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 Type System: Base Types/Pointers

Base types match typical processor

Typical sizes: 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 Type System: 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 Type System: 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 take 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 */
t.string = "gibberish"; /* overwrites t.f */
**Applications of Variant Records**

A primitive form of polymorphism:

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

- `poly.type == CIRCLE`, use `poly.d.radius`.
- `poly.type == SQUARE`, use `poly.d.size`.
- `poly.type == LINE`, use `poly.d.angle`.

---

**Layout of Records and Unions**

Modern processors have byte-addressable memory.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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

- 16-bit integer: 1 0
- 32-bit integer: 3 2 1 0

---

**C’s Type System: Enumerations**

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

Enumeration constants in the same scope must be unique:

```c
enum days {sun, wed, sat};
enum class {mon, wed}; /* error: mon, wed redefined */
```

---

**Strongly-typed Languages**

Strongly-typed: no run-time type clashes.

C is definitely not strongly-typed:

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

Is Java strongly-typed?

Is Tiger strongly-typed?
Polymorphism

Say you write a sort routine:

```c
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;
            }
}
```

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

```c
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;
            }
}

Fast code, but lots of it.

C++ Templates

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

```c
sort<int>(a, 10);
sort<double>(b, 30);
sort<char *>(c, 20);
```

Faking Polymorphism with Objects

```c
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:

- C: `char i[10];`
- FORTRAN: `character(10) i` (4)
- Ada: `i : array (0..9) of character;` (5)
- Pascal: `var i : array [0 .. 9] of char;` (6)

Array Address Calculation

In C:

```c
char a[10]; /* C */
```

```fortran
character(10) i ! FORTRAN
```

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

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

Allocating Arrays

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

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

```c
d = new int[n*2]; /* fixes size */
```

```c
e.append(1); /* may resize */
```

```c
e.append(2); /* may resize */
```
Allocating Fixed-Size Arrays
Local arrays with fixed size are easy to stack.

```c
void foo()
{
    int a;
    int b[10];
    int c;
}
```

returns address FP
```
a
b[0]
... b[9]
c
```

FP + 12

Allocating Variable-Sized Arrays

Variable-sized local arrays aren't as easy.

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

returns address FP
```
a
b[0]
... b[n-1]
c
```

Even worse for multi-dimensional arrays.

Calling the same function with different arguments is easy.

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

returns address FP
```

```
a
b-ptr
... b[n-1]
c
```

Variables remain constant offset from frame pointer.

Allocating Variable-Sized Arrays

As always:
add a level of indirection

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

returns address FP
```
a
b-ptr
... b[n-1]
c
```

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

Static Semantic Analysis

Lexical analysis: Make sure tokens are valid
```c
if 1 3 "This" /* valid */
#all23 /* invalid */
```
Syntactic analysis: Makes sure tokens appear in correct order
```c
for i := 1 to 5 do 1 + break /* valid */
if i 3 /* invalid */
```
Semantic analysis: Makes sure program is consistent
```c
let v := 3 in v + 8 end /* valid */
let v := "f" in v(3) + v end /* invalid */
```

Tiger's Type System

Array types:
```c
type ia = array of int
type ba = array of bar
```
Record types:
```c
type point = { x : int, y : int,
              name : string }```

Tiger's Type System: Nil

The nil keyword is a stand-in for the null pointer.
Can be assigned to any record, but nothing else.
```c
let
    type rec = { x : int, y : int }
    var a : rec := nil
    in if a = nil then a := rec { x=x10, y=20 } end
```

Is not a valid type
```c
let
    var b := nil /* ERROR */
in end
```

Tiger's Type System: Void

There's an implicit void type:
```c
() a := 3
for i := 1 to 10 do b := b + i
while i < 10 do i := i + 1
```

Can't be assigned to anything, but certain expressions must be void.
```c
a := () /* ERROR */
for i := 1 to 10 do i /* ERROR */
for i := 1 to 10 do ( i ; () ) /* OK */
```
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

- 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 an integer.
- 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  1 - 5
\[ \begin{array}{c}
  \text{check(+) = int} \\
  \text{check(-) = int} \\
  \text{check(break) = int} \\
  \text{check(1) = int} \\
  \text{check(5) = int} \\
  \text{FAIL: int \neq void} \\
\end{array} \]

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.

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)

Programming Assignment 2

Perform static semantic analysis of the Tiger program.

Complete TigerSemant.g, an AST tree walker that verifies the program's types are consistent.

I've written classes for types and symbol tables in a package called Semant.

Your job:
- Finish the rules for checking a program.
- Create test programs that verify your rules.
Type Classes

The NIL type corresponds to the `nil` keyword.
The VOID type corresponds to expressions that return no value.

```pseudocode
() let v := 8 in end
if a < 3 then t := 4
```

Type Classes

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

```pseudocode
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.

```pseudocode
type a = {x:int}
type b = a
nil.coerceTo(a) is true
b.coerceTo(a) is true
a.coerceTo(nil) is false
```

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.

```pseudocode
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 Table Scopes

void enterScope() pushes a new scope on a stack.
void leaveScope() removes the topmost one.

```pseudocode
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.

```pseudocode
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 **VarEntries** and **FunEntries**.

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" |
| (DECLS (#(DECLS (decl)+ )))* |
| a=expr |
| { 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("-")) |
| { |
|    t = a; |
| } |
| else { |
|    /* Check other operators */ |
| } |
```

Static Semantics Assignment

Augment **TigerSemant.g** with rules for every node.

Create test cases to test your code. Should have a test for every error message, every successful rule.

You may use, change, or ignore anything in the **Semant** package.

No Makefiles this time: we will compile it ourselves.

TC.java is a front-end that invokes TigerSemant.

A working scanner and parser are available as .class files.

Feel free to use them or your own.