Course Info

Course

- COS 320 - Compiling Techniques
- `http://www.cs.princeton.edu/ dpw`
- Room: Friend 008

Instructor

- Prof. David Walker
- CS 412, 258-7654
- dpw@cs.princeton.edu
- Office Hours: after each class

Course Info

Teaching Assistant

- Jay Ligatti
- CS 418a, 258-1763
- Office Hours: Fridays, 3:30-4:30

Books

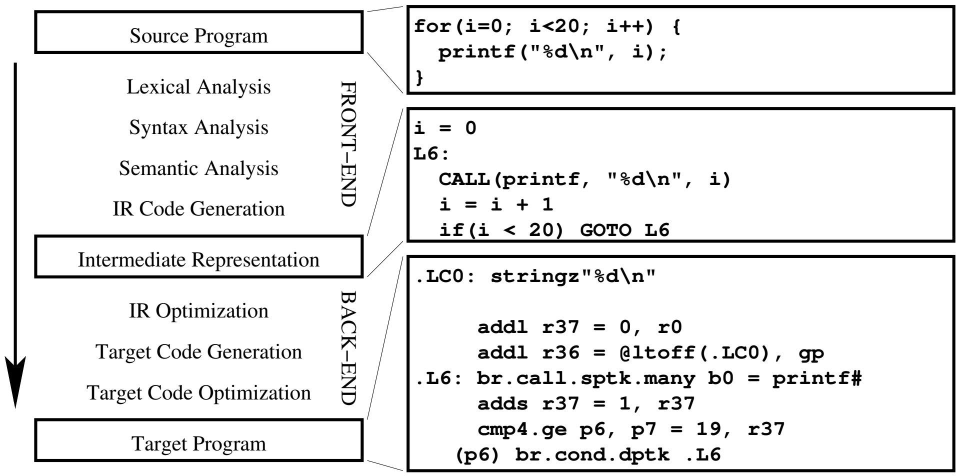
- Jeffrey D. Ullman, Elements of ML Programming, second edition, Prentice Hall.
- Andrew W. Appel, Modern Compiler Implementation in ML. Cambridge University Press, 1998.

What is a Compiler?

A *compiler* is a program that takes a program written in a *source language* and translates it into a functionally equivalent program in a *target language*.

- Source Languages: C, C++, ML, Visual Basic, Java, Pascal, Latex, ...
- Target Languages: x86 Assembly, Java Virtual Machine Language, C, ...
- Compiler can also:
  - Report errors in source
  - Warn of potential problems in source
  - Optimize program

What is a Compiler?



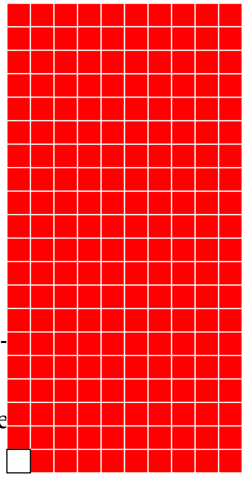
Why Learn About Compilers?

Almost all programs you ever write will be compiled.

Linux

- C = 2,558,100 lines
- x86 assembly = 12,164 lines

99.5% of Linux source goes through a compiler!



Compilers teach us about:

- Programming Languages and how to write better programs
- Computer Architectures and what requirements we have them

Why Learn About Compilers?

Compilers are everywhere.

- C → Assembly
- Assembly → Machine Code
- PHP Server Side Scripts → HTML
- SQL Query → Tables
- Web Search Engines → Matches
- Latex → PostScript
- PostScript → Paper
- Hardware Description Language → Circuit
- SPAM → /dev/null

Why Learn About Compilers?

Techniques are generally applicable.

- IBM developed the first FORTRAN compiler in 1957
- 18 person-years of effort
- This course will show you how to do it in less than a week!

Apply this power

- Write compilers to simplify tedious tasks.
  - Example: Writing an architectural simulator is a pain in the...
  - Use compilers to write them for you! (Liberty Simulation Environment)
  - Example: Writing a compiler is a pain in the...
  - Use compilers to write them for you! (Lex and Yacc)

Assignment 0

1. On the sheet circulating
   - Full Name
   - Email Address
   - Major/Year
   - ML experience
   - Compiler experience
2. Sign up for the course mailing list by sending the words `subscribe cs320` in the body of an email from you to `majordomo@cs.princeton.edu`
3. Read the course web page.
4. Find assignment 1 on the web page, complete it and hand it in by Feb 11 (next Tuesday!). Start early to avoid problems getting ML up and running.

Course Issues

- Grading
- Standard Project - ML!
- Advanced Project
  - Stand-alone projects
  - Propose your own
- Programming policy
- Textbook:
  - Modern Compiler Implementation in ML. Andrew Appel.
  - Elements of ML Programming, second edition. Jeffrey D. Ullman

Who am I?

- I do research on how to make programs and programming languages better
  - faster
  - more reliable
  - more secure
  - easier to maintain
  - easier to debug
  - just generally cooler
- To make this happen, I need to build better compilers—the compiler implements the programming language.
- More specifically, I study logic and type theory and I show how to use these tools to implement better compilers. (I sometimes think about system security and networking just for fun on the side.)
- Education: Cornell University - Computer Science and Programming Languages
- Ph.D. Project: Type Systems for Low-level Languages

What is ML and Why should we use it?

- A Functional Language
- Higher-Order Functions
- Value-Oriented Programming
- Garbage Collection
- Data types and Pattern Matching
- Powerful Type System
- Type Inference
- Expressive Module System
- Elegant and effective tool for building compilers

ML Reference

Useful links:

- A list of ML references and tutorials:
  `http://cm.bell-labs.com/cm/cs/what/smlnj/doc/literature.htm`
- *Introduction to Standard ML* by Robert Harper
- *Using the SML/NJ System* by Peter Lee

Using ML—Mode 1

- On the “Arizona” machines, add `/u/cs320/ml` to path:
  - `ssh arizona.princeton.edu`
  - Edit `.cshrc` to add the line
    `setenv PATH ${PATH}:/u/cs320/ml`
- Interactive mode
  - Invoke: `sml`
  - Type in commands, like in a calculator:

```
Standard ML of New Jersey, Version 110.07...
- 3+4;
val it = 7 : int;
- val x = it * 2;
val x = 14 : int;
```

  - ML evaluates your expression and produces the answer and its type.
  - To quit: `^D`

Using ML—Mode 2

- Compilation Manager
  - Write a program in `file1.sml`:
  - Write another program (or an extension) in `file2.sml`:
  - Create `sources.cm`:

```
Group is
  file1.sml
  file2.sml
```

  - Run ML again by typing `sml` in your shell
  - Type:

```
- CM.make();
```

  - ML reads in all the files mentioned in your `sources.cm` file and compiles them for you.

ML Basics

- Basics
- Arithmetic Operators
- Strings
- Lists
- Functions
- Recursion
- Pattern Matching

Basics

```
% sml
Standard ML of New Jersey, Version 110.0.7,
September 28, 2000 [CM; autoload enabled]
- (* 2. Arithmetic Operations: +,-,*,/,div, mod, ~*)
- 1;
val it = 1 : int
- 1+2*3;
val it = 7 : int
- true;
val it = true : bool
- "hello";
val it = "hello" : string
- "cs320";
```

More Operators

```
- (* Arithmetic *)
- 4.0/2.0;
val it = 2.0 : real
- 4 div 2;
val it = 2 : int
- 5 mod 3;
GC #0.0.0.0.1.7:   (0 ms)
val it = 2 : int
- val x=7;
val x = 7 : int
- val y=8;
val y = 8 : int
- val z=x*y+3;
GC #0.0.0.0.1.9:   (0 ms)
val z = 59 : int
- ~1.0 + 2.0;
val it = 1.0 : real
```


Simple Functions

```
- fun add4(x) = x+4;
val add4 = fn : int -> int
-
- val y = 5*add4(4);
val y = 40 : int
-
- (* type overload *)
- fun square(x) = x*x;
val square = fn : int -> int
-
- fun square(x:real) = x*x;
val square = fn : real -> real
-
- val y = square(2.0);
val y = 4.0 : real
```


Type Errors

```
- 2.0-1;
stdIn:22.1-22.6
Error: operator and operand don't agree [literal]
  operator domain: real * real
  operand:         real * int
  in expression:
    2.0 - 1
```


More Operators

```
- (* String Operators: ^ *)
-
- val y = "hello" ^ "there!";
val y = "hellothere!" : string
-
- (* Comparison Operators: =,<,>,<=,>=,<> *)
-
- val z = 2<=10;
val z = true : bool
-
- (* Logical Operators: not, andalso, orelse *)
-
- 1<5 andalso 2>10;
val it = false : bool
```

Tuples

```
-
- (* List of 2 or more elements of any type,
=    separated by ',', surrounded by ()
=   *)
-
- val tup1 = (3, 4.0, "five");
val tup1 = (3,4.0,"five") : int * real * string
-
- val tup2 = (5, 6.0, "four");
val tup2 = (5,6.0,"four") : int * real * string
-
- val x = #2(tup2);
val x = 6.0 : real
```


Lists 1

```
- (* Zero or more elements of a single type.
=    Separated by ',' and surrounded by []
=  *)
- (* [1,2,3], [2.0], ["hi", "hello"], [1,2.0]=>Illegal! *)
- (* for any type 'T', there is a type 'T list' *)
-
- (* Operations on Lists *)
-
- hd([1,2,3]);
val it = 1 : int
-
- tl([1,2,3]);
val it = [2,3] : int list
```


Simple Functions

```
- fun max3(a:int, b, c) =
=   if a>b then
=     if a>c then a
=     else c
=   else if b>c then b
=        else c;
val max3 = fn : int * int * int -> int
-
- max3(10, 20, 30);
val it = 30 : int
```


Tuples

```
-
- (* ML regards each function as taking a single parameter
=    Multiple params = single tuple
=    All functions return single value,
=    although "multiple values" may be encapsulated into a tu
=
=     (* by the way, comments can be nested *)
=
=  *)
-
- fun sumAndDiff(x:int, y) = (x+y, x-y);
val sumAndDiff = fn : int * int -> int * int
-
- sumAndDiff(2,3);
val it = (5,~1) : int * int
-
```

Recursion

```
- fun fact(n) =
=   if n=0 then 1
=   else n*fact(n-1);
val fact = fn : int -> int
-
- fact(0);
val it = 1 : int
-
- fact(4);
val it = 24 : int
-
- fun listcount(ls) =
=   if ls = nil then 0
=   else 1+listcount(tl(ls));
val listcount = fn : ''a list -> int
```


Pattern Matching 1

```
- (* strength of ML is ability to define functions based
=    on pattern of parameters *)
- (* x::xs => matches non-empty list argument, x:hd, xs:tail
-
- fun listCount(nil) = 0
= |   listCount(x::xs) = 1+listCount(xs);
val listCount = fn : 'a list -> int
-
- listCount([1,2,3,4]);
val it = 4 : int
-
```


Lists 2

```
- [];
val it = nil : 'a list
-
- 2::[2,3,4];
val it = [2,2,3,4] : int list
-
- ["a", "b"]@["c", "d"];
val it = ["a","b","c","d"] : string list
-
- [1]::[2]::nil;
val it = [[1],[2]] : int list list
```

Advice: When trying to figure out what sort of list you have, always decompose the list into its primitive representation. eg: `1::2::3::nil`

Mutual Recursion

```
- fun isEven(n) =
=   if n=0 then true
=   else isOdd(n-1)
= and
= isOdd(n) =
=   if n=0 then false
=   else isEven(n-1);
val isEven = fn : int -> bool
val isOdd = fn : int -> bool
-
- isEven(0);
val it = true : bool
-
- isOdd(0);
val it = false : bool
```

Pattern Matching 3

```
- (* can also pattern-match function result *)
-
- fun sumAndDiff(x:int, y) = (x+y, x-y);
val sumAndDiff = fn : int * int -> int * int
-
- val (sum, diff) = sumAndDiff(10, 5);
val sum = 15 : int
val diff = 5 : int
-
```


Let

```
- fun xten(x:real) = x*x*x*x*x*x*x*x*x*x;
val xten = fn : real -> real

- xten(2.0);
val it = 1024.0 : real


- fun xten(x:real) = let
=   val x5 = x*x*x*x*x
= in
=   x5 * x5
= end;
val xten = fn : real -> real

- xten(2.0);
val it = 1024.0 : real
```


Pattern Matching 2

```
- (* M, N: two integer lists, sorted in increasing order
=   merge(M, N) -> new list with elements of M, N also sorted
-
- fun merge(M, []) = M
=   | merge([], N) = N
=   | merge(x::xs, y::ys) =
=       if (x:int)<y then x::merge(xs, y::ys)
=       else y::merge(x::xs, ys);
val merge = fn : int list * int list -> int list
-
- merge([1,3,5,8], [2,4,9]);
val it = [1,2,3,4,5,8,9] : int list
```


More ML

- Let
- Defining New Types
- Polymorphism
- Higher-Order Functions
- Signatures
- References
- Case Statements
- Exceptions
- Records

Defining New Types

```
- datatype money = Nickel | Dime |Quarter | Bill of
int;
datatype money = Bill of int | Dime | Nickel |
Quarter

- fun value(Nickel) = 5
=    | value(Dime) = 10
=    | value(Quarter) = 25
=    | value(Bill(x)) = x*100;

val value = fn :  money -> int

- value(Bill(10));
 val it = 1000 :  int
```


Polymorphism

A function is *polymorphic* if the types of its parameters are partially or completely flexible.

- Completely Flexible:

```
- fun identity(x) = x;
val identity = fn : 'a -> 'a
- identity(2);
val it = 2 : int
- identity([1,2]);
val it = [1,2] : int list
```

- Partially Flexible:

```
- fun reverse(nil) = nil
= | reverse (x::xs) = reverse(xs) @ [x];
val reverse = fn : 'a list -> 'a list
```


Polymorphic Type Definitions

- Consider:

```
- datatype intTree = Leaf of int
= | Node of intTree * intTree;
datatype intTree =
    Leaf of int | Node of intTree * intTree
```

- Using 'a to represent any type:

```
- datatype 'a btree = Leaf of 'a
= | Node of 'a btree * 'a btree;
datatype 'a btree =
    Leaf of 'a | Node of 'a btree * 'a btree

- val x = Node(Leaf(2), Leaf(3));
val x = Node (Leaf 2,Leaf 3) : int btree
- val y = Node(Leaf("a"), Node(Leaf("b"), Leaf("c")));
val y = Node (Leaf "a",Node (Leaf #,Leaf #)) :
    string btree
```


Higher-Order Functions

- Functions can return functions:

```
- fun add(x:int) = let
=   fun g(y) = x + y
= in
=   g
= end;

val add = fn : int -> int -> int
- val z = add(4);
val z = fn : int -> int
- z(5);
val it = 9 : int
```

Higher-Order Functions

- Functions can be parameters:

```
- fun map(F, nil) = nil
= |   map(F, x::xs) = F(x)::map(F,xs);
val map = fn : ('a -> 'b) * 'a list -> 'b list
- fun square(x:int) = x * x;
val square = fn : int -> int
- map(square, [1,2,3,4]);
val it = [1,4,9,16] : int list
```

- ML allows “anonymous” functions:

```
- map(fn(x) => x * x * x, [1,2,3,4]);
val it = [1,8,27,64] : int list
```


Function Composition

- C(x) = F(G(x))

```
- fun comp(F, G) = let
=   fun C(x) = F(G(x))
= in
=   C
= end;
val comp = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

- fun F(x) = x + 3;
val F = fn : int -> int
- fun G(x) = x * 10;
val G = fn : int -> int
- val H = comp(F, G);
val H = fn : int -> int
- H(10);
val it = 103 : int
```


Signatures

- Signature is a “type” for a structure.
- Describes types and other specifications for some elements of a structure.
- Restricts type of certain structure elements.
- Implements “information hiding” by making certain definitions inaccessible to user.

```
structure someStruct = struct
=   val a = 5;
=   fun listCount(nil) = 0
=   |   listCount(x::xs) = 1 + listCount(xs);
=   datatype color = Red | Green | Blue;
= end;

structure someStruct : sig
    datatype color = Blue | Green | Red
    val a : int
    val listCount : 'a list -> int
  end
```


Signatures

Suppose we want a structure where `listCount` is restricted to `int lists`:

```
- signature S = sig
=   val a : int;
=   val listCount : int list -> int;
=   datatype color = Red | Green | Blue;
= end;

signature S = sig
    ...   val listCount : int list -> int   ...
  end

- structure T : S = someStruct;
structure T : S
- T.listCount([1,2,3]);
val it = 3 : int
- T.listCount(["a","b","c"]);
Error: operator and operand don't agree
```

Signatures

Suppose we don't want the user to access `listCount`:

```
- signature V = sig
=  val a : int;
=   datatype color = Blue | Green | Red;
= end;

signature V = sig
    val a : int
    datatype color = Blue | Green | Red
  end

- structure Z : V = someStruct;
structure Z : V
- Z.a;
val it = 5 : int
- Z.listCount([1,2,3]);
Error: unbound variable or constructor
```


References

```
- val x = 6;
val x = 6 : int
- val x = 7;
val x = 7 : int

- val x = ref 6;
val x = ref 6 : int ref
- x;
val it = ref 6 : int ref
- !x;
val it = 6 : int
- x := 7;
val it = () : unit
- !x;
val it = 7 : int
```

X | 6

X | 7

X | ref: 6

X | ref: ~~6~~ 7

Case Statement

```
- datatype money = Nickel | Dime | Quarter | Bill of int;
datatype money = Bill of int | Dime | Nickel | Quarter

- fun value(Nickel) = 5
= |   value(Dime) = 10
= |   value(Quarter) = 25
= |   value(Bill(x)) = x * 100;

val value = fn : money -> int

- fun value(x) =
=   case x of
=       Nickel => 5
=     | Dime => 10
=     | Quarter => 25
=     | Bill(x) => x * 100;

val value = fn : money -> int
```


Exceptions

- *Exceptions* allow errors to be caught easily regardless of function return type.
- For example: n! not defined for $n < 0$:

```
- exception negativeInt;
exception negativeInt

- fun fact(n) = if n = 0 then 1
=               else if n > 0 then n * fact(n-1)
=                    else raise negativeInt;

val fact = fn :  int -> int

- fact(~1);
uncaught exception negativeInt raised
```

Records

- Can use pattern-matching with records.

```
- fun getAge(person,
=            nil:{name:string,age:int,ID:int} list) =
=        raise notFound
= |    getAge(person, {name=n, age=a, ID=i}::xs) =
=        if person = n then a
=        else getAge(person, xs);
```

- Rather than creating new variable to represent value of a field, may use field name itself. Helps increase clarity.

```
- fun getAge(person,
=            nil:{name:string,age:int,ID:int} list) =
=        raise notFound
= |    getAge(person, {name, age, ID}::xs) =
=        if person = name then age
=        else getAge(person, xs);
```


Records

- *Records* are a more general form of a tuple.

```
- val x = {name="John", age=21, ID=1111};
val x = {ID=1111,age=21,name="John"} :
    {ID:int, age:int, name:string}
- #name(x);
val it = "John" : string
- #age(x);
val it = 21 : int
```

- Tuples are equivalent to numbered records:

```
- val y = ("a","b","c");
val y = ("a","b","c") : string * string * string
- val y = {1="a",2="b",3="c"};
val y = ("a","b","c") : string * string * string
- #3(y);
val it = "c" : string
```