Eager Parameter Evaluation

Sometimes we want parameters evaluated *eagerly*—as soon as they are known.

Consider a sorting routine that breaks an array in half, sorts each half, and then merges together the two sorted halves (this is a *merge sort*).

In outline form it is:

```
sort(inputArray) {
  ...
merge(sort(leftHalf(inputArray)),
      sort(rightHalf(inputArray)));}
```

This definition lends itself nicely to parallel evaluation: The two halves of an input array can be sorted in parallel. Each of these two halves can

again be split in two, allowing parallel sorting of four quarter-sized arrays, then leading to 8 sorts of 1/8 sized arrays, etc.

But,
to make this all work, the two parameters to merge must be evaluated *eagerly*, rather than in sequence.

Type Equivalence

Programming languages use types to describe the values a data object may hold and the operations that may be performed.

By checking the types of values, potential errors in expressions, assignments and calls may be automatically detected. For example, type checking tells us that

```
123 + "123"
```

is illegal because addition is not defined for an integer, string combination.

Type checking is usually done at compile-time; this is *static typing*.

Type-checking may also be done at run-time; this is *dynamic typing*.

A program is *type-safe* if it is impossible to apply an operation to a value of the wrong type. In a type-safe language, plus is never told to add an integer to a string, because its definition does not allow that combination of operands. In type-safe programs an operator can still see an illegal value (e.g., a division by zero), but it can't see operands of the wrong type.

A *strongly-typed* programming language forbids the execution of type-unsafe programs.

Weakly-typed programming languages allow the execution of potentially type-unsafe programs.

The question reduces to whether the programming language allows programmers to “break” the type rules, either knowingly or unknowingly.

Java is strongly typed; type errors preclude execution. C and C++ are weakly typed; you can break the rules if you wish. For example:

```
int i;  int* p;
p = (int *) i * i;
```

Now **p** may be used as an integer pointer though multiplication need not produce valid integer pointers.

If we are going to do type checking in a program, we must decide whether two types, T1 and T2 are equivalent; that is, whether they be used interchangeably.

There are two major approaches to type equivalence:

Name Equivalence:

Two types are equivalent if and only if they refer to exactly the same type declaration.

For example,

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

Is

```
sal = size;
```

allowed?

Using name equivalence, *no*. That is, **salary** $\not\equiv_N$ **size** since these two variables have different type declarations (that happen to be identical in structure).

Formally, we define $\equiv_N$ (name type equivalence) as:

(a) $T \equiv_N T$

(b) Given the declaration

```
Type T1 = T2;
```

$T1 \equiv_N T2$

We treat anonymous types (types not given a name) as an abbreviation for an implicit declaration of a new and unique type name.

Thus

```
int  A[10];
```

is an abbreviation for

```
Type T_new = int[10];
T_new  A;
```

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

- An Introduction to Scheme for C Programmers
  (linked from class web page)
- 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**.

Binary Trees

Binary trees are also called *S-Expressions* in Lisp and Scheme.

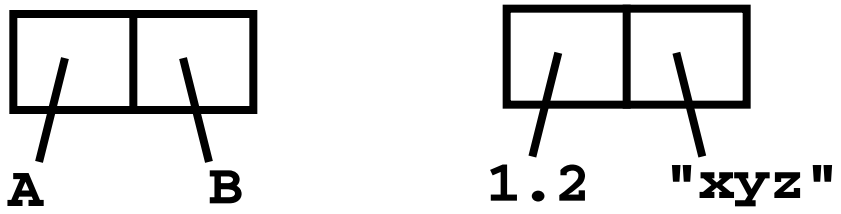
They are of the form

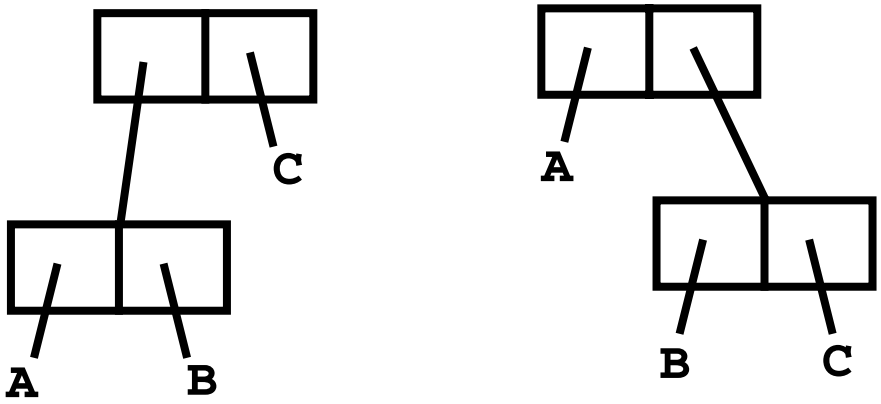
(item . item)

where item is any atomic value or any S-Expression. For example:

```
( A . B )
(1.2  .  "xyz" )
( (A . B ) . C )
( A . (B . C ) )
```

S-Expressions are linearizations of binary trees:





S-Expressions are built and accessed using the predefined functions *cons*, *car* and *cdr*.

cons builds a new S-Expression from two S-Expressions that represent the left and right children.

cons(E1,E2) = (E1 . E2)

car returns are left subtree of an S-Expression.

car (E1 . E2) = E1

cdr returns are right subtree of an S-Expression.

cdr (E1 . E2) = E2

Lists

In Lisp and Scheme lists are a special, widely-used form of S-Expressions.

() represents the empty or null list

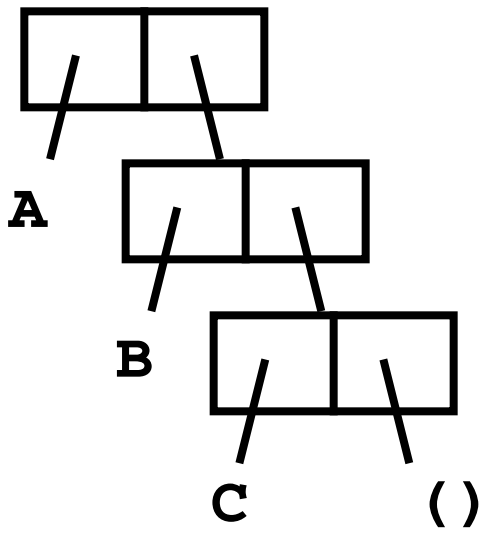
(A) is the list containing **A**.
By definition, **(A)** ≡ **(A . ())**

(A B) represents the list containing **A** and **B**. By definition,
(A B) ≡ **(A . (B . ()))**

In general, **(A B C ... Z)** ≡

(A . (B . (C . ... (Z . ()) ...)))

(A B C)≡



Function Calls

In List and Scheme, function calls are represented as lists.

`(A B C)` means:

Evaluate **`A`** (to a function)

Evaluate **`B`** and **`C`** (as parameters)

Call **`A`** with **`B`** and **`C`** as its parameters

Use the value returned by the call as the "meaning" of **`(A B C)`**.

`cons`, **`car`** and **`cdr`** are predefined symbols bound to built-in functions that build and access lists and S-Expressions.

Literals (of type integer, real, rational, complex, string, character and boolean) evaluate to themselves.

For example (⇒ means "evaluates to")

`(cons 1 2)` ⇒ **`(1 . 2)`**

`(cons 1 () )` ⇒ **`(1)`**

`(car (cons 1 2))` ⇒ **`1`**

`(cdr (cons 1 ()))` ⇒ **`()`**

But,

`(car (1 2))` fails during execution!

Why?

The expression **`(1 2)`** looks like a call, but **`1`** isn't a function! We need some way to "quote" symbols and lists we *don't* want evaluated.

`(quote arg)`

is a special function that returns its argument *unevaluated*.

Thus **`(quote (1 2))`** doesn't try to evaluate the list **`(1 2)`**; it just returns it.

Since quotation is so often used, it may be abbreviated using a single quote. That is

`(quote arg)` ≡ **`'arg`**

Thus

`(car '(a b c))` ⇒ **`a`**

`(cdr '( (A) (B) (C)))` ⇒
`( (B) (C) )`

`(cons 'a '1)` ⇒ **`(a . 1)`**

But,

`('cdr '(A B))` fails!

Why?

User-defined Functions

The list

```
(lambda (args) (body))
```

evaluates to a function with `(args)` as its argument list and `(body)` as the function body.

No quotes are needed for `(args)` or `(body)`.

Thus

`(lambda (x) (+ x 1))` evaluates to the increment function.

Similarly,

```
((lambda (x) (+ x 1))  10) ⇒
  11
```

We can bind values and functions to global symbols using the **`define`** function.

The general form is

```
(define id object)
```

`id` is not evaluated but **`object`** is. **`id`** is bound to the value object evaluates to.

For example,

```
(define pi  3.1415926535)

(define plus1
   (lambda (x) (+ x 1)) )
(define pi*2  (* pi 2))
```

Once a symbol is defined, it evaluates to the value it is bound to:

`(plus1 12)` $\Rightarrow$ **`13`**

Since functions are frequently defined, we may abbreviate

```
(define id
  (lambda (args) (body)) )
```

as

```
(define (id args) (body) )
```

Thus

```
(define (plus1 x) (+ x 1))
```