Semantic Processing

The Lexer and Parser...
- Found lexical and syntax errors
- Built Abstract Syntax Tree

Now...
- Find semantic errors.
- Build information about the program.

Later...
- Generate IR Code
- Optimize IR Code
- Generate Target Code

Semantic Errors

Undefined ID / ID is already defined

- Other name-related checks (e.g., can’t redefine “true”)
- Field labels
- Labels on loops, gotos, etc.

Semantic Errors

Undefined ID / ID is already defined

- Other name-related checks (e.g., can’t redefine “true”)
- Field labels
- Labels on loops, gotos, etc.

Type checks

- For operators and expressions
- For assignment statements
- Wherever expressions are used (e.g., “if” condition must be boolean)

Semantic Errors

Undefined ID / ID is already defined

- Other name-related checks (e.g., can’t redefine “true”)
- Field labels
- Labels on loops, gotos, etc.

Type checks

- For operators and expressions
- For assignment statements
- Wherever expressions are used (e.g., “if” condition must be boolean)

Flow of control

- Return statement (“expr” must / must not be included)
- Break/continue statement must be within a loop or switch
- Unreachable code? Might want to detect it.

Semantic Errors

Undefined ID / ID is already defined

- Other name-related checks (e.g., can’t redefine “true”)
- Field labels
- Labels on loops, gotos, etc.

Type checks

- For operators and expressions
- For assignment statements
- Wherever expressions are used (e.g., “if” condition must be boolean)

Flow of control

- Return statement (“expr” must / must not be included)
- Break/continue statement must be within a loop or switch
- Unreachable code? Might want to detect it.

Procedure calls

- Wrong number of arguments
- Type of arguments
- Void / non-void conflict

Semantic Errors

Undefined ID / ID is already defined
- Other name-related checks (e.g., can’t redefine “true”)
- Field labels
- Labels on loops, gotos, etc.

Type checks
- For operators and expressions
- For assignment statements
- Wherever expressions are used (e.g., “if” condition must be boolean)

Flow of control
- Return statement (“expr” must / must not be included)
- Break/continue statement must be within a loop or switch
- Unreachable code? Might want to detect it.

Procedure calls
- Wrong number of arguments
- Type of arguments
- Void / non-void conflict

OOP-related checks
- Does this class understand this message?
- Is this field in this class?
- Is private / public access followed?

“Blocks”

Contain variables
May be nested
May contain variable declarations

```
{   var x,y: int;
    ...
    {   var x: double;
        ...
    }
    ...
}
```

Blocks in C++ and Java:

```
void foo {
  double x;
  ...
  for (int x = 0; ...) {
    ...
  }
  ...
}
```

Declarations of Variables

Apply to the statements in the block
...and statements in nested blocks
...unless “hidden” by other declarations

PCAT

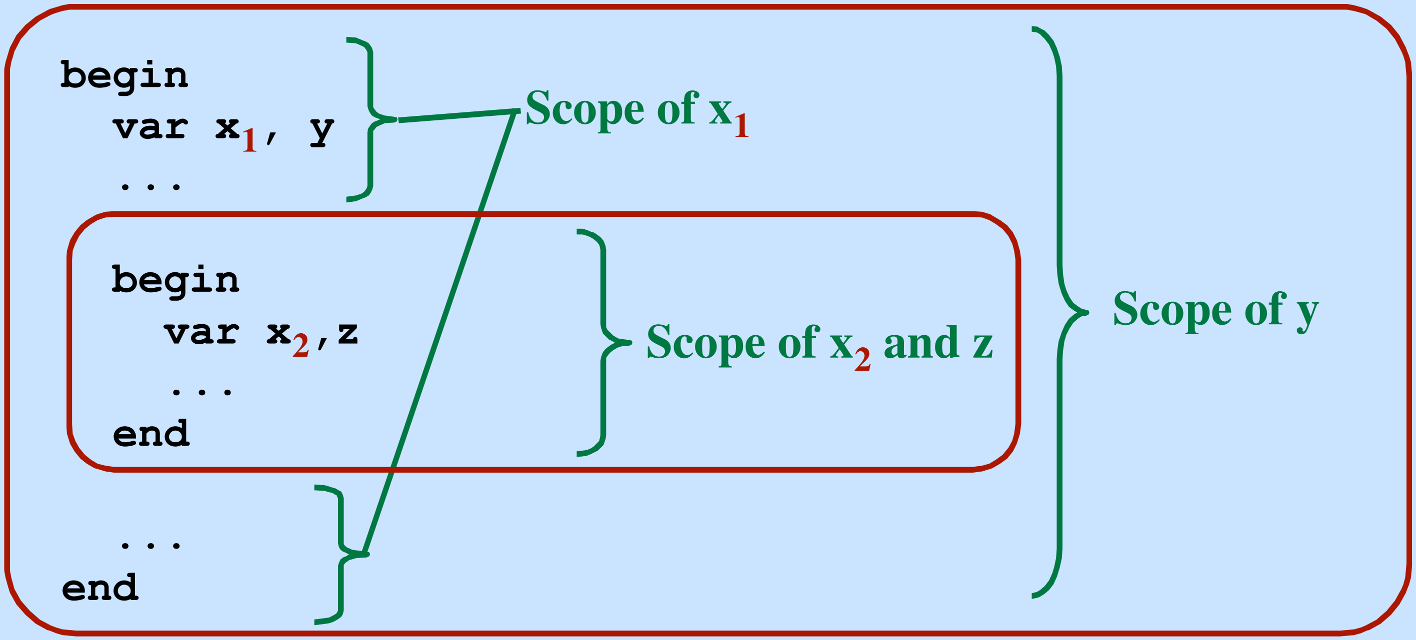
Each “body” is a block
Outermost (main) block (at level 0)
Each procedure constitutes a new block

Scope

(Also: “Lexical scope of variables”)

Where is the variable visible? The scope of the variable.

Scope rules are given in the language definition.



begin
  var x_1**, y**
  ...
  begin
    var x_2**,z**
    ...
  end
  ...
end

Variations

"Variable X's scope extends from the beginning of the block in which it was declared, through the end of the block."

"Variable X's scope extends from the point of its declaration through the end of the block."

"... Unless hidden by a new declaration of a variable with the same name!"

Variations

"Variable X's scope extends from the beginning of the block in which it was declared, through the end of the block."

"Variable X's scope extends from the point of its declaration through the end of the block."

"... Unless hidden by a new declaration of a variable with the same name!"

PCAT

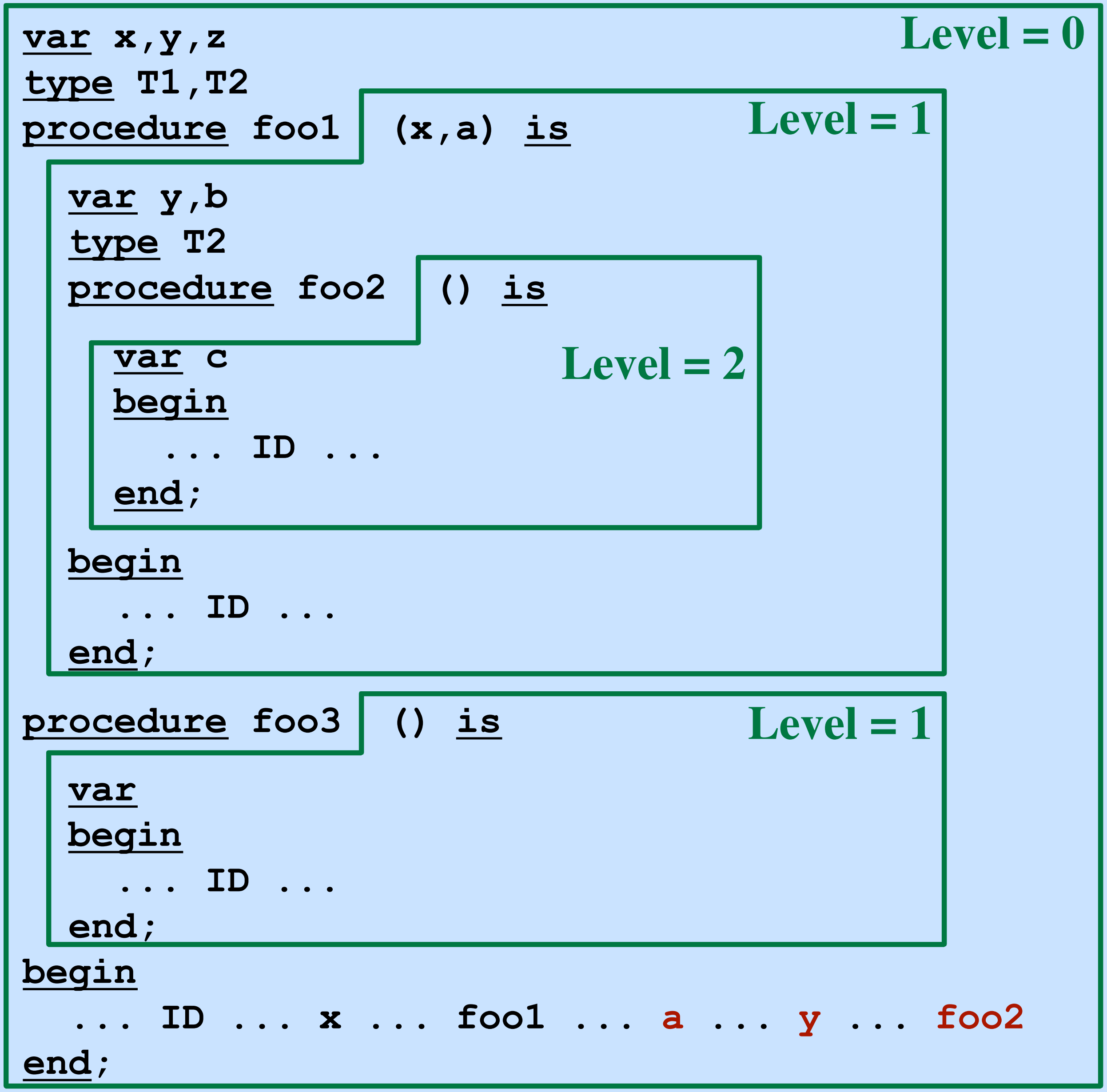
Variables

Visible (i.e., usable) only after their declaration.

Types, Procedures

Visible from the beginning of the block (to allow recursion).

PASS 1: Enter ID's into symbol table
PASS 2: Check all uses



```
var x,y,z                                      Level = 0
type T1,T2
procedure foo1  (x,a) is                   Level = 1
  var y,b
  type T2
  procedure foo2  () is
    var c                          Level = 2
    begin
      ... ID ...
    end;
  begin
    ... ID ...
  end;
procedure foo3  () is                      Level = 1
  var
  begin
    ... ID ...
  end;
begin
  ... ID ... x ... foo1 ... a ... y ... foo2
end;
```

"Static" Level
"Lexical" Level
(Textual)

Functions as Data

```
var f,g: function;
...
f = function (a,b: Int) : Int is
      var t: Int;
        t = a*b;
        return t-1;
    endFunction;
...
g = f;
...
i = g(7,5);
```

This is like a constant.
(It is an expression.)
Within it is a new block.

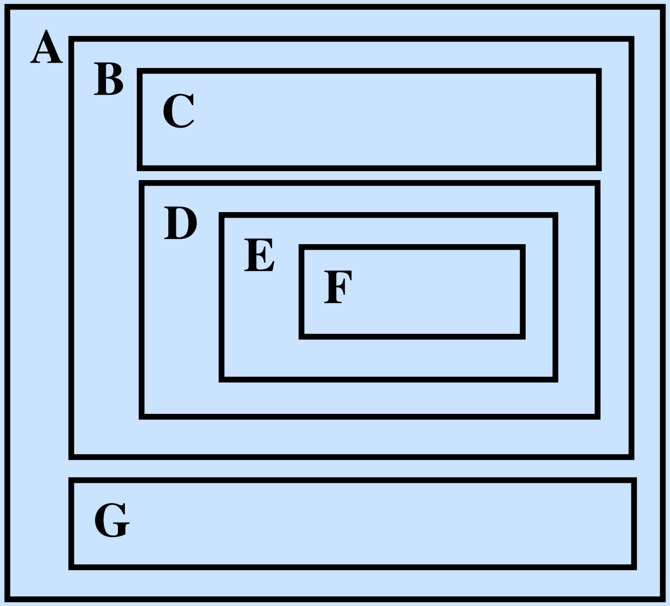
"Lambda Expressions"
"Closures"
"Nameless Functions"

This idea is very powerful!

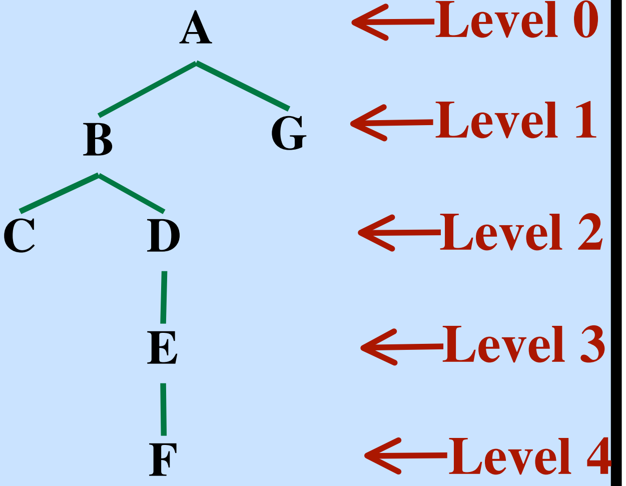
Programs may have more complex behavior
Programmers work at higher level of abstraction

```
begin A
 begin B
  begin C
  end
  begin D
   begin E
    begin F
    end
   end
  end
 end
 begin G
 end
end
```

Blocks are Nested

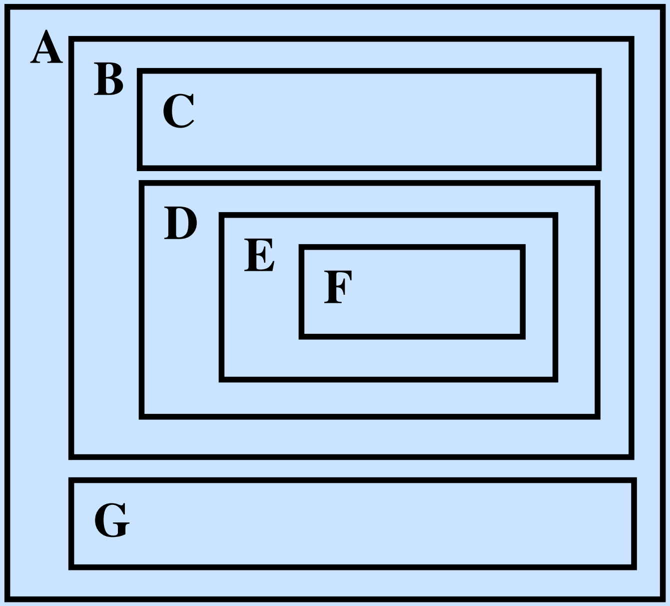


A sequential scan of the program will follow a depth-first traversal of this tree!

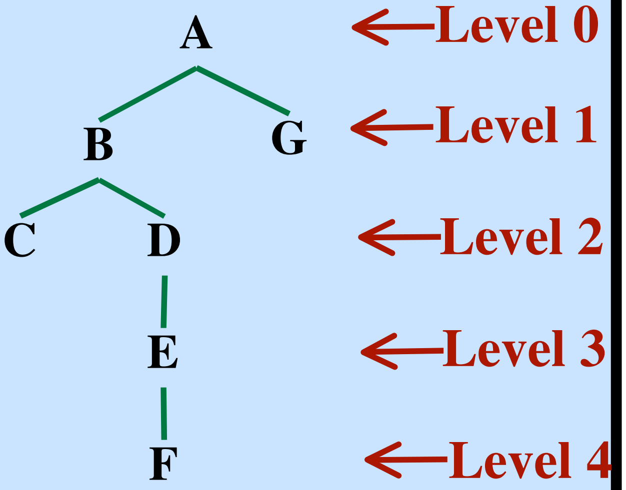


Blocks are Nested

```
begin A
 begin B
  begin C
  end
  begin D
   begin E
    begin F
    end
   end
  end
 end
 begin G
 end
end
```

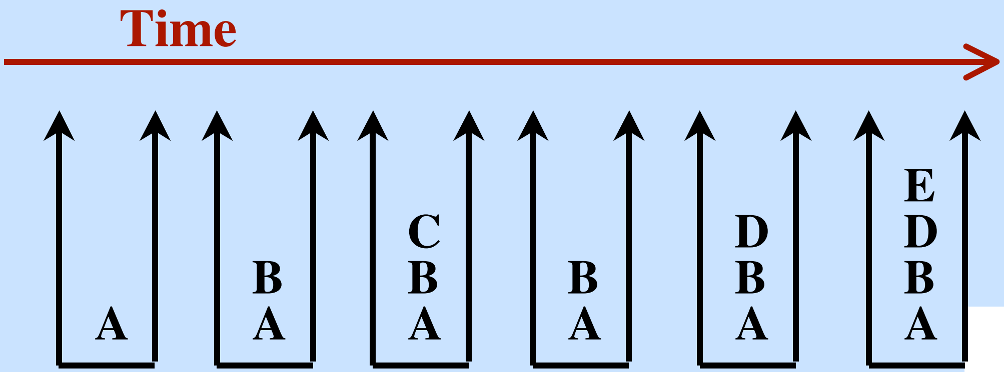


A sequential scan of the program will follow a depth-first traversal of this tree!



The symbol table will work like a stack

openScope = push
closeScope = pop



Goals of Type Checking

Make sure the programmer uses data correctly.

`x + y`	must have numeric types
`x = a;`	types must match (or be "compatible")
`if (expr) then...`	type of expression must be boolean
`a[i]`	"a" must have type array, "i" must have type integer
`r.f`	"r" must have type record.
`foo (a,b,c)`	args must have the right types
`p*`	"p" must be a pointer

Goals of Type Checking

Make sure the programmer uses data correctly.

`x + y`	must have numeric types
`x = a;`	types must match (or be “compatible”)
`if (expr) then...`	type of expression must be boolean
`a[i]`	“a” must have type array, “i” must have type integer
`r.f`	“r” must have type record.
`foo (a,b,c)`	args must have the right types
`p*`	“p” must be a pointer

Need to select the appropiate target operators.

`x+y`	Need to determine “integerAdd” or “doubleAdd”...

Goals of Type Checking

Make sure the programmer uses data correctly.

`x + y`	must have numeric types
`x = a;`	types must match (or be “compatible”)
`if (expr) then...`	type of expression must be boolean
`a[i]`	“a” must have type array, “i” must have type integer
`r.f`	“r” must have type record.
`foo (a,b,c)`	args must have the right types
`p*`	“p” must be a pointer

Need to select the appropiate target operators.

`x+y`	Need to determine “integerAdd” or “doubleAdd”...

Need to insert coercion routines, where necessary.

PCAT:	`i/j` ⇒ `int2real(i)/int2real(j)`

Goals of Type Checking

Make sure the programmer uses data correctly.

`x + y`	must have numeric types
`x = a;`	types must match (or be “compatible”)
`if (expr) then...`	type of expression must be boolean
`a[i]`	“a” must have type array, “i” must have type integer
`r.f`	“r” must have type record.
`foo (a,b,c)`	args must have the right types
`p*`	“p” must be a pointer

Need to select the appropiate target operators.

`x+y`	Need to determine “integerAdd” or “doubleAdd”...

Need to insert coercion routines, where necessary.

PCAT:	`i/j` ⇒ `int2real(i)/int2real(j)`

Determine how much space to allocate for each variable.

Integer ⇒ 32 bits
Double ⇒ 64 bits
Char ⇒ 8 bits
Boolean ⇒ 1 bit

Types

Each language has its own notions of "type."

Basic Types (also called "primitive types")

```
integer, real, character, boolean
```

Constructed Types

Built from other types...

```
array of ...
record { ... }
pointer to ...
function (...) → ...
```

Notations in other languages:

```
int [100] a;

int *p;
int (* foo) (...) {...}
```

We must represent types within out compiler.
Might want a little language of "*type expressions*".
To make explicit...
the universe of all possible types.

Basic Types

Each has a name

```
integer
real
boolean
char
...
void
type_error
```

Close correspondence with keywords in the langauge

Each basic type is a set of values.

Each type will have several

Predefined operators on the values

Void

A type with zero values

Used for typing functions

Type_Error

Used to deal with semantic errors (not really a type)

Array Types

PCAT	`array of real`
Pascal	`array [1..10] of real`
C	`double x [10]`
Java	`double []`
Portlandish	`Array [Integer,Real]`

Element Type (or “Base Type”)

Can be any type

Can even be other array type

```
array of array of real
a[i][j] = (a[i])[j]
```

Index Type

Usually “integer”

...but other possibilities

Pascal: `array [Days] of real`

Often implicit, not really a part of the type

Is the size of the array part of the type???

Pointer Types

PCAT-style	`var p: ptr to integer;`
Pascal	`var p: ↑ integer;`
C	`int * p;`
Java	`MyRec p;`

Element Type (or "Base Type")

Can be any type.

Typical Operations

Comparison	`==`
Copy	`=`
Dereference	`*p`
Increment	`p++`
Convert to/from integer	`p = (int *) 0x0045ff00;`

Record Types ("Structs")

PCAT

```
var r: record
          value: real;
          count: integer;
        end;
```

C

```
struct {
  double value;
  int count;
} r;
```

Java

```
class MyRec {
  double value;
  int count;
}
MyRec r;
```

Each record consists of several values of different types
"components," "fields"

Each component value has different type

The component values are identified by names ("field names")

`r.value`

Product Types (Tuple Types)

Each tuple object consists of several component values.
Each component value has a different type.
(Similar to record types).
Component values are identified by position, not name.

To specify a product type:

Notation #1:

```
var t1: integer × boolean;
    t2: real × real × real × real;
```

Notation #2:

```
var t1: (integer, boolean);
    t2: (real, real, real, real);
```

To specify a tuple:

```
t1 = <6,true>;
t1 = (6,true);
t1 = [6,true];
```

To access the component values:

```
i = t1.1;            i = first(t1);
x = t2.3;            x = third(t2);
```

List Types

Each list object consists of zero or more values, all with the same type.

To specify a list type:

Notation #1:

```
var myList: list of integer;
```

Notation #2:

```
var myList: List[integer];
```

To get the first element of the list:

```
i = head(myList);                 i = car(myList);
```

To get a new list of everything else:

```
otherList = tail(myList);         i = cdr(myList);
```

Add an element to the front and create a new list:

```
newList = cons(i,myList)          newList = i.myList;
```

To create a list:

```
myList = [];                      myList = null;
myList = [3,5,7];                 myList = 3.5.7.null;
```

Other operations:

```
length, append, isEmpty
```

Function Types

Some languages include function types.

Need to associate types with function names.

Functions are “first-class” objects (e.g., they can be stored in arrays, etc.).

To specify a function type:

Notation #1:

DomainType → *RangeType*

```
var f: integer → boolean;
    g: real × real × real × real → void;
```

Notation #2:

function (*DomainTypes*) **returns** *RangeType*

```
var f: function (integer) returns boolean;
    g: function (real, real, real, real);
```

Operations:

Creation and Copy

```
f = function (a:int) returns boolean
      ...
      return ...;
    endFunction
```

Application/Invocation `g (1.5, 2.5, 3.5, 4.5);`

Comparison is usually not allowed.

Working with × and →

Assumptions:

× is associative

```
  (int ×  int) × int
= int × (int ×  int)
= int ×  int ×  int
```

× has greater precedence than →

```
   int × int   → int
= (int × int)  → int
```

→ is right associative

```
  int →  int → int
= int → (int → int)
```

Example

```
type Complex = real × real;

var c: Complex;

c = <1.2, 3.4>);
<x,y> = c;

function ComplexMult: Complex × Complex → Complex

              Complex × Complex → Complex
            = (Complex × Complex) → Complex
            = ((real × real) × (real × real)) → (real × real)
            = real × real × real × real → real × real

<x,y> = ComplexMult (c, <5.6,7.8>);
```

Higher-Order Functions

```
function AddOne: real → real;
AddOne = function (x:real) returns real
            return x + 1.0;
         endFunction;
x = AddOne(123.0);
x = AddOne(AddOne(AddOne(AddOne(AddOne(123.0)))));
```

Imagine a function which takes 2 arguments:

- A function, f
- An integer, N

It returns a function which...

when applied to argument x, will apply function f, N times.

```
function Repeat: (real → real) × int → (real → real);
g = Repeat(AddOne,5);        // g will add 5
x = g(123.0);
x = (Repeat(AddOne,5)) (123.0);
```

`Repeat` is a "Higher-Order Function."

At least one argument or result is another function!

A Syntax of Types

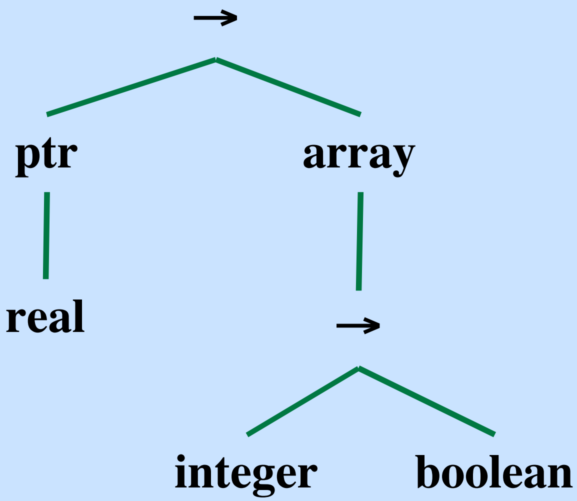
T → `int`
→ `real`
→ `bool`
→ `char`
→ `void`
→ `TypeError`
→ `array of` T
→ `list of` T
→ `ptr to` T
→ `record` `ID` `:` T $\{$ `,` `ID` `:` T $\}^+$ `endRecord`
→ T × T
→ T → T
→ `(` T `)`

Represent each type with a tree
An AST

Using Trees To Represent Types

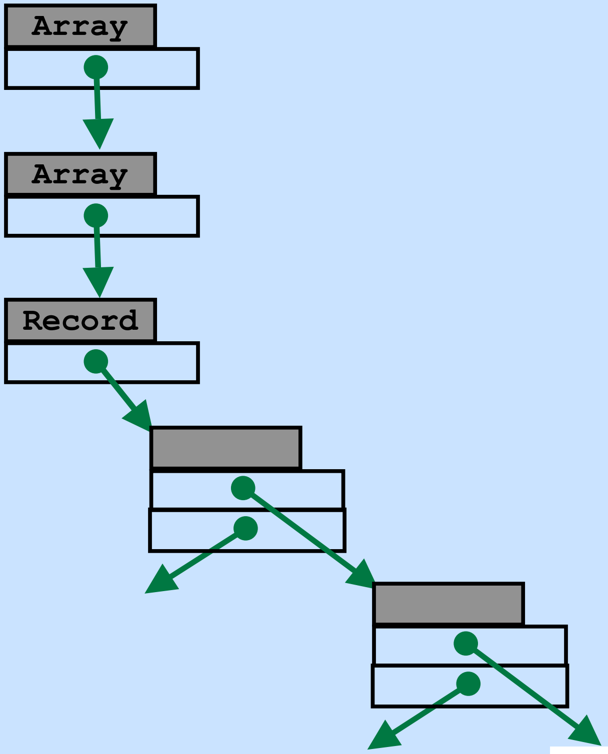
```
type T1 is (ptr to real) → (array of (integer → boolean));
```

The representation of T1...



In our PCAT compiler...

```
array of array of record ... end;
```



Naming Types

Associate a name with a type.

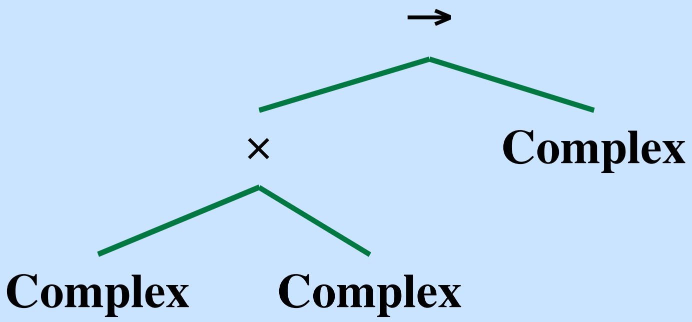
$$\texttt{type }\underbrace{\texttt{MyRec}}_{\textit{name}}\texttt{ is }\underbrace{\texttt{record ... end;}}_{\textit{type}}$$

Example:

```
type Complex is real × real;
function ComplexMult (x, y: Complex) returns Complex is ...;
```

Or perhaps...

```
var ComplexMult: Complex × Complex → Complex;
```



`Complex × Complex → Complex`

Naming Types

Associate a name with a type.

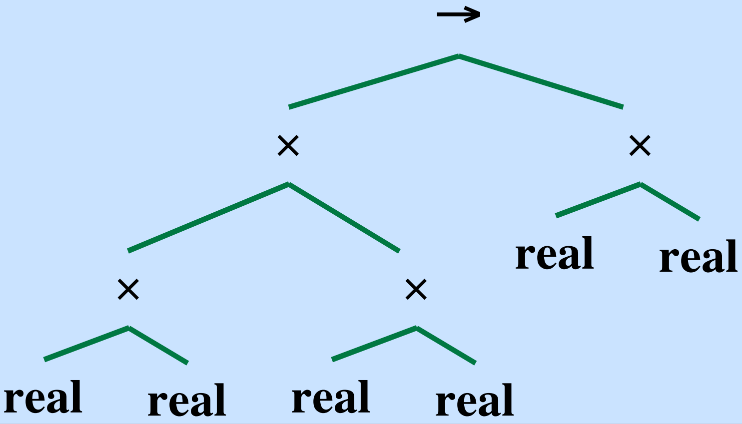
$$\underline{\texttt{type}}\ \underbrace{\texttt{MyRec}}_{\textit{name}}\ \underline{\texttt{is}}\ \underbrace{\underline{\texttt{record}}\ \texttt{...}\ \underline{\texttt{end}}\texttt{;}}_{\textit{type}}$$

Example:

```
type Complex is real × real;
function ComplexMult (x, y: Complex) returns Complex is ...;
```

Or perhaps...

```
var ComplexMult: Complex × Complex → Complex;
```



`real × real × real × real → real × real`

Static v. Dynamic Type Checking

“Static” Type Checking

- Performed by the compiler
- Errors detected?
  - Print a descriptive message and keeping checking
    - Patch up the AST
    - Must cope with previous errors

Static v. Dynamic Type Checking

“Static” Type Checking

- Performed by the compiler
- Errors detected?
  - Print a descriptive message and keeping checking
    - Patch up the AST
    - Must cope with previous errors

“Dynamic” Type Checking

- Checking done at run-time
- Compiler does not know about types.

```
var x, y, z;
...
x = y + z;
```

Integer or Floating Addition?
At runtime, do y and z contain integers or reals or ...?

- Each variable contains:
  - A value
  - Type information (“type tags”)
- Examples:
  - Smalltalk / Squeak
  - Lisp

Untyped Languages

Example: Assembly Language

- There may be different types of data (integer, float, pointers, etc.)
- The programmer says which operations to use (iadd, fadd, ...)
- A type is not associated with each variable.
- If the programmer makes mistakes, the results are wrong.

Untyped Languages

Example: Assembly Language

- There may be different types of data (integer, float, pointers, etc.)
- The programmer says which operations to use (iadd, fadd, ...)
- A type is not associated with each variable.
- If the programmer makes mistakes, the results are wrong.

Strongly Typed Languages

- Each value has an associated type.
- Guarantees that no type-errors can happen.

Example:

```
x = "abc";
y = "def";
z = x - y;
```

Error!

This operation cannot be done on this type of data.

- C/C++

  Type errors can occur, especially with casting.
  "It is the programmer's responsibility!"

Untyped Languages

Example: Assembly Language

- There may be different types of data (integer, float, pointers, etc.)
- The programmer says which operations to use (iadd, fadd, ...)
- A type is not associated with each variable.
- If the programmer makes mistakes, the results are wrong.

Strongly Typed Languages

- Each value has an associated type.
- Guarantees that no type-errors can happen.

Example:

```
x = “abc”;
y = “def”;
z = x - y;
```

Error!

This operation cannot be done on this type of data.

- C/C++

  Type errors can occur, especially with casting.

  “It is the programmer’s responsibility!”

Strong, Static Type Checking

- The compiler checks all types before runtime.
- No type-errors can occur.

Examples: Java, PCAT

Types In PCAT

Basic Types:

- `int`
- `real`
- `bool`
- `string`
- `type_of_nil`

Constructed Types:

- `array`
- `record`

Other:

- `typeError`

The type rules for “nil” are different

```
myArr := nil;
myRec := nil;
```

Representation of a type:

- Pointer to the AST for the type
- Type_Error
  - We’ll use “null” pointer

Approach To Static Type Checking

- ***Need to describe types***

  A representation of types

- ***Associate a type with each variable.***

  The variable declaration associates a type with a variable.
  This info is recorded (in the symbol table).

- ***Associate a type with each expression***

  ...and each sub-expression.

- ***Work bottom-up***

  The type is a "synthesized" attribute

- ***Check operators***

  `expr1 + expr2`

  Is the type of the expressions "integer" or "real"?

- ***Check other places that expressions are used***

  `LValue := Expr ;`

  Is the type of "expr" equal to the type of the L-Value?

  `if (expr) ...`

  Is the type of the expression "boolean"?

Operator Overloading

PCAT Example:

```
  var x,y: int;
  ...
    x+y
  ...
```

PCAT has two kinds of addition

The "+" operation is "***overloaded***"

Multiple meanings:

`iadd`

`fadd`

Also multiple kinds of negation, subtraction, multiplication, comparison, ...

Select correct operation based on argument types.

We'll use the term "***mode***"

`INTEGER_MODE`

`REAL_MODE`

Tells which form of addition will be needed.

Type Conversions

PCAT Example:

```
  var i: int,
         x: real;
  ... (x + i) ...
```

Must convert the integer value to a real value first.
Real addition (`fadd`) will be used.
The result will be a real.

Implicit Type Conversions (also called "Coercions")

- The language definition tells when they are needed.
- Compiler must insert special code to perform the operation.

Explicit Type Conversions (also called "Casting")

```
... (i + (int) x) ...
```

- The programmer requests a specific conversion.
- The language definition tells when they allowed.
- The compiler may (or may not) need to insert special code.

Types In PCAT: Unary Operators

Given: Type of operand
Determine: Type of result

	`not`	`+`	`-`
int		int	int
real		real	real
bool	bool		
string			
array			
record			
type error			

Blank entries indicate “type error”

Types In PCAT: Unary Operators

Given: Type of operand

Determine: Type of result

	`not`	`+`	`-`
int		int	int
real		real	real
bool	bool		
string			
array			
record			
type error			

Blank entries indicate "type error"

Implementation Ideas:

7 × 3 array

```
ResultType[bool,not] ⇒ bool
```

Sequence of IF tests...

```
if (op == PLUS) or (op == MINUS) then
    if typeOfOperand == int then
        resultType = int;
    elseIf typeOfOperand == real then
        resultType = real;
    else
        resultType = null;    // TypeError;
    endIf
elseIf (op == NOT) then ...
```

Types In PCAT: Binary Operators

Operand 1	Operand 2	+ - *	/	and or	= <>	< <= > >=	div mod	:=
int	int	int	real*		bool	bool	int	ok
int	real	real*	real*		bool*	bool*		
real	int	real*	real*		bool*	bool*		ok*
real	real	real	real		bool	bool		ok
bool	bool			bool	bool			ok
type error	(any)							
(any)	type error							
(other)	(other)				bool**			ok**

* means the int argument(s) must be coerced to real

** means ok if the arguments are the same type

Types In PCAT: Binary Operators

Operand 1	Operand 2	+ - *	/	and or	= <>	< <= > >=	div mod	:=
int	int	int	real*		bool	bool	int	ok
int	real	real*	real*		bool*	bool*		
real	int	real*	real*		bool*	bool*		ok*
real	real	real	real		bool	bool		ok
bool	bool			bool	bool			ok
type error	(any)							
(any)	type error							
(other)	(other)				bool**			ok**

* means the int argument(s) must be coerced to real

** means ok if the arguments are the same type

Implementation Ideas:

Use a 7 × 7 × 15 array? Nah...

Switch on operator first, then on operand type.

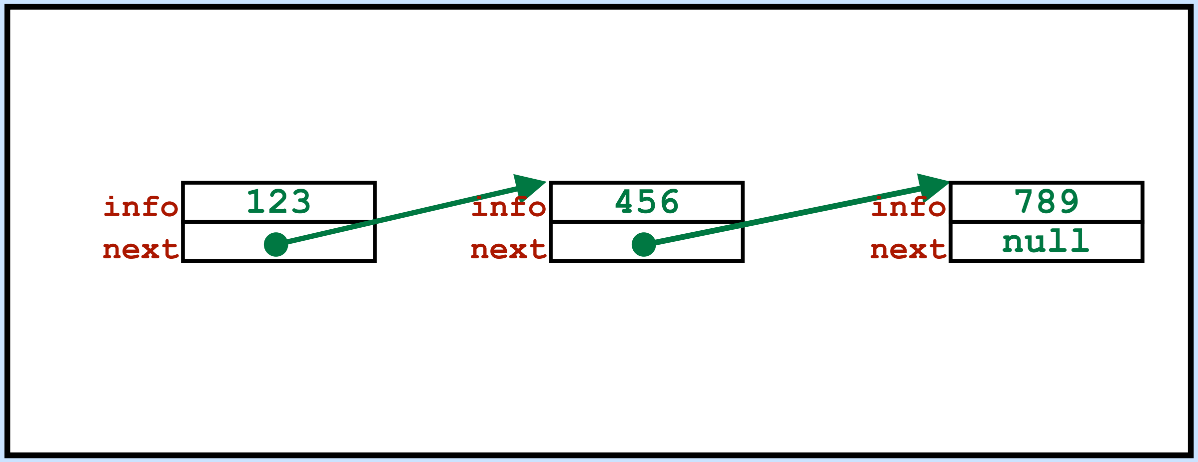
Recursive Types

```
type MyRec is record
                info: integer;
                next: MyRec;
              end;
var x: MyRec := MyRec { info := 789;
                        next := null };
```

Recursive Types

```
type MyRec is record
                info: integer;
                next: MyRec;
              end;
var x: MyRec := MyRec { info := 789;
                        next := null };
```

All records and arrays will go into the **"Heap"**.

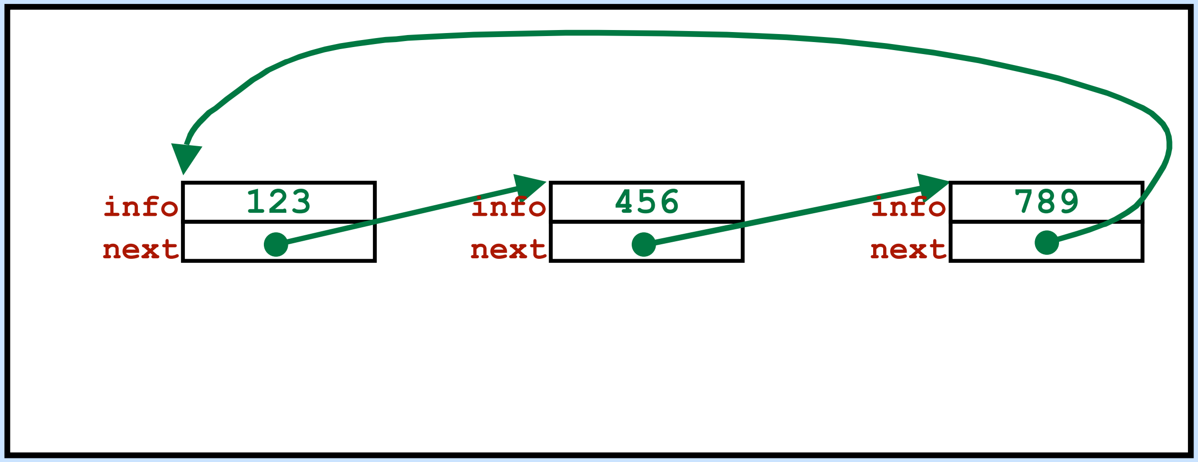


The Heap

Recursive Types

```
type MyRec is record
                info: integer;
                next: MyRec;
              end;
var x: MyRec := MyRec { info := 789;
                        next := null };
```

All records and arrays will go into the **"Heap"**.

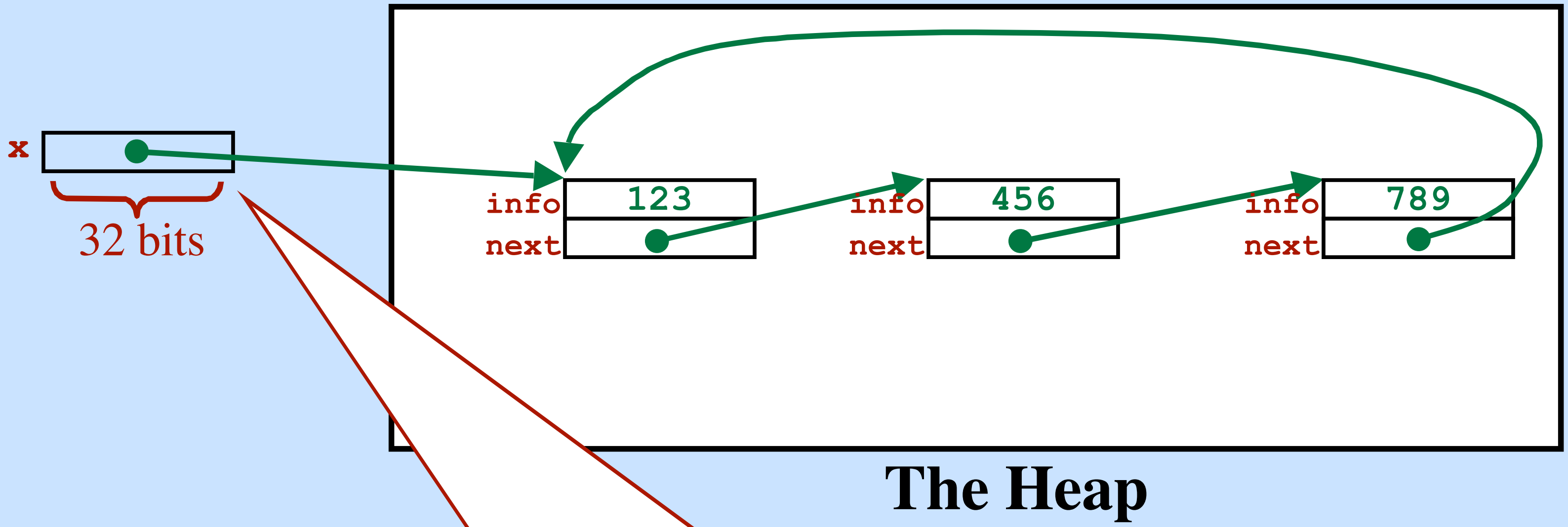


The Heap

Recursive Types

```
type MyRec is record
                info: integer;
                next: MyRec;
              end;
var x: MyRec := MyRec { info := 789;
                        next := null };
```

All records and arrays will go into the **"Heap"**.

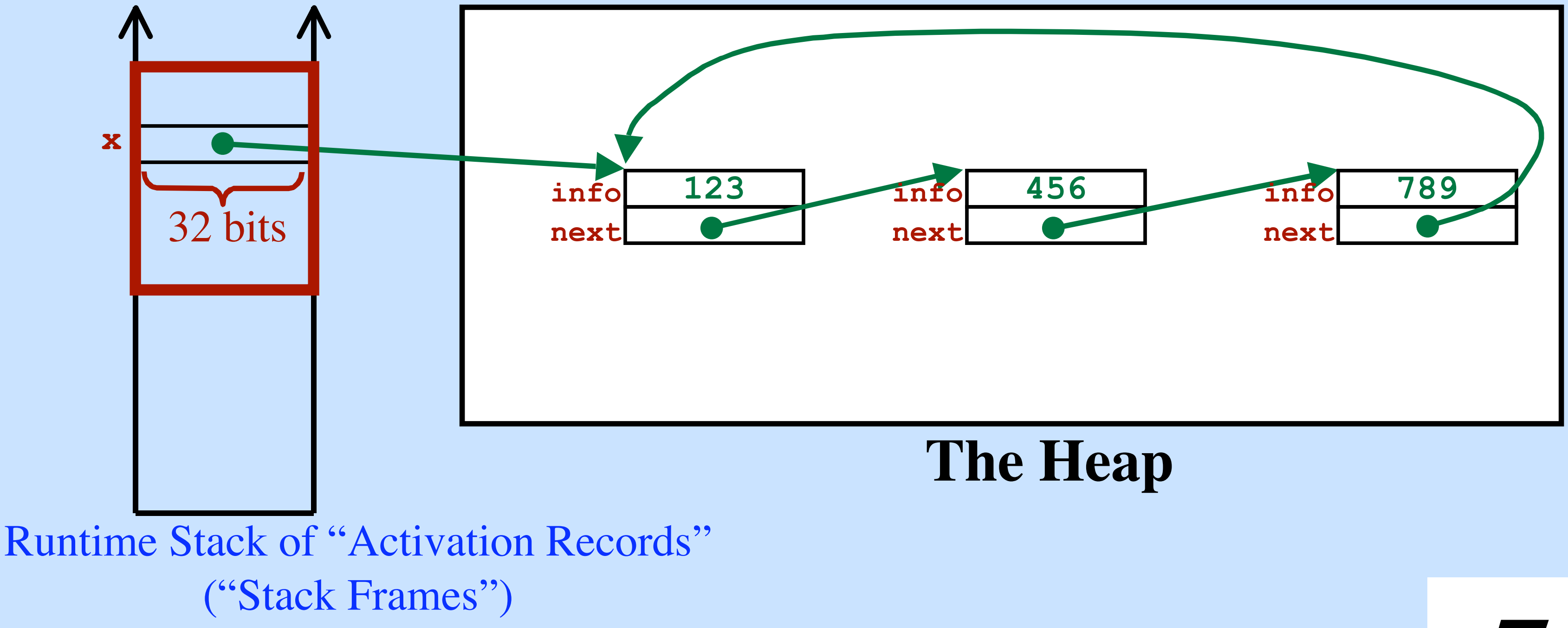


Our Implementation: all variables will be 32 bits

Recursive Types

```
type MyRec is record
                 info: integer;
                 next: MyRec;
              end;
var x: MyRec := MyRec { info := 789;
                        next := null };
```

All records and arrays will go into the **"Heap"**.



The Heap

Runtime Stack of "Activation Records"
("Stack Frames")

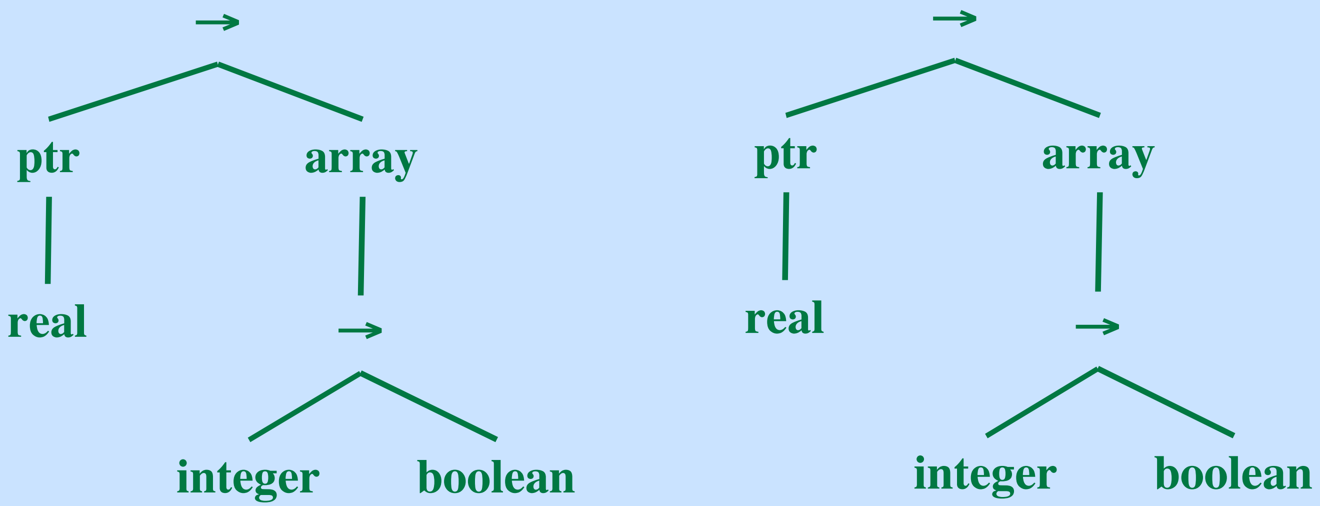
Type Equivalence

What does it mean to say "type of operand 1" = *"type of operand 2"?*

```
type T1 is record
             f: int;
             g: real;
           end;
     T2 is record
             f: int;
             g: real;
           end;
     T3 is T2;
var x: T1,
    y: T2,
    z: T3;
...
x := y;
```

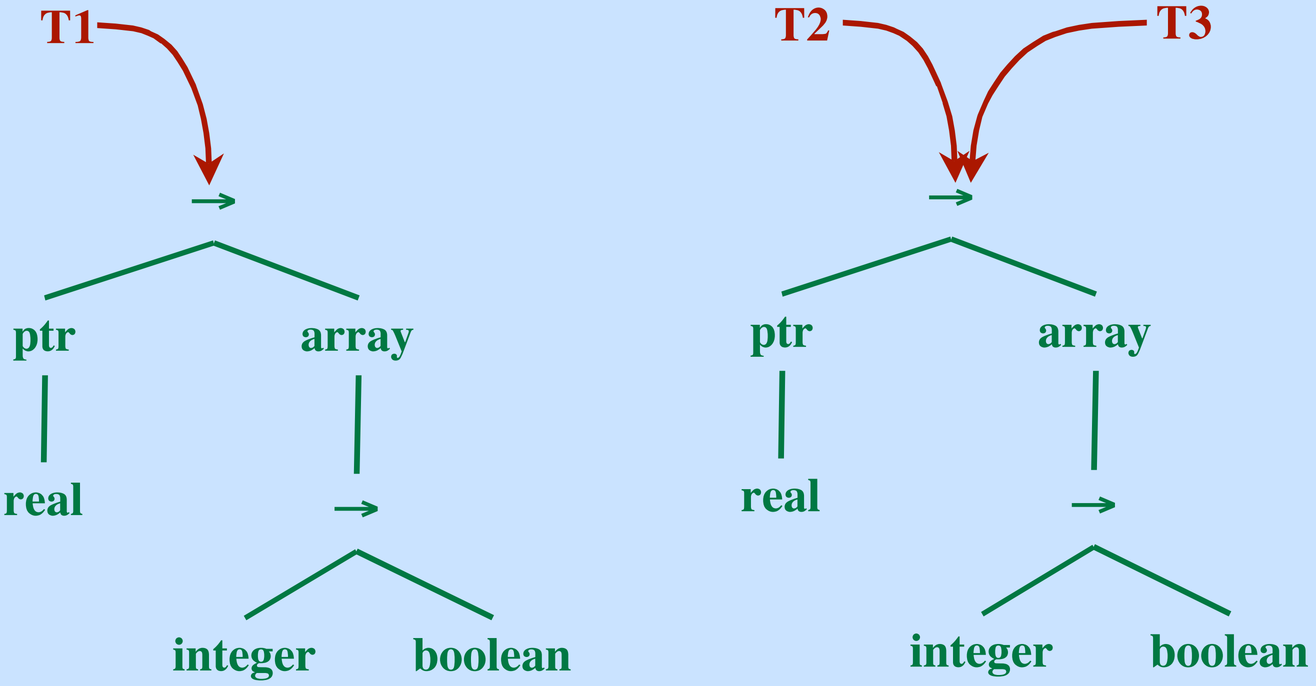
Is the type of "x" the same as the type of "y"?
Is the type of "y" the same as the type of "z"?

Types are represented as trees.



Types are represented as trees.
Types may be named.

```
type T1 is ... ;
```



“Structural Equivalence”

Are the trees equivalent?

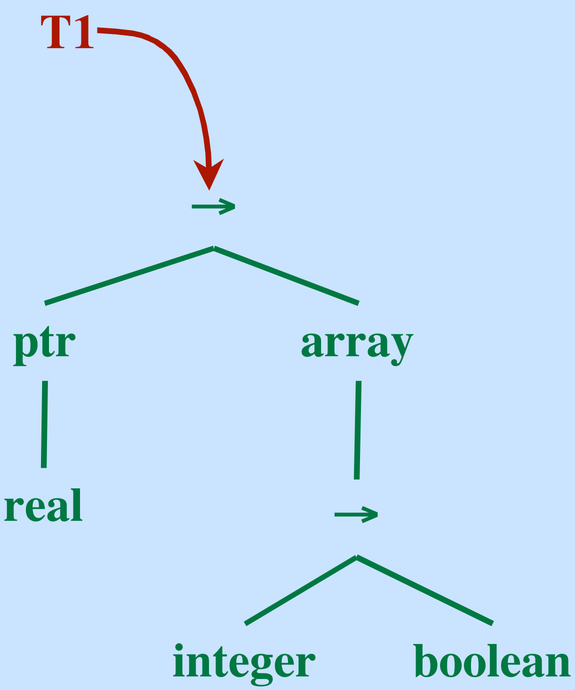
Isomorphic (same shape, same nodes)

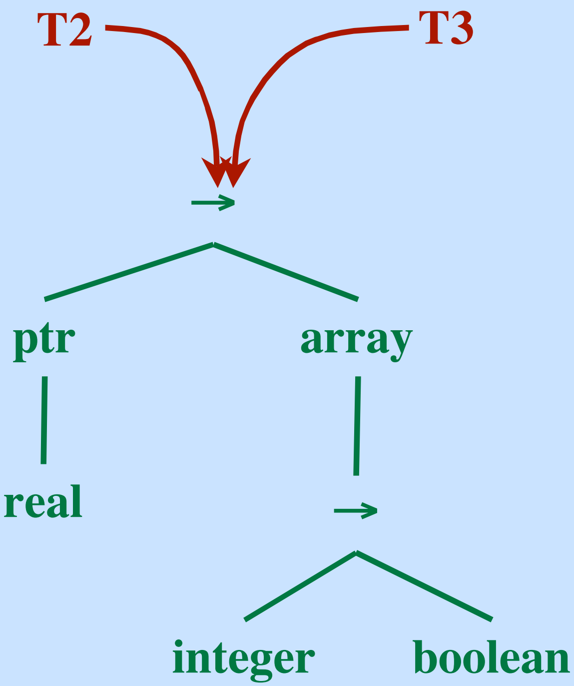
Must walk the trees to check.

“Name Equivalence”

Are they the same tree?

Compare pointers





Testing Structural Equivalence

```
function typeEquiv (s, t) returns boolean
  if (s and t are the same “basic” type) then
    return true
  elseif (s = “array of s1”) and (t = “array of t1”) then
    return typeEquiv (s1,t1)
  elseif (s = “s1 × s2”) and (t = “t1 × t2”) then
    return typeEquiv (s1,t1) and typeEquiv (s2,t2)
  elseif (s = “ptr to s1”) and (t = “ptr to t1”) then
    return typeEquiv (s1,t1)
  elseif (s = “s1 → s2”) and (t = “t1 → t2”) then
    return typeEquiv (s1,t1) and typeEquiv (s2,t2)
  else
    return false
  endIf
endFunction
```