# Thoughts on Assignment 5
# PL Category: Lisp-Family PLs
# Introduction to Scheme

CS 331 Programming Languages

Lecture Slides

Wednesday, March 19, 2025

Glenn G. Chappell
Department of Computer Science
University of Alaska Fairbanks
ggchappell@alaska.edu

# Thoughts on Assignment 5

# Thoughts on Assignment 5
# Overview

Assignment 5 is mostly programming in Haskell.

There are three exercises.

A. Run some code in Prolog—another secret message.
B. Write a module in Haskell, containing five specified things—mostly involving functions that work with lists.
C. Write a stand-alone program in Haskell, of the kind that might be assigned in an introductory programming class.

# Thoughts on Assignment 5
# Exercise B [1/4]

In Exercise B, you are to write a Haskell module, in file `PA5.hs`.

There is a test program (`pa5_test.hs`) and a skeleton version of the file you are to write (`PA5.hs`). The skeleton file will compile and execute with the test program, but it will not pass the tests.

The skeleton file includes type annotations for each of the entities you are to write in Exercise B. Furthermore, these type annotations are correct; you will not need to change them.

I suggest that you begin with the posted version of `PA5.hs`, and that you leave the type annotations alone.

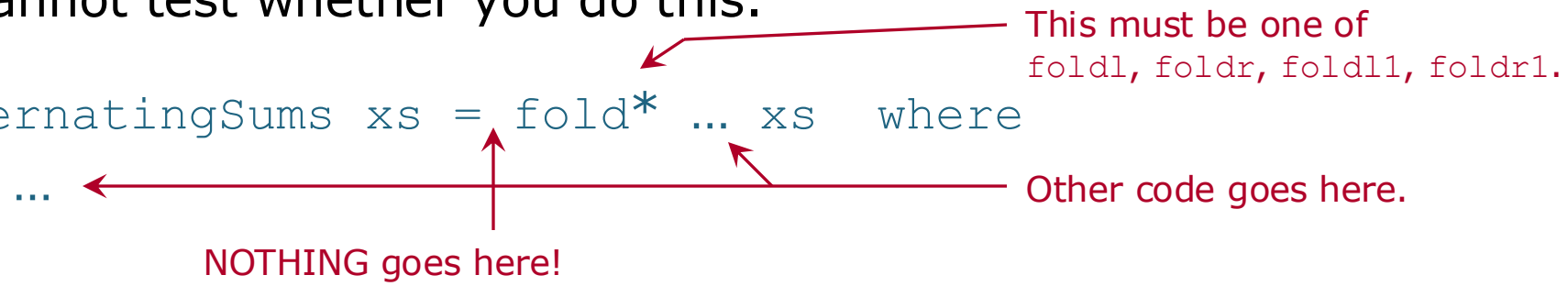
# Thoughts on Assignment 5
# Exercise B [2/4]

One requirement cannot be checked by the test program.

You are to write a function `alternatingSums`. This is given a list of numbers; it returns a 2-item tuple containing the sum of the even-index items of the given list and the sum of the odd-index items (zero-based indexing).

```
> alternatingSums [1,10,2,20,3,30]
(6,60)
```

`alternatingSums` is required to be be written as a **fold**, but I cannot test whether you do this.



# Thoughts on Assignment 5
# Exercise B [3/4]

How do we write a fold? Consider `foldl`:

```
foldl f z xs
```

`xs` is the list we are operating on.

`z` is the starting value. It is what is returned if `xs` is empty.

`f` is a 2-parameter function that takes a partial result (what we get from folding all but the last item) and a new item. It returns the result we want for the whole list.

For example, the following returns the sum of all list items:

```
foldl (+) 0 [3,6,2,5,3,7]
```

# Thoughts on Assignment 5
# Exercise B [4/4]

`foldr` is the same as `foldl`, except that it works from the other direction.

```
foldr f z xs
```

`xs` is the list we are operating on.

`z` is the starting value. It is what is returned if `xs` is empty.

`f` is a 2-parameter function that takes a new item and a partial result (what we get from folding all but the *first* item). It returns the result we want for the whole list.

# Thoughts on Assignment 5
# Exercise C [1/3]

In Exercise C, you are to write a stand-alone Haskell program that inputs numbers and prints their sum.

Your code will need to do some I/O tasks repeatedly, many times. The most straightforward way to do this is to write a function that does the I/O and calls itself tail recursively.

Such a function only does things that need to happen repeatedly. For example, if you need to do X once and then do Y over and over again, then write:

- A function that does Y and calls itself.
- A function that does X and then calls the other function.

# Thoughts on Assignment 5
# Exercise C [2/3]

In an I/O do-block, everything is either

- an expression whose value is an I/O action,
- the above preceded by "*variable* `<-`", or
- "`let` *variable* `=` …".

You will almost certainly need an if-then-else inside a do-block. Each branch of the if-then-else must evaluate to an I/O action.

If you have more than one I/O task to perform in one of the branches, then combine these into a single I/O action, using `do`.

What if you have nothing to do in one of the branches? You still must have both branches (in Haskell, the `else` is not optional). And each must evaluate to an I/O action. You can create an I/O action that does nothing by using "`return ()`".

See `squarenums.hs` and `bestnum.hs` for example code.

# Thoughts on Assignment 5
# Exercise C [3/3]

We have not covered error checking on I/O and conversion from `String`. So your program in Exercise C is *not* required to be fully **robust**—that is, able to deal gracefully with any input.

In other words, your program is allowed to crash if the user types an illegal value.

# Unit Overview
# The Scheme Programming Language

Topics

- ✓ PL feature: identifiers & values
- ✓ PL feature: reflection
- PL category: Lisp-family PLs
- Introduction to Scheme
- Scheme: basics
- Scheme: evaluation
- Scheme: data
- Scheme: macros

# Review

## PL Feature: Reflection

**Reflection** in a computer program refers to the ability of the program to deal with its own code.

C and C++ offer essentially no support for reflection. Haskell is similar.

Lisp-family programming languages (Common Lisp, Scheme, EMACS Lisp, Clojure, Logo, and others) form the gold standard for reflection support. Code can easily be made available to a program in the form of an abstract syntax tree, which can be transformed arbitrarily and then executed.

# PL Category: Lisp-Family PLs

# PL Category: Lisp-Family PLs
# Background [1/4]

In the late 1950s, MIT professor John McCarthy developed a mathematical formalism for describing computation.

His most important notation was the **Symbolic Expression**, or **S-expression**. An S-expression is either an **atom** (basically a word), **nil** (empty pair of parentheses), or a **pair** (S-expression dot S-expression in parentheses). Using pairs, we can construct large S-expressions from small pieces.

| | |
|---|---|
| **Atom** | ABC |
| **Nil** | () |
| **Pair** | ( *???* . *???* ) |

(THIS . (IS . (AN . ((S . (EXPRESSION . ())) . ()))))

A shorter form uses a parenthesized list of space-separated items. Something like "(A . (B . (C . ())))" is written as "(A B C)".

(THIS IS AN (S EXPRESSION)) Same as above

# PL Category: Lisp-Family PLs
# Background [2/4]

Steve Russell, at that time a Dartmouth student, became aware of McCarthy's work. He observed that one of the operations of the formalism—*eval*—if implemented as a computer program, would be an interpreter for a programming language.

In 1958, Russell wrote such an implementation in machine language on an IBM 704. The resulting programming language became known as **Lisp**, for LISt Processor.

Lisp and associated PLs are noted for their excellent support for reflection. Code and data are stored in the same structures (binary trees representing S-expressions). Modification of existing code, followed by execution of the modified code, is common.

# PL Category: Lisp-Family PLs
# Background [3/4]

Lisp caught on rapidly in the Artificial Intelligence research community. By the 1970s several dialects were in use.

In the late 1960s, a Lisp dialect called **Logo** was released. This was aimed at teaching programming concepts.

The mid-1970s saw the development of EMACS (originally Editor MACroS), a Lisp-scriptable text editor. A descendant continues to be actively developed by the GNU project; it is widely used.

**Lisp machines**, computers aimed at running Lisp, were sold in the 1980s. They are no longer made, but emulators are available.

In 1984, a unified Lisp standard was produced: **Common Lisp**. An ANSI standard was published in 1994.

# PL Category: Lisp-Family PLs
# Background [4/4]

A number of important programming-language concepts had their first major implementations in Lisp-family PLs: recursion, tree structures, closures, dynamic typing, higher-order functions, encapsulated loops like map, filter, and zip, garbage collection, and REPLs.

Interest in Lisp died down after the 1980s. But Lisp appears to have enjoyed something of a resurgence in the last couple of decades.

Of particular interest is **Clojure** (pronounced "closure") a Lisp-family PL for the Java Virtual Machine (JVM). Originally written by Rich Hickey in 2007, Clojure continues to be actively developed.

# PL Category: Lisp-Family PLs
# Typical Characteristics [1/2]

Characteristics of typical Lisp-family PLs:

- Simple syntax based on the S-expression. A program is a list or sequence of lists. The first item of a list is a function (usually "**procedure**" in the Lisp world); the rest are its arguments.

For example, `(a + 2) * -b` in a typical Lisp-family PL:

```
(* (+ a 2) (- b))
```

Where have we seen the above format for an expression before?

Tweak the notation. Replace parentheses with braces, separate list items by commas, and place each atom in double quotes:

```
{"*", {"+", "a", "2"}, {"-", "b"}}
```

Q. Where have we seen this before?

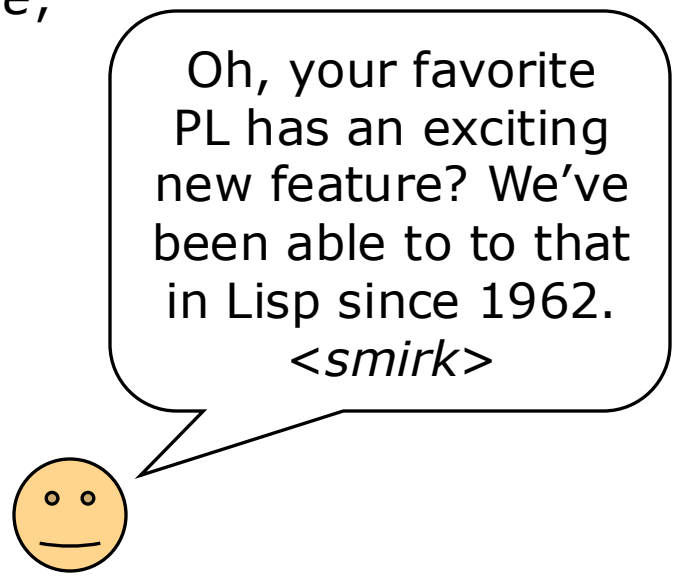
A. This was our first stab at an AST representation in Lua.

**Lisp source code is a direct representation of its own AST!**

# PL Category: Lisp-Family PLs
# Typical Characteristics [2/2]

Characteristics of typical Lisp-family PLs (cont'd):

- Support for reflection is excellent. Source-code syntax and storage format are those of the PL's primary data structure. The PL supports **macros**: specified code transformations.
- Typing is dynamic, implicit, and structural (duck typing).
- There is good support for functional programming: functions are first-class, and higher-order functions are supported.
- But Lisp-family PLs are typically not *pure*; mutable data is usually allowed.
- New constructions can be defined: flow of control, definitions, etc.
- Execution can be interactive (REPL) or via previously compiled code.
- Accomplished Lisp programmers tend to be insufferably fond of Lisp.
  (But maybe they're onto something.)



# Introduction to Scheme

# Introduction to Scheme
# History [1/2]

As with many PLs, the early history of Lisp was one of ever-increasing complexity. As a result, the Common Lisp standard is huge, including sophisticated exception handling, an object system, and many other features.

Perhaps as a reaction to this tendency, a Lisp-family PL called **Scheme** was created at the MIT AI Lab around 1970, by Guy Steele and Gerald Sussman.

In contrast to Common Lisp, Scheme follows a minimalist design philosophy, with a small, simple core and versatile tools for extending the PL with new constructions.

Scheme's minimalism probably will not be impressive, as Lua is rather more minimalist. But compared to Common Lisp, Scheme is *tiny*.

# Introduction to Scheme
# History [2/2]

Scheme differs from traditional Lisp in a number of significant ways. Thus, while some say Scheme is a dialect of Lisp, others emphatically deny this. But Scheme clearly belongs in the Lisp *family* of PLs.

Scheme has been standardized in a series of documents: a standard issued by IEEE in 1991 and a series of reports by the Scheme Language Steering Committee. The most recent was released in 2013: the Revised$^7$ Report on the Algorithmic Language Scheme (R7RS).

In 1994 the Rice University Programming Languages Team released **PLT Scheme**. In 2010 this was renamed **Racket**. Distributed with Racket is a simple IDE called **DrRacket**, which runs on all major platforms.

We will use DrRacket, but not the Racket PL. DrRacket also supports standard Scheme.

# Characteristics — Introduction

Scheme is a Lisp-family PL with a minimalist design philosophy.

Scheme code consists of parenthesized lists, which may contain atoms or other lists. List items are separated by space; blanks and newlines between list items are treated the same.

```
(define (hello-world)
  (begin
    (display "Hello, world!")
    (newline)
    )
  )
```

When a list is evaluated, the value of the first item is usually a **procedure** (think "function"); other items are its arguments.

# Introduction to Scheme
# Characteristics — Type System [1/2]

The type system of Scheme is similar to that of Lua.

- Typing is dynamic.
- Typing is implicit. Type annotations are generally not used.
- Type checking is structural. Duck typing is used.
- There is a high level of type safety: operations on invalid types are not allowed, and implicit type conversions are rare.
- There is a fixed set of types.

Lua's fixed set of types includes only 8 types, while Scheme has something like 36. We look at some of these next.

# Characteristics — Type System [2/2]

Two heavily used types are **pair** and **null**, which are used to construct lists, as in S-expressions.

Values of all other types are **atoms**. Here are a few of these:

- Booleans. Literals are `#t` (*true*) and `#f` (*false*).
- Strings. Literals use double quotes: `"This is a string."`
- Characters. Here is the 'a' character literal: `#\a`
- Symbols. A **symbol** is an identifier: `abc  !@a=  a-long-symbol?` In Scheme, symbols are not just names of things; they are *things*. In other words, Scheme has first-class identifiers.
- Numbers. There are seven number types, including arbitrarily large integers (like Haskell's `Integer` and Python's `int`), floating-point numbers, exact rational numbers, and complex numbers.
- Procedures. A **procedure** is what we would call a first-class function. A procedure may be bound to a name (a symbol), or it may be unnamed.

Scheme has no special syntax for flow of control.

Here is some Lua code and more or less equivalent Scheme code.

**Lua**

```
if x == 3 then
    io.write("three")
else
    io.write("other")
end
```

**Scheme**

```
(if (= x 3)
    (display "three")
    (display "other")
    )
```

Scheme uses the same kind of syntax for flow of control (`if`), operators (`=`), and regular function calls (`display`). In Lua—and most other PLs—different syntax is used for these three.

## Characteristics — Miscellaneous

As with Lua, Scheme local variables are lexically scoped. Scheme globals have dynamic scope.

Scheme has good support for functional programming. It might be called a functional PL. But it is not a *pure* functional PL, as it it does allow for mutable data.

Like all Lisp-family PLs, the syntax and storage format of code is the same as that of the programming language's primary data structure. It is natural to manipulate existing code. Transformed code can be executed as part of the same program. Thus, Scheme has excellent support for reflection.

# Introduction to Scheme
# Build & Execution

The standard filename suffix for Scheme source files is `.scm`.

Scheme allows for both interactive execution and compilation to a native code executable file. We will not be doing the latter.

We will execute Scheme using the **DrRacket** IDE.

- The upper part of the DrRacket window, which it calls *definitions*, is a source-code editor, with the usual open-save interface. This semester, the first code line in this part must always be as follows:

```
#lang scheme
```

- Clicking "Run" executes the code in the editor. Symbols defined in this code may then be used in the lower part, which is a REPL.
- After clicking "Run", type Scheme code in the REPL to execute it.

# Introduction to Scheme
# Some Programming [1/2]

TO DO

- Try out interactive Scheme in DrRacket.
- Write a hello-world program in Scheme and execute it.

*Done. See* `hello.scm`.

# Introduction to Scheme
# Some Programming [2/2]

I have written a Scheme program that computes and prints Fibonacci numbers: `fibo.scm`.

TO DO

- Run `fibo.scm`.

*See* `fibo.scm`.