Department of Computer Science and Engineering
University of California at San Diego

CSE 230
Winter 1998

CSE 230: Principles of Programming Languages

This course is an introduction to the most important concepts underlying modern programming language design, and to some theoretical underpinnings of programming languages.

The topics covered include functional programming and the ML language, object-oriented programming and the Java language, type systems, the lambda calculus, and the elements of denotational semantics.

Lectures:

Tuesdays and Thursdays 11:10am to 12:30pm, in Warren Lecture Hall room 2111.

Lecture notes will be made available online through the class web page at `http://www-cse.ucsd.edu/classes/wi98/cse255/`.

Instructor: Charles Elkan, Associate Professor.

Office: APM 5131 until further notice.
Office hours: Mondays and Wednesdays 2pm to 3pm, or by appointment.
Telephone: 534-8897
Email: `elkan@cs.ucsd.edu`
Secretary: Lisa Bodecker, APM 3141, 822-0702

References:

Four textbooks are recommended, but none are required. The books are

- *Elements of Functional Programming* by Chris Reade (Addison Wesley),
- *Elements of ML Programming* by Jeffrey Ullman (Prentice-Hall),
- *Principles of Programming Languages* by R. D. Tennent (Prentice-Hall), and
- *The Java Programming Language* by Ken Arnold and James Gosling (Addison Wesley).

These books should be easily available for borrowing from students who took CSE230 in previous years.

Assignments:

There will be nine weekly problem sets, most including some programming work. Each assignment will be given out on a Tuesday, and due the following Tuesday.

You are encouraged to discuss the problems and assignments with other people in the class, and to work in groups, but each person must hand in solutions written up independently.

Given that no teaching assistant is available for this course, each assignment will be graded by a team of two students with guidance from the instructor. To choose which assignment you will grade, and with whom, please fill out and return the grading form distributed today.

Examinations:

There will be an in-class open book final examination. There will be no midterm examination.

Computer resources:

Information on where to find ML is below. Java has been installed on the APE lab machines.

For this course you should preferentially use computers in the Advanced Programming Environment (APE) lab, located in APM 6426. The APE lab web page is `http://www-cse.ucsd.edu/cse/restricted/APE_lab.html`. You should not use `beowulf` or other non-instructional servers.

If you are not a CSE grad student, then send email to `csehelp@cs.ucsd.edu` to request access to APE lab resources.

Department of Computer Science and Engineering
University of California at San Diego

CSE 230
Winter 1998

Schedule of Topics

The page references below are to *Elements of Functional Programming* by Chris Reade, except the denotational semantics references are to *Principles of Programming Languages* by R. D. Tennent. Note that a few topics are covered in neither book.

date	*topic*	*pages*
Jan 6	Language support for high level programming	1–11
Jan 7	The basics of ML	17–33, 80–91
Jan 13	Typing and polymorphism	12–17, 49–54
Jan 15	Higher-order functions	54–70, 91–99
Jan 20	Abstract syntax	202–204, 149–169
Jan 22	Structural and well-founded induction	33–37, 111–115, 182–188
Jan 27	Closures and scoping	67–69
Jan 29	Semantics of parameter passing	
Feb 3	Semantics of expression evaluation	266–284
Feb 5	Abstract data types	226–232, 254–260
Feb 10	Object-oriented programming	
Feb 12	Java and the future of programming	
Feb 17	Semantics of objects	
Feb 19	Lambda calculus basics	415–421
Feb 24	Programming in the lambda calculus	436–439
Feb 26	Recursion in the lambda calculus	421–428
Mar 3	Denotational semantics basics	T211–219
Mar 5	More denotational semantics	T220–242
Mar 10	Domain theory basics	T35–41, 343–355
Mar 12	TO BE DECIDED	

Department of Computer Science and Engineering | CSE 230
University of California at San Diego | Winter 1998

The Basics of Using ML

The ML system we have is *Standard ML of New Jersey*. The system is installed in the directory `/net/bonzo/class/development/elkan/cse230/sml-0.93/`. ML runs under SunOS 4.1.X and under Solaris. Manuals and example programs are available in the `doc` subdirectory, and the binary is in `bin/sml`. The best general reference is the book *Elements of ML Programming* by Jeffrey Ullman.

When you launch Standard ML of New Jersey, this puts you into the interactive system. The top level prompt is a hyphen `-`, and the secondary prompt which is printed when input is incomplete is an equals sign `=`. Input to the top level interpreter (i.e. declarations and expressions) must be terminated by a semicolon and carriage return before the system will evaluate it. If you get the secondary prompt when you do not expect it, typing `;` will often complete your input.

When input is complete, the system then prints out a response indicating the effect of the evaluation. Expressions are treated as implicit declarations of a standard variable named `it`. For example,

```
- 3; <return>               user input appears after the prompt
val it = 3 : int            this is the system response
```

This means that the value of the last top level expression evaluated can be referred to using the name `it`.

Typing the interrupt character `control-C` will interrupt the compiler and return you to top level, unless some function is catching the Interrupt exception, which is a dangerous thing to do. Typing `control-D` (EOF) at top level will cause an exit to the shell, or the parent process from which sml was run.

To execute system commands, you must first open the `System` package by typing

```
- open System;
```

The signatures of all the functions made available by this package are displayed. One is the function `system` of type `string->unit`. This spawns a process to execute its argument string as a shell command. To find out what the current directory is, for example, evaluate the expression

```
- system "pwd";
```

This will cause the current directory to be printed out. To change the current working directory of ML use the function `cd` of type `string -> unit`, whose argument should be a path name denoting a directory.

The operator `use` of type `string->unit` interprets its argument as a Unix file name relative to the current directory and loads the text from that file as though it had been typed in. `use` should normally be executed at top level, but the loaded files can also contain calls of use to recursively load other files. So if you have ML code in file `hw`, you can execute it with

```
- use "hw";
```

If you are using multiple windows, you can edit `hw` in a second window.

If you do not have access to multiple windows, try the following. Create a file called (say) `start` with the following contents:

```
open System;
fun vi f = system ("vi " ^ f);
```

Inside sml enter

```
- use "start";
```

You can then edit the file `hw` without exiting sml by typing

```
- vi "hw";
```

The error messages produced by the compiler are not always as helpful as they should be, and there are often too many of them. The compiler attempts to recover from syntactic and type errors so that it can detect as many errors as possible during a compilation. Unfortunately, it is not very graceful in recovery, and the process can cause numerous spurious secondary error messages.

When compiling files, the error messages include a line number. For simple syntactic errors this line number is often accurate or off by just one line. For other classes of errors, including type errors, the line number may not be very useful, since it will often just indicate the end of the declaration containing the error, and this declaration can be quite large.

There are a number of different forms of type error message, and it may require some practice before you become adept at interpreting them. The most common form indicates a mismatch between the type of a function (or operator) and its argument (or operand). A representation of the offending expression is usually included, but this is an image of the internal abstract syntax for the expression and may differ significantly from the original source code. For instance, an `if...then...else...` expression is represented internally as a case expression over a boolean value: `case ... of true => ... | false => ...`