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;
        ...
    }
    ...
}
```

**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

**Blocks in C++ and Java:**

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

begin
  var x_2, z
  ...
  end
... end
```

Scope of $x_1$

Scope of $x_2$ and $z$

Scope of $y$
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!”
Semantics - Part 1

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
type T1,T2
procedure foo1 (x,a) is
    var y,b
type T2
    procedure foo2 () is
        var c
        begin
            ... ID ...
        end;
    begin
        ... ID ...
    end;
procedure foo3 () is
    begin
        ... ID ...
    end;
begin
    ... ID ...
foo1 ...
a ...
y ...
foo2
end
```

© Harry H. Porter, 2005
Semantics - Part 1

**Functions as Data**

```plaintext
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 idea is very powerful!

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

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

© Harry H. Porter, 2005
Blocks are Nested

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

© Harry H. Porter, 2005
Goals of Type Checking

Make sure the programmer uses data correctly.

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

Need to select the appropriate 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 appropriate target operators.

- \(x+y\) Need to determine “integerAdd” or “doubleAdd”...

Need to insert coercion routines, where necessary.

- `PCAT:` \(i/j \Rightarrow \text{int2real}(i)/\text{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 appropriate target operators.

- \( x+y \) Need to determine “integerAdd” or “doubleAdd”...

Need to insert coercion routines, where necessary.

- PCAT: \( i/j \Rightarrow \text{int2real}(i)/\text{int2real}(j) \)

Determine how much space to allocate for each variable.

- Integer \( \Rightarrow \) 32 bits
- Double \( \Rightarrow \) 64 bits
- Char \( \Rightarrow \) 8 bits
- Boolean \( \Rightarrow \) 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 our 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

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 \texttt{var p: ptr to integer;}
Pascal \texttt{var p: ↑ integer;}
C \texttt{int * p;}
Java \texttt{MyRec p;}

Element Type (or “Base Type”)
Can be any type.

Typical Operations
Comparison \texttt{==}
Copy \texttt{=}
Dereference \texttt{*p}
Increment \texttt{p++}
Convert to/from integer \texttt{p = (int *) 0x0045ff00;}
Record Types ("Structs")

PCAT

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

C

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

Java

```plaintext
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 x boolean;
t2: real x real x real x 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:*

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

*Notation #2:*

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

To get the first element of the list:

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

To get a new list of everything else:

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

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

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

To create a list:

```hs
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:**

\[ \text{DomainType} \rightarrow \text{RangeType} \]

\[
\begin{align*}
\text{var } f & : \text{integer } \rightarrow \text{boolean}; \\
g & : \text{real } \times \text{real } \times \text{real } \times \text{real } \rightarrow \text{void};
\end{align*}
\]

**Notation #2:**

\[
\begin{align*}
\text{function } (\text{DomainTypes}) \text{ returns } \text{RangeType} \\
\text{var } f & : \text{function } (\text{integer}) \text{ returns } \text{boolean}; \\
g & : \text{function } (\text{real}, \text{real}, \text{real}, \text{real});
\end{align*}
\]

Operations:

**Creation and Copy**

\[
\text{f} = \text{function } (a:\text{int}) \text{ returns } \text{boolean} \\
\ldots \\
\text{return } \ldots; \\
\text{endFunction}
\]

**Application/Invocation**

\[
\text{g} (1.5, 2.5, 3.5, 4.5);
\]

**Comparison** is usually not allowed.
Semantic - Part 1

**Working with $\times$ and $\to$**

Assumptions:
- $\times$ is associative
  - $(\text{int} \times \text{int}) \times \text{int} = \text{int} \times (\text{int} \times \text{int})$
  - $= \text{int} \times \text{int} \times \text{int}$
- $\times$ has greater precedence than $\to$
  - $\text{int} \times \text{int} \rightarrow \text{int}$
  - $= (\text{int} \times \text{int}) \rightarrow \text{int}$
- $\rightarrow$ is right associative
  - $\text{int} \rightarrow \text{int} \rightarrow \text{int}$
  - $= \text{int} \rightarrow (\text{int} \rightarrow \text{int})$
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

```haskell
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.

```haskell
function Repeat: (real -> real) x 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

\[
\text{type } T_1 \text{ is (ptr to real)} \rightarrow (\text{array of (integer } \rightarrow \text{ boolean)})\;.
\]

The representation of \( T_1 \)...

In our PCAT compiler...

\[
\text{array of array of record ... end;}
\]
Naming Types

Associate a name with a type.

```plaintext
type MyRec is record ... end;
```

**Example:**

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

**Or perhaps...**

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

```
Complex × Complex
    ↓
    × Complex
```

```
Complex × Complex → Complex
```

© Harry H. Porter, 2005
Naming Types

Associate a name with a type.

```pascal
type MyRec is record ... end;
```

**Example:**

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

**Or perhaps...**

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

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

© Harry H. Porter, 2005
Static v. Dynamic Type Checking

“Static” Type Checking
- Performed by the compiler
- Errors detected?
  - Print a descriptive message and keep checking
  - Patch up the AST
  - Must cope with previous errors
"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;
  ```
- Each variable contains:
  - A value
  - Type information ("type tags")
- Examples:
  - Smalltalk / Squeak
  - Lisp

Integer or Floating Addition?
At runtime, do y and z contain integers or reals or ...?
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:* 
- \( x = \text{"abc";} \)
- \( y = \text{"def";} \)
- \( z = x - y; \)

- C/C++
  - Type errors can occur, especially with casting.
  - “It is the programmer’s responsibility!”

**Error!**
This operation cannot be done on this type of data.
Untyped Languages

*Example:* Assembly Language
- There may be different types of data (integer, float, pointers, etc.)
- The programmer specifies 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;
  ```

- 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

Error!
This operation cannot be done on this type of data.
Types In PCAT

**Basic Types:**
- int
- real
- bool
- string
- type_of_nil

**Constructed Types:**
- array
- record

**Other:**
- typeError

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

The type rules for “nil” are different

myArr := nil;
myRec := nil;
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
  \( \text{expr1} + \text{expr2} \)
  Is the type of the expressions “integer” or “real”?

• Check other places that expressions are used
  \( \text{LValue} := \text{Expr} ; \)
  Is the type of “expr” equal to the type of the L-Value?
  \( \text{if (expr)} \ldots \)
  Is the type of the expression “boolean”?
Operator Overloading

PCAT Example:

```pascal
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:

```plaintext
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”)

```plaintext
... (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

<table>
<thead>
<tr>
<th></th>
<th>not</th>
<th>+</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>int</td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>array</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>record</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type error</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Types In PCAT: Unary Operators

**Given:** Type of operand
**Determine:** Type of result

**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

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operand 2</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>and</th>
<th>or</th>
<th>=</th>
<th>&lt;&gt;</th>
<th>&gt;=</th>
<th>&lt;=</th>
<th>div</th>
<th>mod</th>
<th>:=</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>int</td>
<td></td>
<td>real*</td>
<td></td>
<td>bool</td>
<td>bool</td>
<td>int</td>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>real</td>
<td>real*</td>
<td>real*</td>
<td></td>
<td>bool*</td>
<td>bool*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>int</td>
<td>real*</td>
<td>real*</td>
<td></td>
<td>bool*</td>
<td>bool*</td>
<td>ok*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real</td>
<td></td>
<td>bool</td>
<td>bool</td>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td></td>
<td></td>
<td></td>
<td>bool</td>
<td>bool</td>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type error</td>
<td>(any)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(any)</td>
<td>type error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(other)</td>
<td>(other)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* means the int argument(s) must be coerced to real
** means ok if the arguments are the same type
### Types In PCAT: Binary Operators

<table>
<thead>
<tr>
<th>Operand 1</th>
<th>Operand 2</th>
<th>+</th>
<th>-*</th>
<th>/</th>
<th>and</th>
<th>or</th>
<th>=</th>
<th>&lt;&gt;</th>
<th>&gt;=</th>
<th>&lt;=</th>
<th>div</th>
<th>mod</th>
<th>:=</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>int</td>
<td>int</td>
<td>real*</td>
<td></td>
<td>bool</td>
<td>bool</td>
<td>int</td>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td>real</td>
<td>real*</td>
<td>real*</td>
<td></td>
<td>bool*</td>
<td>bool*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>int</td>
<td>real*</td>
<td>real*</td>
<td></td>
<td>bool*</td>
<td>bool*</td>
<td>ok*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real</td>
<td>real</td>
<td>real</td>
<td>real</td>
<td></td>
<td>bool</td>
<td>bool</td>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>bool</td>
<td>real</td>
<td>real</td>
<td></td>
<td>bool</td>
<td>bool</td>
<td>ok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>type error</td>
<td>(any)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(any)</td>
<td>type error</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(other)</td>
<td>(other)</td>
<td></td>
<td>bool**</td>
<td></td>
<td></td>
<td>ok**</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* means the int argument(s) must be coerced to real
** means ok if the arguments are the same type

**Implementation Ideas:**

Use a $7 \times 7 \times 15$ array? Nah...
Switch on operator first, then on operand type.
Recursive Types

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

var x: MyRec := MyRec { info := 789;
                        next := null };```

© Harry H. Porter, 2005
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

```plaintext
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

```pascal
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

```plaintext
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".

Runtime Stack of "Activation Records"
(“Stack Frames”)
Type Equivalence

What does it mean to say “type of operand 1” = “type of operand 2”?

```pascal
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.

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

```
T1

  ptr
    real

  array
    integer
    boolean

T2

  ptr
    real

  array
    integer
    boolean

T3
```

© Harry H. Porter, 2005
“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 x s2”) and (t = “t1 x 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