CS421 Compilers and Interpreters

Zhong Shao
Dept. of Computer Science
Yale University
Spring 2015

Course Structure

• Course home page: http://flint.cs.yale.edu/cs421
  all lecture notes and other course-related information are available on this
class home page.
• 13-week lectures (based on Appel book + Ullman book + other)
  compiler basics, internals, algorithms, and advanced topics, etc.
• 7 programming assignments
  build a compiler compiling Tiger progs into the X86 assembly code.
• Occasional problem sets plus a final exam
• Use the SML/NJ environment on the Zoo Linux PCs

Why Study Compilers?

or why take CS421?

• To enhance understanding of programming languages
• To have an in-depths knowledge of low-level machine executables
• To write compilers and interpreters for various programming languages and
domain-specific languages
  Examples: Java, JavaScript, C, C++, C#, Modula-3, Scheme, ML, Tcl/Tk, Database Query Lang.,
  Mathematica, Matlab, Shell-Command-Languages, Awk, Perl, your .mailrc file, HTML, TeX,
  PostScript, Kermit scripts, ...
• To learn various system-building tools : Lex, Yacc, ...
• To learn interesting compiler theory and algorithms.
• To learn the beauty of programming in modern programming lang.

Systems Environments

• To become a real computer professional, you must not only know how to write
good programs, but also know how programs are compiled and executed on
different machines.
• Core Systems Environments include: programming languages, compilers,
  computer architectures, and operating systems
  1. a language for you to express what to do
  2. a translator that translates what you say to what machine knows
  3. an execution engine to execute the actions
  4. a friendly operating environment that connects all the devices
• Application Systems Environments include: distributed systems, computer
  networks, parallel computations, database systems, computer graphics,
  multimedia systems.
### Compilers and Interpreters

Given a program $P$ written in language $L$,

- A **compiler** is simply a translator; compiling a program $P$ returns the corresponding machine code (e.g., Power PC) for $P$.
- An **interpreter** is a translator plus a virtual machine engine; interpreting a program $P$ means translating $P$ into the virtual machine code $M$ and then executing $M$ upon the virtual machine and return the result.

In summary, we will focus on the following:

- how to write a translator?
- what are the possible source languages and target languages?
- what are the possible physical or virtual machine architectures?
- (a little bit on) why does the translation preserve the semantic meaning?

### Table 1: various forms of translators

<table>
<thead>
<tr>
<th>$L$</th>
<th>$L'$</th>
<th>translator</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++, ML, Java</td>
<td>assembly/machine code</td>
<td>compiler</td>
</tr>
<tr>
<td>assembly lang.</td>
<td>machine code</td>
<td>assembler</td>
</tr>
<tr>
<td>“object” code (*.o file)</td>
<td>“executable” code (a.out)</td>
<td>linker/loader</td>
</tr>
<tr>
<td>macros/text</td>
<td>text</td>
<td>macro processor (cpp)</td>
</tr>
<tr>
<td>troff/Tex/HTML</td>
<td>PostScript</td>
<td>document formatter</td>
</tr>
<tr>
<td>any file (e.g., foo)</td>
<td>compressed file (foo.Z)</td>
<td>file compressor</td>
</tr>
</tbody>
</table>

### Compilation Phases

- **Source code**
  - Lexical analysis (lexer)
  - Syntax analysis (parser)
  - Abstract syntax
  - Semantic & type analysis
  - (Valid) abstract syntax
- **Intermediate code**
  - Intermediate code generator
  - Code optimization
  - (Better) intermediate code
  - Machine code generator
  - Machine code
  - Instr. sched. and reg. alloc.
  - (Faster) machine code

### Programming Assignments

- **Tiger source code**
  - Lexer (as2, using ml-lex)
  - Syntax analysis (parser)
  - (Valid) abstract syntax
  - Intermediate code generator
  - Code optimization
  - (Better) intermediate code
  - Machine code
  - Instr. sched. and reg. alloc.
  - (Faster) machine code
An Example of Tiger

(* A program to solve the 8-queens problem, see Appel's book *)

let

val N := 8

type intArray = array int

var row := intArray [ N ] of 0

var col := intArray [ N ] of 0

var diag1 := intArray [N+N-1] of 0

var diag2 := intArray [N+N-1] of 0

function printboard() =

(for i := 0 to N-1

do (for j := 0 to N-1


doi (if col[i]=j then " O" else ".");

print("\n"));

print("\n")

function try(c:int) =

(* for i:= 0 to c do print("."); print("\n"); flush(); *)

if c=N then printboard()

else for r := 0 to N-1


do if row[r]=0 & diag1[r+c]=0 & diag2[r+7-c]=0

then (row[r]:=1; diag1[r+c]:=1; diag2[r+7-c]:=1;


col[c]:=r; try(c+1);

row[r]:=0; diag1[r+c]:=0; diag2[r+7-c]:=0)

in try(0)

end

Using the SML/NJ compiler

• Add /c/cs421/bin to the front of your PATH variable

• Type sml to run the SML/NJ compiler (used in assignment 1)

• Type CM.make "sources.cm"; to run the separate compilation system (the makefile is called sources.cm, used in as2 -- as7)

• Ctrl-c exits the compiler; Ctrl-c breaks the execution; Ctrl-z stops the execution as normal Unix programs

• Three ways to run ML programs: (1) type in your code in the interactive prompt inside sml; (2) edit your ML code in a file, say, foo.sml; then inside sml type use "foo.sml"; (3) use the separate compilation system;

• The directory /c/cs421/as contains all the files needed for doing all 7 programming assignments in Appel's book.

Why Standard ML ?

• Efficiency

• Safety and simplicity

• Statically-typed

• Powerful module system

• Garbage collection (automatic memory management)

• Low-level systems programming support

• Higher-order functions

• Polymorphism

• Other features: formal definitions, type inference, value-oriented prog.

ML Tutorial

• Integers: 3, 54; Negative Integers: ~3, ~54

• Reals: 3.0, 3.14, ~3.32E~7;

• Overloaded arithmetic operators: +, -, *, /, <, >, <=, >

• Boolean: true, false; operators: andalso, orelse, not

• Strings: "hello world\n", "yale university", ...

• Lists: [], 3::4::nil, [2,3], ["freshman", "senior"], ...

• Expressions: constant, list expr, cond. expr, let expr, function application

• Declarations:

  value binding:

  val x = 3;
  val y = x + x;

  function-value binding:

  fun fac n = if n=0 then 1
               else n*(fac(n-1));
ML Tutorial (cont’d)

- **Function values**
  The expression “fn var => exp” denotes the function with formal parameter var and body exp. The fn is pronounced “lambda”.
  
  examples: 
  
  val f = fn x => (fn y => (x+y+3))
  
  it is equivalent to 
  
  fun f x y = x+y+3

- **Constructed values**
  
  pair and tuple: 
  
  (3, 4.5), ("email", 4.5+x, true)
  
  records: 
  
  {lab1 = exp1, ..., labn = expn} (n>=0)
  
  examples: 
  
  {make = "Ford", built = 1904}
  
  unit: denoted as (), used to represent 0-tuple or empty record {}

ML Tutorial (cont’d)

- **Extract the n-th field of a n-tuple**
  
  val x = (3,4.5,"hello")
  
  val y = #1(x)
  
  val z = #3(x)

- **Extract a specific field of a record**
  
  val car = {make = "Ford", year = 1984}
  
  val m = #make(car)
  
  val y = #year(car)

ML Tutorial (cont’d)

- **Patterns** — a form to decompose constructed values, commonly used in value binding and function-value binding.
  
  val pat = exp 
  
  fun var(pat) = exp
  
  variable pattern: 
  
  val x = 3
  
  fun f(y) = x+y+2
  
  pattern for pairs, tuples, and records:
  
  val pair = (3,4)
  
  val (x,y) = pair
  
  val car = {make = "Ford", built = 1904}
  
  fun modernize{make = m, built = year} =
  
  {make = m, built = year+1}

  wildcard pattern: _
  
  unit pattern: ()
  
  constant pattern: 3, 4.5
  
  constructor pattern: []
**ML Tutorial (cont’d)**

- **Pattern Matching Examples:**
  ```ml
  fun length l = case l
     of [] => 0
         | [a] => 1
         | _::r => 1 + (length r)
  
  fun length [] = 0
      | length [a] = 1
      | length (_::r) = 1 + (length r)
  
  fun even 0 = true
      | even n = odd(n-1)
  
  and odd 0 = false
      | odd n = even(n-1)
  ```

- **Type Expressions**
  - int, bool, real, string, int list, t1*t2, t1->t2
  - x : int
  - fac : int -> int
  - f : int -> int -> int
  - modernize : {make : string, build : int} ->
    {make : string, build : int}
  - length : 'a list -> int
    (3,4.0) : int * real

- **Type Abbreviations**
  ```ml
  type tycon = ty
  
  Examples:
  type car = {make : string, built : int}
  type point = real * real
  type line = point * point
  ```

- **Datatype declarations:**
  ```ml
  datatype tycon = con1 of ty1
     | con2 of ty2
     .......
     | conn of tyn
  
  This declares a new type, called “tycon” with n value constructors
  con1 ,..., conn. The “of tyi” can be omitted if coni is nullary.
  
  Examples: datatype color = RED | GREEN | BLUE
  
  this introduces a new type color and 3 new value constructors
  RED, GREEN, and BLUE, all have type color. A value constructor
  can be used both as a value and as a pattern, e.g.,
  ```

  ```ml
  fun swap(RED) = GREEN
      | swap(GREEN) = BLUE
      | swap(BLUE) = RED
  ```

**ML Tutorial (cont’d)**

- **Datatype declaration example :**
  ```ml
  datatype 'a list = nil
     | :: of 'a * 'a list
  
  fun map f [] = []
      | map f (a::r) = (f a)::(map f r)
  
  fun rev l = let fun h([], r) = r
      | h(a::z, r) = if p a then h(z, a::r)
        in h(l, [])
      end
  
  fun filter(p, l) =
      let fun h([], r) = rev r
          | h(a::z, r) = if p a then h(z, a::r)
            else h(z, r)
        in h(l, [])
      end
  ```
**ML Tutorial (cont’d)**

- **Datatype declaration example:**
  
  \[
  \text{datatype btree = LEAF} \\
  | \text{NODE of int * btree * btree}
  \]

  \[
  \text{fun depth LEAF = 0} \\
  | \text{depth (NODE(_,t1,t2)) = max(depth t1,depth t2)+1}
  \]

  \[
  \text{fun insert(LEAF, k) = NODE(k,LEAF,LEAF)} \\
  | \text{insert(NODE(i,t1,t2),k) = if k > i then NODE(i,t1,insert(t2,k))} \\
  | \text{else if k < i then NODE(i,insert(t1,k),t2) else NODE(i,t1,t2)}
  \]

  \[
  \text{fun preord(LEAF) = ()} \\
  | \text{preord(NODE(i,t1,t2)) = (print i; preord t1; preord t2)}
  \]

- **Use datatype to define a small language (Prog. Assignment 1):**
  
  \[
  \text{type id = string} \\
  \text{datatype binop = PLUS | MINUS | TIMES | DIV} \\
  \text{datatype stm = SEQ of stm * stm} \\
  | \text{ASSIGN of id * exp} \\
  | \text{PRINT of exp list}
  \]

  \[
  \text{and exp = VAR of id} \\
  | \text{CONST of int} \\
  | \text{BINOP of exp * binop * exp} \\
  | \text{ESEQ of stm * exp}
  \]

  (* sample program: a = 5 + 3; print a *)

  \[
  \text{val prog =} \\
  \text{SEQ(ASSIGN("a",BINOP(CONST 5,PLUS,CONST 3)),} \\
  \text{PRINT[VAR "a"])
  \]

- **Find out the size of program written in the above small language ...**

  \[
  \text{fun sizeS (SEQ(s1,s2)) = sizeS(s1) + sizeS(s2)} \\
  | \text{sizeS (ASSIGN(i,e)) = 2 + sizeE(e)} \\
  | \text{sizeS (PRINT l) = 1 + sizeEL(l)}
  \]

  \[
  \text{and sizeE (BINOP(e1,_,e2) = sizeE(e1)+sizeE(e2)+2} \\
  | \text{sizeE (ESEQ(a,e)) = sizeS(a)+sizeE(e)} \\
  | \text{sizeE _ = 1}
  \]

  \[
  \text{and sizeEL [] = 0} \\
  | \text{sizeEL (a::r) = (sizeE a)+(sizeEL r)}
  \]

  Then \[\text{sizeS(prog)}\] will return 8.

- **Homework:** read Ullman Chapter 1-3, read Appel Chapter 1, and do Programming Assignment #1 (due January 28, 2015)