Conclude:

δ = List(Int)

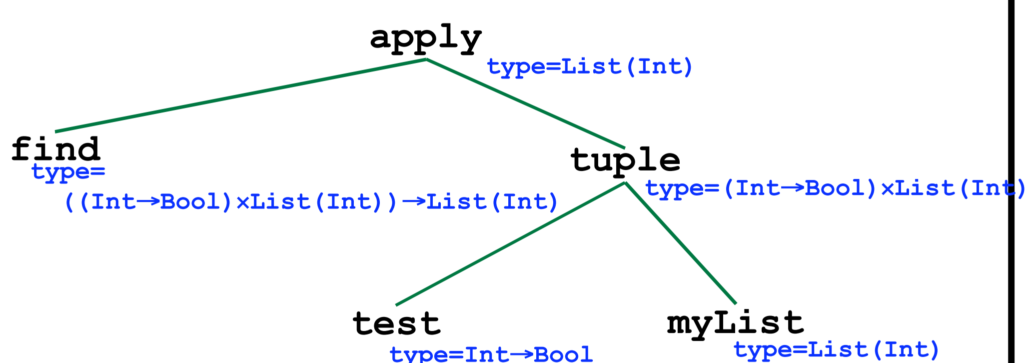
Parse Tree (Annotated with Synthesized Types)



Apply Node:
Match
List(Int)
δ

Conclude:
δ = List(Int)

Parse Tree (Annotated with Synthesized Types)



Results:
α = Int
β = Int
δ = List(Int)
γ = (Int→Bool) × List(Int)

Unification of Two Expressions

Example:

$t_1 = \alpha \times \text{Int}$

$t_2 = \text{List}(\beta) \times \gamma$

Is there a substitution that makes $t_1 = t_2$?

> "t_1 ***unifies with*** t_2"
> if and only if there is a substitution S such that
> $S(t_1) = S(t_2)$

Here is a substitution that makes $t_1 = t_2$:

$\alpha \leftarrow \text{List}(\beta)$

$\gamma \leftarrow \text{Int}$

Other notation for substitutions:

$\{\alpha/\text{List}(\beta), \gamma/\text{Int}\}$

Most General Unifier

There may be several substitutions.
Some are *more general* than others.

Example:

$t_1 = \alpha \times \text{Int}$

$t_2 = \text{List}(\beta) \times \gamma$

Unifying Substitution #1:

$\alpha \leftarrow \text{List}(\text{List}(\text{List}(\text{Bool})))$

$\beta \leftarrow \text{List}(\text{List}(\text{Bool}))$

$\gamma \leftarrow \text{Int}$

Unifying Substitution #2:

$\alpha \leftarrow \text{List}(\text{Bool} \times \delta)$

$\beta \leftarrow \text{Bool} \times \delta$

$\gamma \leftarrow \text{Int}$

Unifying Substitution #3:

$\alpha \leftarrow \text{List}(\beta)$

$\gamma \leftarrow \text{Int}$

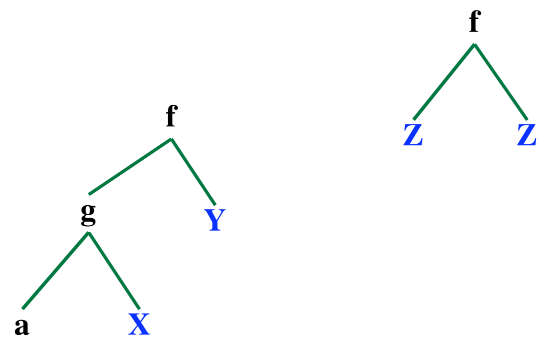
This is the "Most General Unifier"

Unifying Two Terms / Types

Unify these two terms:

f(g(a,X),Y)
f(Z,Z)

Unification makes the terms identical.



Unifying Two Terms / Types

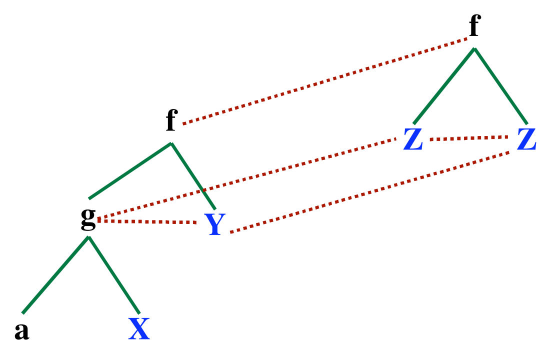
Unify these two terms:

f(g(a,X),Y)
f(Z,Z)

Unification makes the terms identical.

The substitution:

Y ← Z
Z ← g(a,X)



Unifying Two Terms / Types

Unify these two terms:

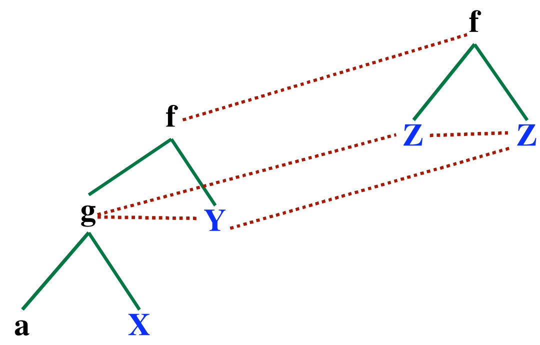
f(g(a,X),Y)
f(Z,Z)

Unification makes the terms identical.

The substitution:

Y ← Z
Z ← g(a,X)

Merge the trees into one!



Unifying Two Terms / Types

Unify these two terms:

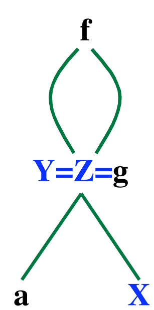
f(g(a,X),Y)
f(Z,Z) → **f(g(a,X),g(a,X))**

Unification makes the terms identical.

The substitution:

Y ← Z
Z ← g(a,X)

Merge the trees into one!



Unifying Two Terms / Types

Unify these two terms:

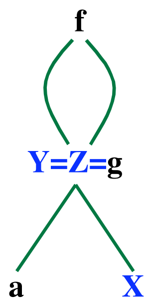
f(g(a,X),Y) → **f(g(a,X),g(a,X))**
f(Z,Z)

Unification makes the terms identical.

The substitution:

Y ← Z
Z ← g(a,X)

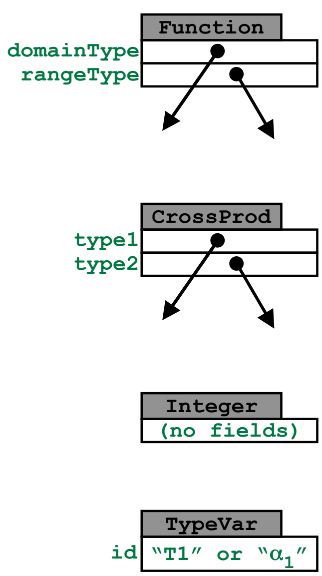
Merge the trees into one!



Same with unifying types!

(Int × List(X)) × Y
Z × Z

Representing Types With Trees

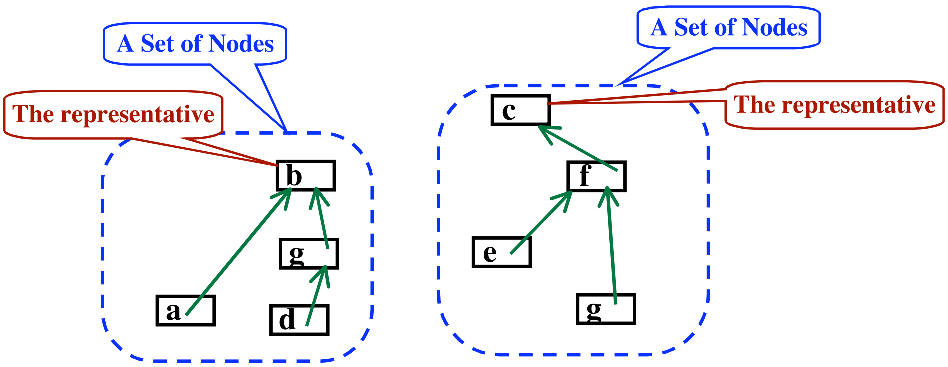


Same for other basic and constructed types
Real, Bool, List(T), etc.

Merging Sets

Approach: Will work with sets of nodes.
Each set will have a "representative" node.

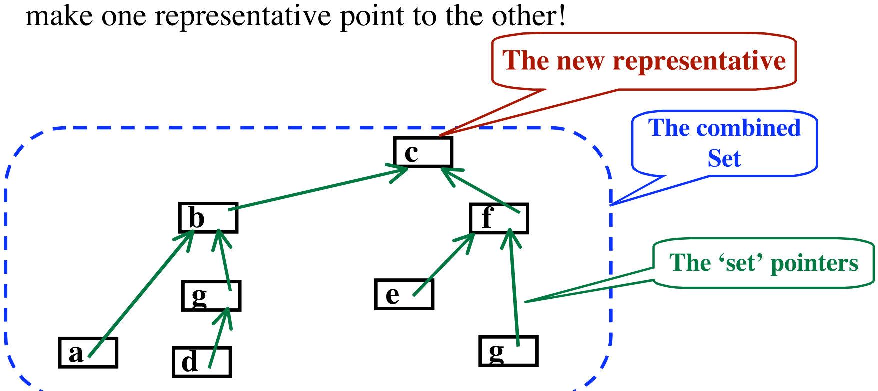
Goal: Merge two sets of nodes into a single set.
When two sets are merged (the "union" operation)...
make one representative point to the other!



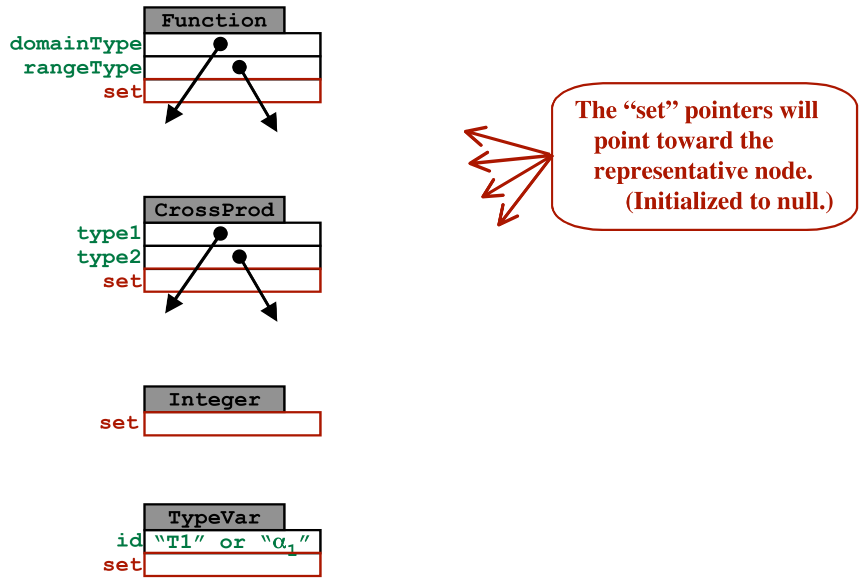
Merging Sets

Approach: Will work with sets of nodes.
Each set will have a "representative" node.

Goal: Merge two sets of nodes into a single set.
When two sets are merged (the "union" operation)...
make one representative point to the other!



Representing Type Expressions



Merging Sets

`Find(p) → ptr`

Given a pointer to a node, return a pointer to the representative of the set containing p.

Just chase the "set" pointers as far as possible.

`Union(p,q)`

Merge the set containing p with the set containing q.

Do this by making the representative of one of the sets point to the representative of the other set. If one representative is a variable node and the other is not, always use the non-variable node as the representative of the combined, merged sets. In other words, make the variable node point to the other node.

The Unification Algorithm

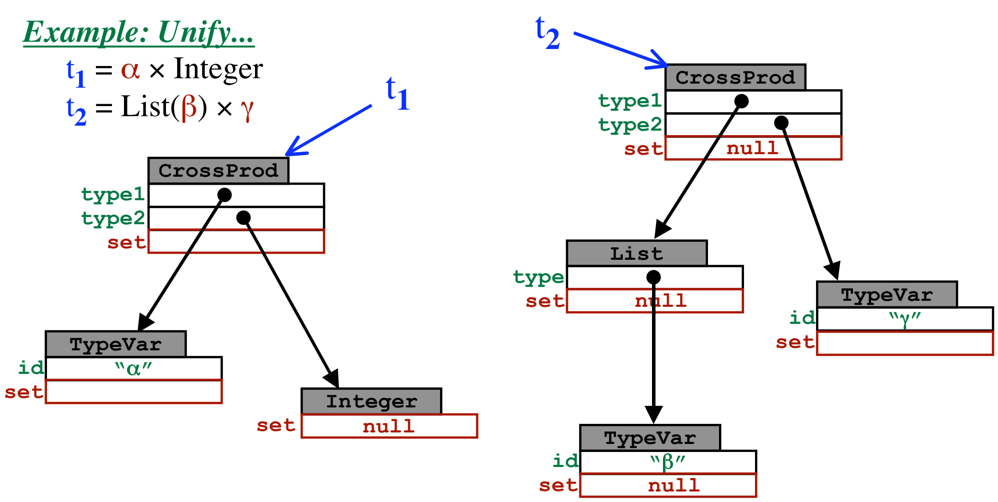
```
function Unify (s', t': Node) returns bool
  s = Find(s')
  t = Find(t')
  if s == t then
    return true
  elseIf s and t both point to INTEGER nodes then
    return true
  elseIf s or t points to a VARIABLE node then
    Union(s,t)
  elseif s points to a node FUNCTION(s1,s2) and
         t points to a node FUNCTION(t1,t2) then
    Union(s,t)
    return Unify(s1,t1) and Unify(s2,t2)
  elseif s points to a node CROSSPROD(s1,s2) and
         t points to a node CROSSPROD(t1,t2) then
    Union(s,t)
    return Unify(s1,t1) and Unify(s2,t2)
  elseIf ...
  else
    return false
  endIf
```

Etc., for other type constructors and basic type nodes

Example: Unify...

$t_1 = \alpha \times \text{Integer}$

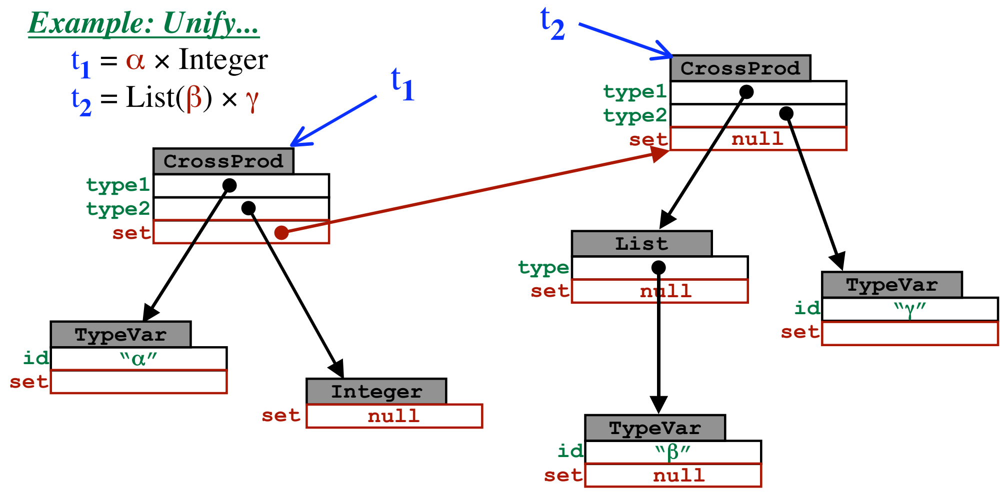
$t_2 = \text{List}(\beta) \times \gamma$



Example: Unify...

$t_1 = \alpha \times$ Integer

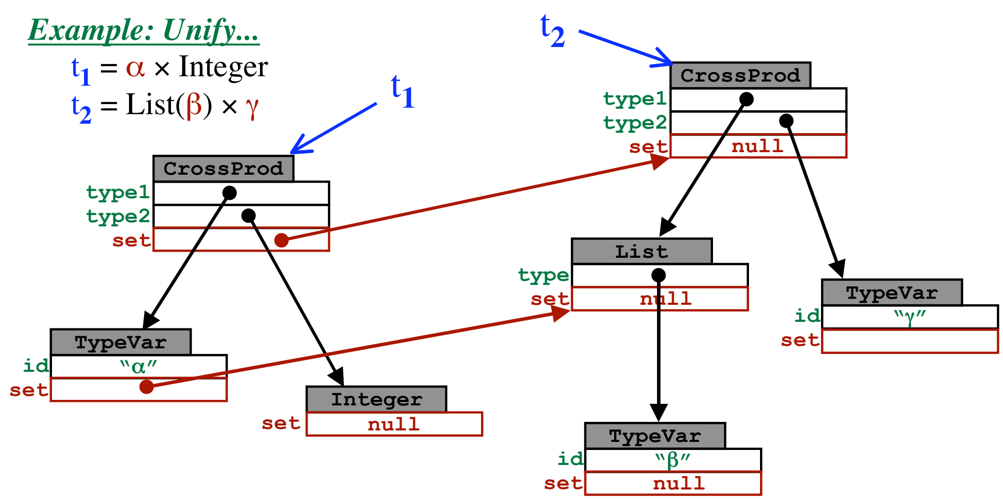
$t_2 =$ List(β) $\times\ \gamma$



Example: Unify...

$t_1 = \alpha \times$ Integer

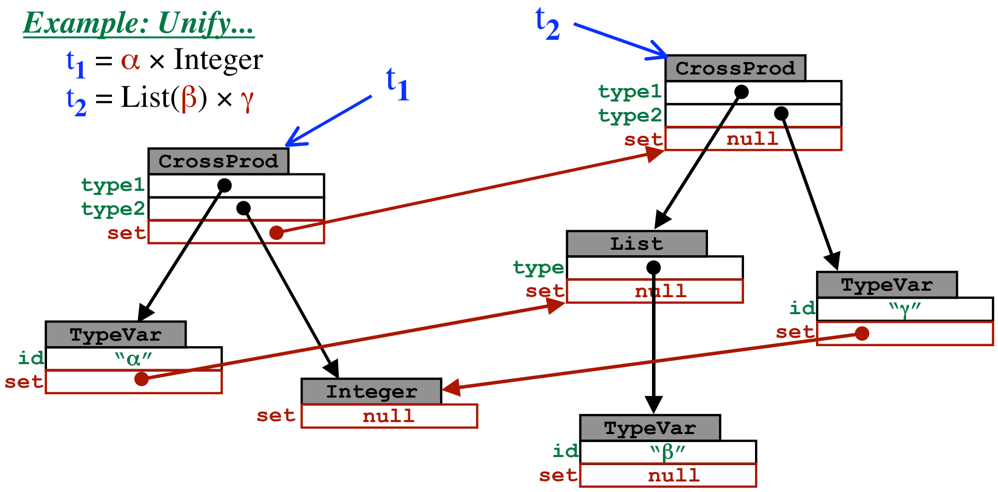
$t_2 =$ List(β) $\times\ \gamma$



Example: Unify...

$t_1 = \alpha \times$ Integer

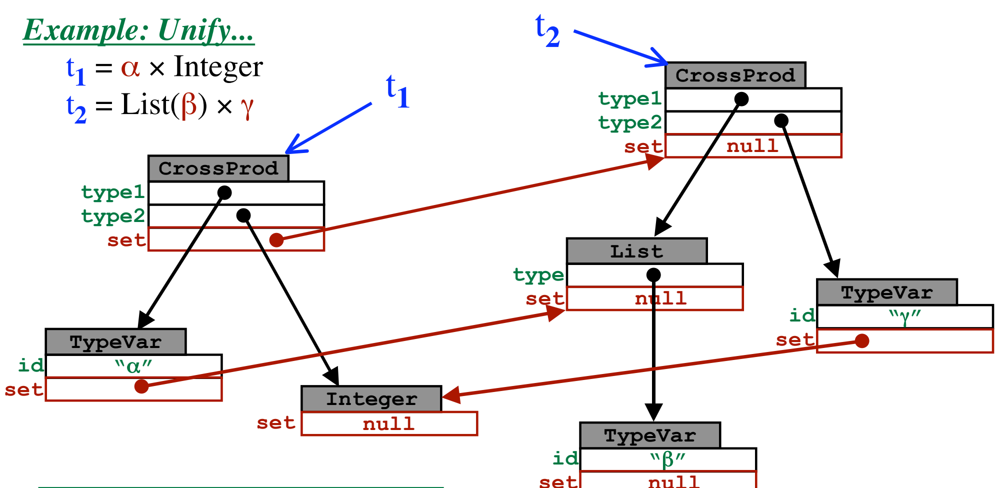
$t_2 =$ List(β) $\times \gamma$



Example: Unify...

$t_1 = \alpha \times$ Integer

$t_2 =$ List(β) $\times \gamma$



Recovering the Substitution:

$\alpha \leftarrow$ List(β)

$\gamma \leftarrow$ Integer

Type-Checking with an Attribute Grammar

`Lookup(string) → type`

Lookup a name in the symbol table and return its type.

`Fresh(type) → type`

Make a copy of the type tree.

Replace all variables (consistently) with new, never-seen-before variables.

`MakeIntNode() → type`

Make a new leaf node to represent the "Int" type

`MakeVarNode() → type`

Create a new variable node and return it.

`MakeFunctionNode(type1,type2) → type`

Create a new "Function" node and return it.

Fill in its domain and range types.

`MakeCrossNode(type1,type2) → type`

Create a new "Cross Product" node and return it.

Fill in the types of its components.

`Unify(type1,type2) → bool`

Unify the two type trees and return true if success.

Modify the type trees to perform the substitutions.

Type-Checking with an Attribute Grammar

$E \rightarrow$ **id**	`E.type = Fresh(Lookup(id.svalue));`
$E \rightarrow$ **int**	`E.type = MakeIntNode();`

$E_0 \rightarrow E_1\ E_2$	`p = MakeVarNode();`
	`f = MakeFunctionNode(E2.type, p);`
	`Unify(E1.type, f);`
	`E0.type = p;`

$E_0 \rightarrow (E_1 , E_2)$	`E0.type = MakeCrossNode(E1.type, E2.type);`
$E_0 \rightarrow (E_1)$	`E0.type = E1.type ;`

Conclusion

Theoretical Approaches:

- Regular Expressions and Finite Automata
- Context-Free Grammars and Parsing Algorithms
- Attribute Grammars
- Type Theory
    - Function Types
    - Type Expressions
    - Unification Algorithm

Make it possible to parse and check complex, high-level programming lanaguages!

Would not be possible without these theoretical underpinnings!

The Next Step? Generate Target Code and Execute the Program!