Structural Equivalence

An alternative notion of type equivalence is structural equivalence (denoted $\equiv_S$). Roughly, two types are structurally equivalent if the two types have the same definition, independent of where the definitions are located. That is, the two types have the same definitional structure.

Formally,

(a) $T \equiv_S T$

(b) Given the declaration

```
Type T = Q;
```

$T \equiv_S Q$

(c) If T and Q are defined using the same type constructor and corresponding parameters in the two definitions are equal or structurally equivalent

then $T \equiv_S Q$

Returning to our previous example,

```
type PackerSalaries = int[100];
type AssemblySizes = int[100];
PackerSalaries  salary;
AssemblySizes  size;
```

salary $\equiv_S$ **size** since both are arrays and **100=100** and **int** $\equiv_S$ **int**.

Which notion of Equivalence do Programming Languages Use?

C and C++ use structural equivalence except for structs and classes (where name equivalence is used). For arrays, size is ignored.

Java uses structural equivalence for scalars. For arrays, it requires name equivalence for the component type, ignoring size. For classes it uses name equivalence except that a subtype may be used where a parent type is expected. Thus given

```
void subr(Object O) { ... };
```

the call

```
subr(new Integer(100));
```

is OK since **`Integer`** is a subclass of **`Object`**.

Automatic Type Conversions

C, C++ and Java also allow various kinds of automatic type conversions.

In C, C++ and Java, a **`float`** will be automatically created from an **`int`**:

```
float f = 10; // No type error
```

Also, an integer type (**`char`**, **`short`**, **`int`**, **`long`**) will be *widened*:

```
int i = 'x';
```

In C and C++ (but not Java), an integer value can also be *narrowed*, possibly with the loss of significant bits:

```
char c = 1000000;
```

Reading Assignment

- Roosta: Sections 13.1, 13.2
- The Scheme Language Definition (linked from class web page)

Lisp & Scheme

Lisp (*Lis*t *P*rocessing Language) is one of the oldest programming languages still in wide use.

It was developed in the late 50s and early 60s by John McCarthy.

Its innovations include:

- Support of symbolic computations.
- A *functional programming style* without emphasis on assignments and side-effects.
- A naturally recursive programming style.
- Dynamic (run-time) type checking.
- Dynamic data structures (lists, binary trees) that grow without limit.

- Automatic garbage collection to manage memory.
- Functions are treated as “first class” values; they may be passed as arguments, returned as result values, stored in data structures, and created during execution.
- A formal semantics (written in Lisp) that defines the meaning of all valid programs.
- An Integrated Programming Environment to create, edit and test Lisp programs.

Scheme

Scheme is a recent dialect of Lisp.

It uses lexical (static) scoping.

It supports true first-class functions.

It provides program-level access to control flow via *continuation* functions.

Atomic (Primitive) Data Types

Symbols:

Essentially the same form as identifiers. Similar to enumeration values in C and C++.

Very flexible in structure; essentially any sequence of printable characters is allowed; anything that starts a valid number (except + or -) *may not* start a symbol.

Valid symbols include:

```
abc  hello-world   +  <=!
```

Integers:

Any sequence of digits, optionally prefixed with a + or -. Usually unlimited in length.

Reals:

A floating point number in a decimal format (**`123.456`**) or in exponential format (**`1.23e45`**). A leading sign and a signed exponent are allowed (**`-12.3`**, **`10.0e-20`**).

Rationals:

Rational numbers of the form integer/integer (e.g., **`1/3`** or **`9/7`**) with an optional leading sign (**`-1/2`**, **`+7/8`**).

Complex:

Complex numbers of the form num+num i or num-num i, where num is an integer or real number. Example include **`1+3i`**, **`-1.5-2.5i`**, **`0+1i`**).

String:

A sequence of characters delimited by double quotes. Double quotes and backslashes must be escaped using a backslash. For example

```
"Hello World" "\"Wow!\""
```

Character:

A single character prefixed by `#\`. For example, `#\a`, `#\0`, `#\\`, `#\#`. Two special characters are `#\space` and `#\newline`.

Boolean:

True is represented as `#t` and false is represented as `#f`.