

# Types and Static Semantic Analysis

COMS W4115



Prof. Stephen A. Edwards

Fall 2006

Columbia University

Department of Computer Science

# Data Types

What is a type?

*A restriction on the possible interpretations of a segment of memory or other program construct.*

Useful for two reasons:

Runtime optimization: earlier binding leads to fewer runtime decisions. E.g., Addition in C efficient because type of operands known.

Error avoidance: prevent programmer from putting round peg in square hole. E.g., In Java, can't open a complex number, only a file.

# Are Data Types Necessary?

No: many languages operate just fine without them.

Assembly languages usually view memory as undifferentiated array of bytes. Operators are typed, registers may be, data is not.

Basic idea of stored-program computer is that programs be indistinguishable from data.

Everything's a string in Tcl including numbers, lists, etc.



# C's Types: Base Types/Pointers

Base types match typical processor

<b>Typical sizes:</b>	8	16	32	64
	char	short	int	long
			float	double

Pointers (addresses)

```
int *i;    /* i is a pointer to an int */
char **j;  /* j is a pointer to
           a pointer to a char */
```

# C's Types: Arrays, Functions

## Arrays

```
char c[10]; /* c[0] ... c[9] are chars */
double a[10][3][2]; /* array of 10
                    arrays of 3 arrays
                    of 2 doubles */
```

## Functions

```
/* function of two arguments
   returning a char */
char foo(int, double);
```

# C's Types: Structs and Unions

Structures: each field has own storage

```
struct box {  
    int x, y, h, w;  
    char *name;  
};
```

Unions: fields share same memory

```
union token {  
    int i;  
    double d;  
    char *s;  
};
```



# Composite Types: Records

A record is an object with a collection of fields, each with a potentially different type. In C,

```
struct rectangle {  
    int n, s, e, w;  
    char *label;  
    color col;  
    struct rectangle *next;  
};
```

```
struct rectangle r;  
r.n = 10;  
r.label = "Rectangle";
```

# Applications of Records

Records are the precursors of objects:

Group and restrict what can be stored in an object, but not what operations they permit.

Can fake object-oriented programming:

```
struct poly { ... };
```

```
struct poly *poly_create();
```

```
void poly_destroy(struct poly *p);
```

```
void poly_draw(struct poly *p);
```

```
void poly_move(struct poly *p, int x, int y);
```

```
int poly_area(struct poly *p);
```



# Composite Types: Variant Records

A record object holds all of its fields. A variant record holds only one of its fields at once. In C,

```
union token {  
    int i;  
    float f;  
    char *string;  
};
```

```
union token t;  
t.i = 10;  
t.f = 3.14159;    /* overwrites t.i */  
char *s = t.string; /* returns gibberish */
```

# Applications of Variant Records

A primitive form of polymorphism:

```
struct poly {  
    int x, y;  
    int type;  
    union { int radius;  
            int size;  
            float angle; } d;  
};
```

If `poly.type == CIRCLE`, use `poly.d.radius`.

If `poly.type == SQUARE`, use `poly.d.size`.

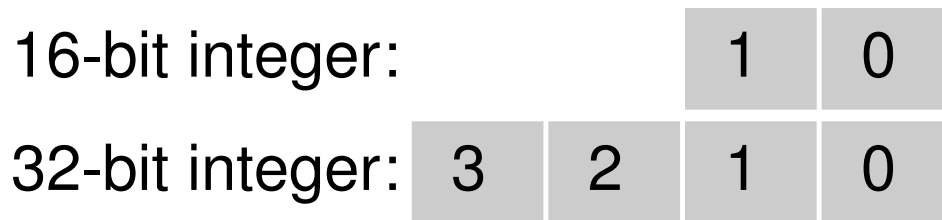
If `poly.type == LINE`, use `poly.d.angle`.

# Layout of Records and Unions

Modern processors have byte-addressable memory.



Many data types (integers, addresses, floating-point numbers) are wider than a byte.



# Layout of Records and Unions

Modern memory systems read data in 32-, 64-, or 128-bit chunks:

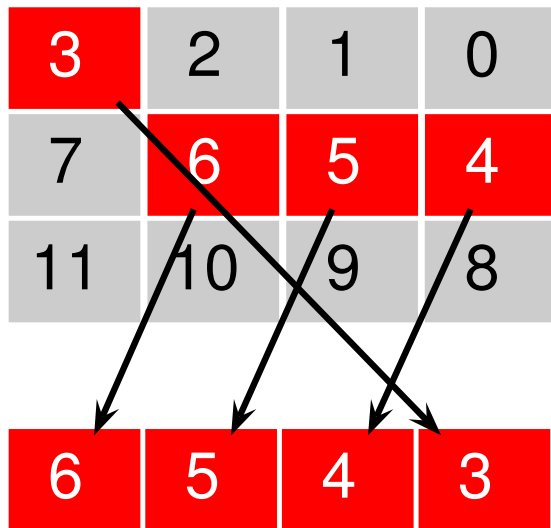
3	2	1	0
7	6	5	4
11	10	9	8

Reading an aligned 32-bit value is fast: a single operation.

3	2	1	0
7	6	5	4
11	10	9	8

# Layout of Records and Unions

Slower to read an unaligned value: two reads plus shift.



SPARC prohibits unaligned accesses.

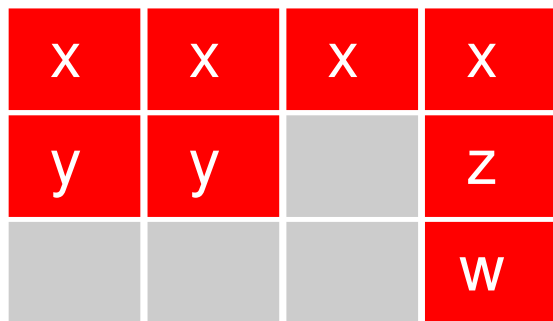
MIPS has special unaligned load/store instructions.

x86, 68k run more slowly with unaligned accesses.

# Layout of Records and Unions

Most languages “pad” the layout of records to ensure alignment restrictions.

```
struct padded {  
    int x;    /* 4 bytes */  
    char z;   /* 1 byte  */  
    short y;  /* 2 bytes */  
    char w;   /* 1 byte  */  
};
```



 : Added padding

# C's Type System: Enumerations

```
enum weekday {sun, mon, tue, wed,  
              thu, fri, sat};
```

```
enum weekday day = mon;
```

Enumeration constants in the same scope must be unique:

```
enum days {sun, wed, sat};
```

```
enum class {mon, wed}; /* error: mon, wed  
                        redefined */
```

# C's Type System

Types may be intermixed at will:

```
struct {
    int i;
    union {
        char (*one) (int);
        char (*two) (int, int);
    } u;
    double b[20][10];
} *a[10];
```

Array of ten pointers to structures. Each structure contains an int, a 2D array of doubles, and a union that contains a pointer to a char function of one or two arguments.



# Strongly-typed Languages

Strongly-typed: no run-time type clashes.

C is definitely not strongly-typed:

```
float g;  
union { float f; int i } u;  
u.i = 3;  
g = u.f + 3.14159; /* u.f is meaningless */
```

Is Java strongly-typed?

# Statically-Typed Languages

Statically-typed: compiler can determine types.

Dynamically-typed: types determined at run time.

Is Java statically-typed?

```
class Foo {  
    public void x() { ... }  
}  
class Bar extends Foo {  
    public void x() { ... }  
}  
void baz(Foo f) {  
    f.x();  
}
```

# Polymorphism

Say you write a sort routine:

```
void sort(int a[], int n)
{
    int i, j;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if ( a[j] < a[i] ) {
                int tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}
```



# Polymorphism

To sort doubles, only need to change a few types:

```
void sort(double a[], int n)
{
    int i, j;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if ( a[j] < a[i] ) {
                double tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}
```



# C++ Templates

```
template <class T> void sort(T a[], int n)
{
    int i, j;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if (a[j] < a[i]) {
                T tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}

int a[10];
sort<int>(a, 10);
```

# C++ Templates

C++ templates are essentially language-aware macros. Each instance generates a different refinement of the same code.

```
sort<int> (a, 10);
```

```
sort<double> (b, 30);
```

```
sort<char *> (c, 20);
```

Fast code, but lots of it.

# Faking Polymorphism with Objects

```
class Sortable {
    bool lessthan(Sortable s) = 0;
}

void sort(Sortable a[], int n) {
    int i, j;
    for ( i = 0 ; i < n-1 ; i++ )
        for ( j = i + 1 ; j < n ; j++ )
            if ( a[j].lessthan(a[i]) ) {
                Sortable tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
            }
}
```

# Faking Polymorphism with Objects

This `sort` works with any array of objects derived from `Sortable`.

Same code is used for every type of object.

Types resolved at run-time (dynamic method dispatch).

Does not run as quickly as the C++ template version.



# Arrays



Most languages provide array types:

```
char i[10];
```

```
/* C */
```

```
character(10) i
```

```
! FORTRAN
```

```
i : array (0..9) of character; -- Ada
```

```
var i : array [0 .. 9] of char; { Pascal }
```

# Array Address Calculation

In C,

```
struct foo a[10];
```

`a[i]` is at  $a + i * \text{sizeof}(\text{struct foo})$

```
struct foo a[10][20];
```

`a[i][j]` is at  $a + (j + 20 * i) * \text{sizeof}(\text{struct foo})$

⇒ Array bounds must be known to access 2D+ arrays

# Allocating Arrays

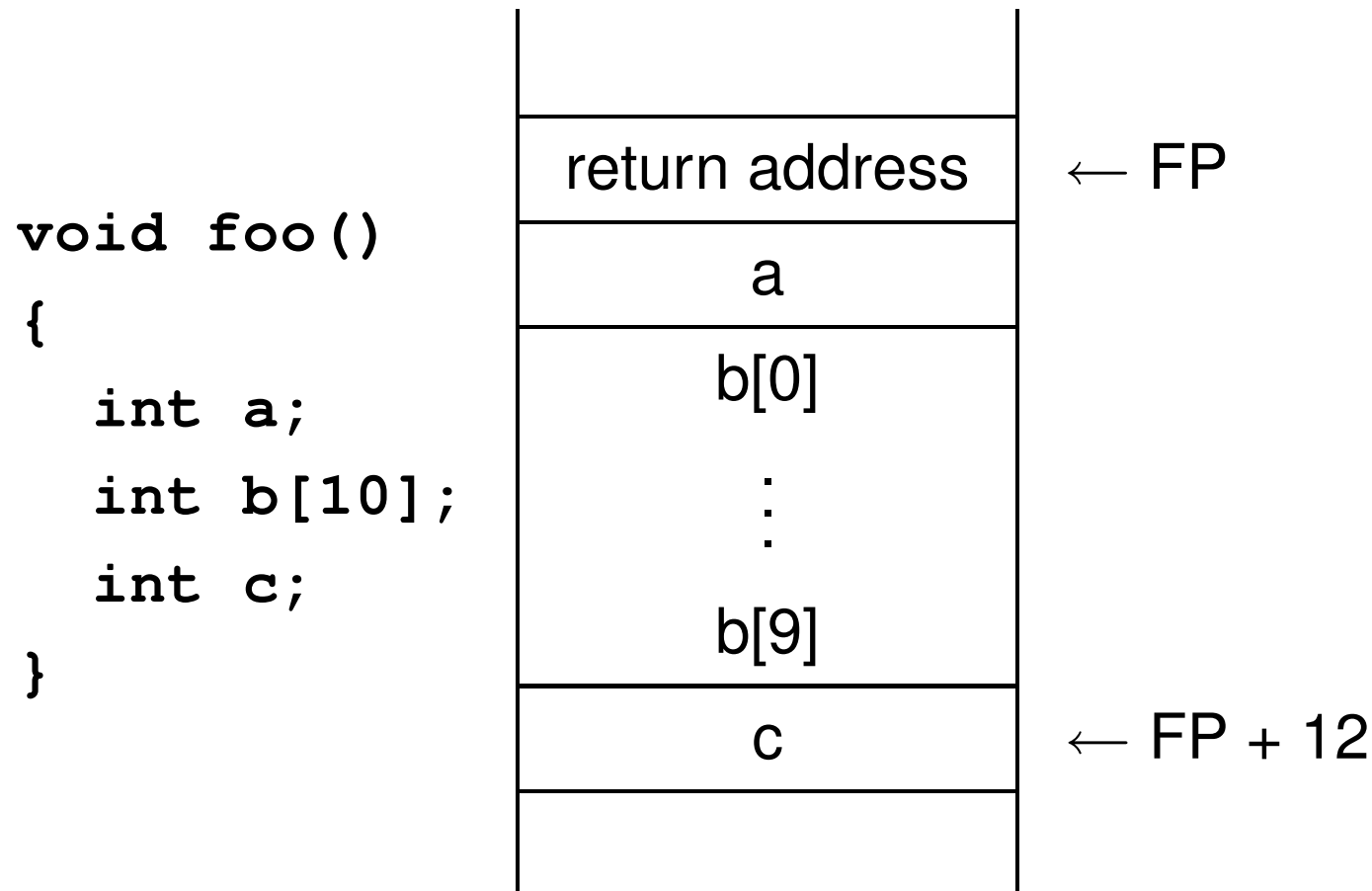
```
int a[10];           /* static */

void foo(int n)
{
    int b[15];       /* stacked */
    int c[n];        /* stacked: tricky */
    int d[];         /* on heap */
    vector<int> e;    /* on heap */

    d = new int[n*2]; /* fixes size */
    e.append(1);      /* may resize */
    e.append(2);      /* may resize */
}
```

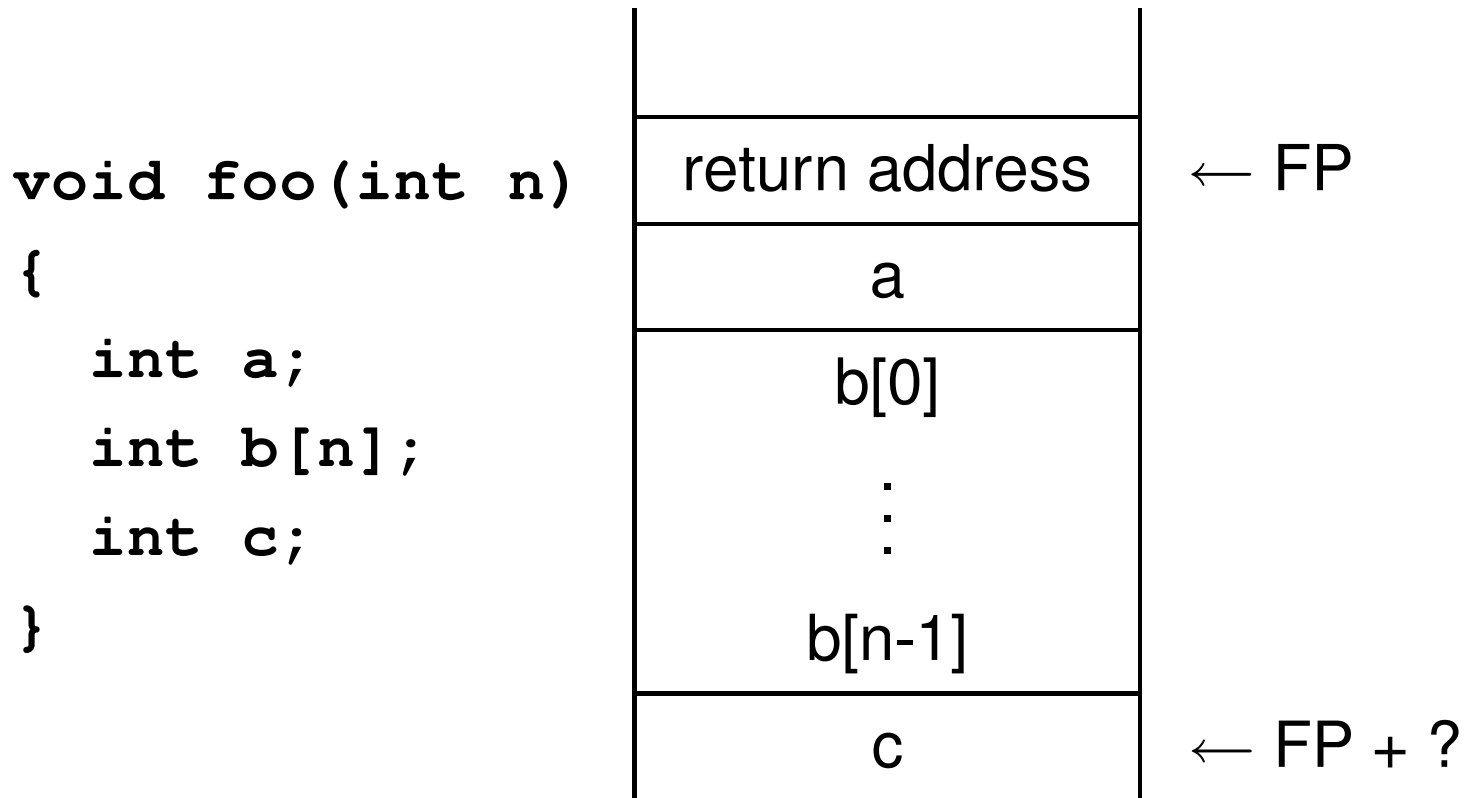
# Allocating Fixed-Size Arrays

Local arrays with fixed size are easy to stack.



# Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

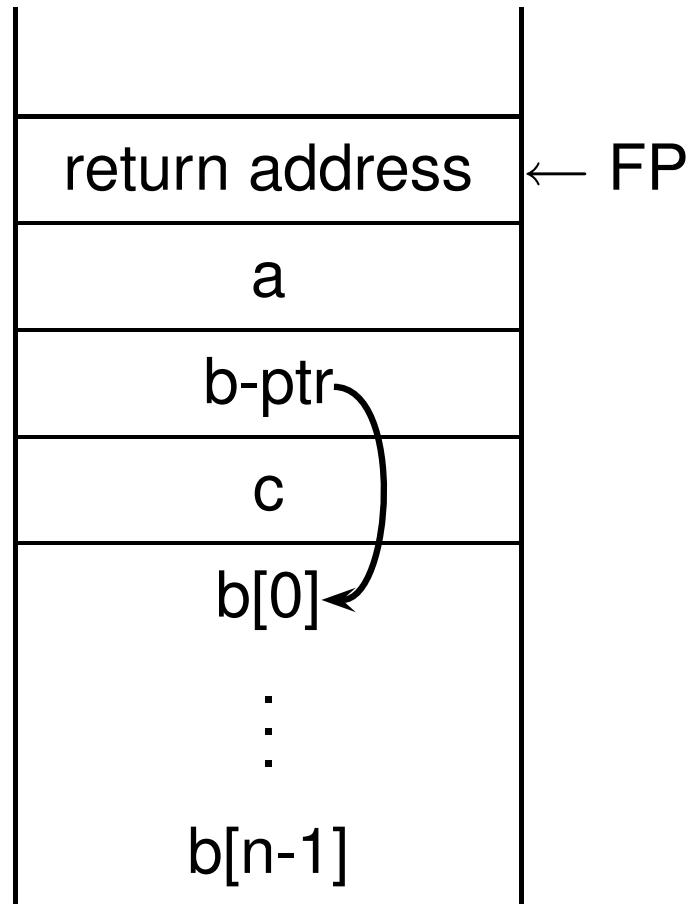


Doesn't work: generated code expects a fixed offset for c.  
Even worse for multi-dimensional arrays.

# Allocating Variable-Sized Arrays

As always:  
add a level of indirection

```
void foo(int n)
{
    int a;
    int b[n];
    int c;
}
```



Variables remain constant offset from frame pointer.

# Static Semantic Analysis

# Static Semantic Analysis

Lexical analysis: Make sure tokens are valid

```
if i 3 "This"           /* valid */  
#a1123                  /* invalid */
```

Syntactic analysis: Makes sure tokens appear in correct order

```
for i := 1 to 5 do 1 + break /* valid */  
if i 3                      /* invalid */
```

Semantic analysis: Makes sure program is consistent

```
let v := 3 in v + 8 end      /* valid */  
let v := "f" in v(3) + v end /* invalid */
```



# Name vs. Structural Equivalence

```
let
  type a = { x: int, y: int }
  type b = { x: int, y: int }
  var i : a := a { x = 1, y = 2 }
  var j : b := b { x = 0, y = 0 }
in
  i := j
end
```

Not legal because **a** and **b** are considered distinct types.

# Name vs. Structural Equivalence

```
let
  type a = { x: int, y: int }
  type b = a
  var i : a := a { x = 1, y = 2 }
  var j : b := b { x = 0, y = 0 }
in
  i := j
end
```

Legal because **b** is an alias for type **a**.

`{ x: int, y: int }` creates a new type, not the `type` keyword.

# Things to Check

Make sure variables and functions are defined.

```
let var i := 10
in i(10,20) /* Error: i is a variable */
end
```

Verify each expression's types are consistent.

```
let var i := 10
      var j := "Hello"
in i + j /* Error: i is int, j is string */
end
```

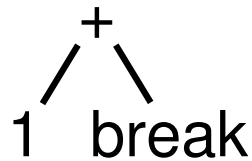
# Things to Check

- Used identifiers must be defined
- Function calls must refer to functions
- Identifier references must be to variables
- The types of operands for unary and binary operators must be consistent.
- The first expression in an `if` and `while` must be a Boolean.
- It must be possible to assign the type on the right side of an assignment to the lvalue on the left.
- ...

# Static Semantic Analysis

Basic paradigm: recursively check AST nodes.

`1 + break`



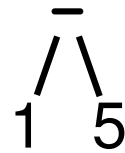
check(+)

check(1) = int

check(break) = void

FAIL: int  $\neq$  void

`1 - 5`



check(-)

check(1) = int

check(5) = int

Types match, return int

Ask yourself: at a particular node type, what must be true?

# Implementing Static Semantics

Recursive walk over the AST.

Analysis of a node returns its type or signals an error.

Implicit “environment” maintains information about what symbols are currently in scope.

**TigerSemant.g** is a tree grammar that does this.

# TigerSemant.g

```
expr returns [Type t]
{ Type a, b, c; t = env.getVoidType(); }
: "nil" { t = env.getNilType(); }
| t=lvalue
| STRING { t = env.getStringType(); }
| NUMBER { t = env.getIntType(); }
| #( NEG a=expr
    { /* Verify expr is an int */
      if ( !(a instanceof Semant.INT) )
        semantError(#expr,
                    "Operand not integer");
      t = env.getIntType();
    } )
```

# Type Classes

```
package Semant;

public abstract class Type {
    public Type actual()
    public boolean coerceTo(Type t)
}

public INT()           // int
public STRING()        // string
public NIL()           // nil
public VOID()          // ()
public NAME(String n) // type a = b
public ARRAY(Type e)   // array of int
public RECORD(String n, Type t, RECORD next)
```



# Type Classes

The NIL type corresponds to the `nil` keyword.

The VOID type corresponds to expressions that return no value.

`()`

`let v := 8 in end`

`if a < 3 then t := 4`

# Type Classes

The RECORD class is a linked list representation of record types.

```
type point = { x: int, y: int }
```

```
new RECORD("x", intType,  
           new RECORD("y", intType, null))
```

# Type Classes

actual() returns the actual type of an alias, e.g.,

```
type a = int
```

```
type b = a
```

```
type c = b
```

c.actual() will return the INT type.

# Type Classes

`coerceTo()` answers the “can this be assigned to” question.

```
type a = {x:int}
```

```
type b = a
```

`nil.coerceTo(a)` is true

`b.coerceTo(a)` is true

`a.coerceTo(nil)` is false

# Environment.java

```
package Semant;

public class Environment {
    public Table vars = new Table();
    public Table types = new Table();
    public INT getIntType()
    public VOID getVoidType()
    public NIL getNilType()
    public STRING getStringType()

    public void enterScope()
    public void leaveScope()
}
```

# Symbol Tables

```
package Semant;  
  
public class Table {  
    public Table()  
    public Object get(String key)  
    public void put(String key, Object value)  
    public void enterScope()  
    public void leaveScope()  
}
```

# Symbol Tables

Operations:

`put (String key, Object value)` inserts a new named object in the table, replacing any existing one in the current scope.

`Object get (String key)` returns the object of the given name, or `null` if there isn't one.

# Symbol Table Scopes

`void enterScope()` pushes a new scope on a stack.

`void leaveScope()` removes the topmost one.

```
Table t = new Table();
t.put("a", new VarEntry(env.getIntType()));
t.put("a", new VarEntry(env.getStringType()));
t.get("a"); // string
t.enterScope();
t.get("a"); // string
t.put("a", new VarEntry(env.getIntType()));
t.get("a"); // int
t.leaveScope();
t.get("a"); // string
```



# Symbol Table Objects

Discriminates between variables and functions.

Stores extra information for each.

```
package Semant;
```

```
public VarEntry(Type t)
```

```
public FunEntry(RECORD f, Type r)
```

RECORD argument represents the function arguments;  
other is the return type.

# Symbol Tables and the Environment

The environment has two symbol tables:

- `types` for types

Objects stored in symbol table are `Types`

- `vars` for variables and functions

Objects are `VarEntryS` and `FunEntryS`.

# Rule for an Identifier

```
lvalue returns [Type t]
{ Type a, b; t = env.getVoidType(); }

: i:ID {
Entry e = (Entry) env.vars.get(i.getText());
if ( e == null )
    semantError(i, i.getText()+" undefined");
if ( !(e instanceof VarEntry) )
    semantError(i, i.getText()+" not variable");
VarEntry v = (VarEntry) e;
t = v.ty;
}
```

# Rule for Let

```
| # ( "let"  
    { env.enterScope(); }  
    # (DECLS (# (DECLS (decl)+ )) * )  
    a=expr  
    {  
        env.leaveScope();  
        t = a;  
    }  
)
```

# Partial rule for Var

```
decl { Type a, b; }
: #( "var" i:ID
    (a=type | "nil" { a = null; } )
    b=expr
    {
    /* Verify a=b if a != null */
    /* Make sure b != nil if a == null */
    env.vars.put(i.getText(), new VarEntry(b))
    }
)
```

# Partial rule for BINOP

```
| #( BINOP a=expr b=expr {
  String op = #expr.getText();
  if ( op.equals("+") || op.equals("-") ||
      op.equals("*") || op.equals("/") ) {
    if (!(a instanceof Semant.INT) ||
        !(b instanceof Semant.INT))
      semantError(#expr, op+" operands not int")
    t = a;
  } else {
    /* Check other operators */
  }
}
)
```