CS226/326 Compilers for Computer Languages

David MacQueen
Department of Computer Science
Spring 2003

Why Study Compilers?

- *To learn to write compilers and interpreters for various programming languages and domain specific languages*

  E.g. Java, Javascript, C, C++, C#, Modula-3, Scheme, ML, Tcl, SQL, MatLab, Mathematica,Shell, Perl, Python,HTML, XML, TeX,PostScript

- *To enhance understanding of programming languages*
- *To understand how programs work at the machine level*
- *To learn useful system-building tools like Lex and Yacc*
- *To learn interesting compiler theory and algorithms*
- *To experience building a significant system in a modern programming language (SML)*

Compilers are Translators

$L_1 \longrightarrow$ *Translator* $\longrightarrow L_2$

L_1	*Translator*	L_2
C, ML, Java, ...	compiler	assembly/machine code
assembly language	assembler	machine code
object code (.o files)	link loader	executable code
macros+text	macro processor (cpp)	text
troff/TeX	document formatter	PostScript/PDF

Compilers and Interpreters

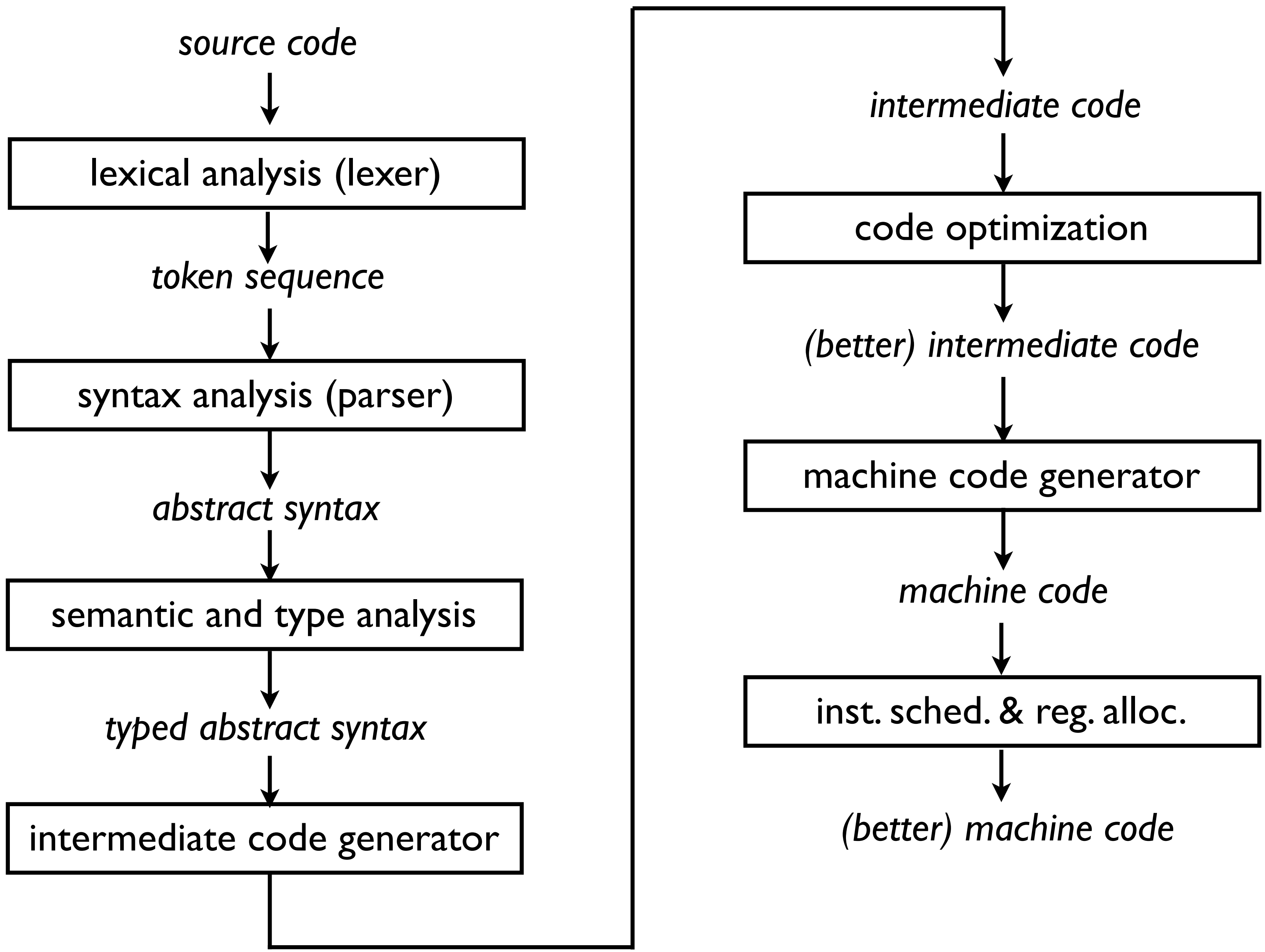
Given a program P (source code) written in language L

- A **compiler** is simply a translator; compiling P produces the corresponding *machine code* (PowerPC, Sparc), also known as the *object code*.
- An **interpreter** is a virtual machine (*i.e.* a program) for directly executing P (or some machine representtion of P).
- A **virtual machine-based compiler** is a hybrid involving translation P into a virtual machine code M and an virtual machine interpreter that executes M (e.g. the Java Virtual Machine). Virtual machine code is sometimes called *byte code*.

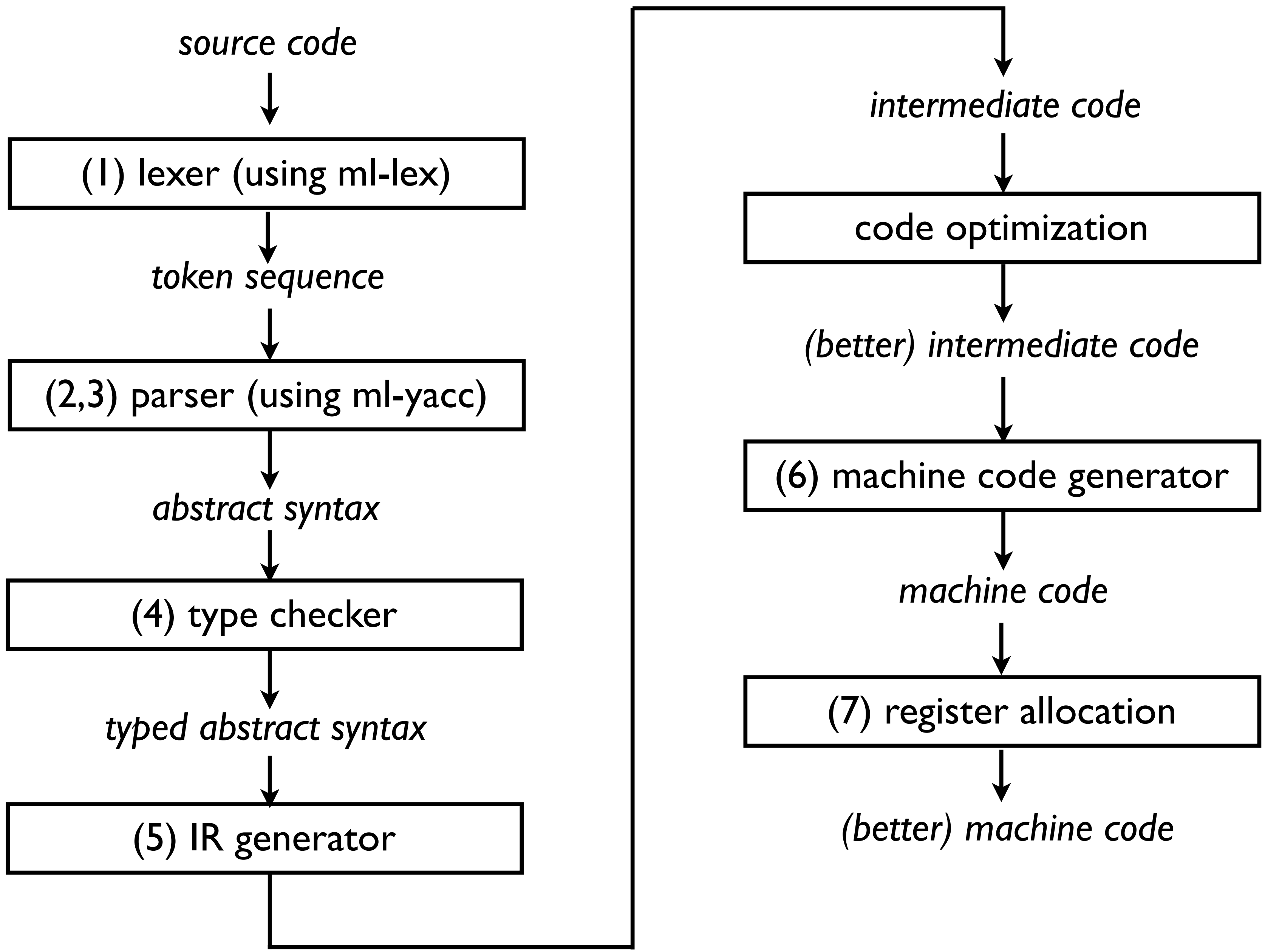
We will focus on the following:

- How to characterize the source language L and the target language.
- How to translate from one to the other.

Compilation Phases



Programming Assignments



A Tiger Program

```
/* A program to solve the 8-queens problem */

let
    var N := 8

    type intArray = array of 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
            do print(if col[i]=j then " O" else " .");
            print("\n"));
        print("\n"))

    function try(c:int) =
      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
```

Why Standard ML?

A language particularly suited to compiler implementation.

- Efficiency
- Safety
- Simplicity
- Higher-order functions
- Static type checking with type inference
- Polymorphism
- Algebraic types and pattern matching
- Modularity
- Garbage collection
- Exception handling
- Libraries and tools

Using the SML/NJ Compiler

- *Type "`sml`" to run the SML/NJ compiler*

  Normally installed in /usr/local/bin, which should be in your PATH.

- `Cntl-d` *exits the compiler,* `Cntl-c` *interrupts execution.*

- *Three ways to run ML programs:*

  1. type in code in the interactive read-eval-print loop
  2. edit ML code in a file, say `foo.sml`, then type command

     `use "foo.sml";`

  3. use Compilation Manager (CM):

     `CM.make "sources.cm";`

- *Template code in dir* `/stage/classes/current/22600-1/code`

ML Tutorial 1

Expressions

- *Integers:* `3, 54, ~3, ~54`
- *Reals:* `3.0, 3.14159, ~3.2E2`
- *Overloaded arithmetic operators:* `+, -, *, /, <, <=`
- *Booleans:* `true, false, not, orelse, andalso`
- *Strings:* `"abc", “hello world\n", x^".sml"`
- *Lists:* `[], [1,2,3], ["x","str"], 1::2::nil`
- *Tuples:* `(), (1,true), (3,"abc",true)`
- *Records:* `{a=1,b=true}, {name="fred",age=21}`
- *conditionals, function applications, let expressions, functions*

ML Tutorial 2

Declarations: *binding a name to a value*

value bindings

```
val x = 3
val y = x + 1
```

function bindings

```
fun fact n =
    if n = 0 then 1
    else n * fact(n-1)
```

Let expressions: *local definitions*

```
let decl in expr end
```

```
let val x = 3
    fun f y = (y, x*y)
 in f(4+x)
end
```

ML Tutorial 3

Function expressions

The expression "**`fn`** *`var`* **`=>`** *`exp`*" denotes a function with formal parameter *`var`* and body *`exp`*.

```
val inc = fn x => x + 1
```

is equivalent to

```
fun inc x = x + 1
```

ML Tutorial 4

Compound values

Tuples: `(exp`$_1$`, ... , exp`$_n$`)`

```
(3, 4.5)
val x = (”foo”, x*1.5, true)

val first = #1(x)
val third = #3(x)
```

Records: `{lab`$_1$` = exp`$_1$`, ... , lab`$_n$` = exp`$_n$`}`

```
val car = {make = “Ford”, year = 1910}

val mk = #make car
val yr = #year car
```

ML Tutorial 5

Patterns

a form to decompose compound values, commonly used in value bindings and function arguments

```
val pat = exp                fun f(pat) = exp
```

variable patterns:

```
val x = 3
⇒ x = 3
fun f(x) = x+2
```

tuple and record patterns:

```
val pair = (3,4.0)
val (x,y) = pair
⇒ x = 3, y = 4.0

val {make=mk, year=yr} = car
⇒ mk = "Ford", yr = 1910
```

ML Tutorial 6

Patterns

wildcard pattern: _ *(underscore)*

constant patterns: `3, "a"`

```
fun iszero(0) = true
  | iszero(_) = false
```

constructor patterns:

```
val list = [1,2,3]
val fst::rest = list
⇒ fst = 1, rest = [2,3]

val [x,_,y] = list
⇒ x = 1, y = 3
```

ML Tutorial 7

Pattern matching

match rule: pat **=>** exp

match: pat_1 **=>** exp_1 | ... | pat_n **=>** exp_n

When a match is applied to a value v, we try rules from left to right, looking for the first rule whose pattern matches v. We then bind the variables in the pattern and evaluate the expression.

case expression: **case** *exp* **of** *match*
function expression: **fn** *match*
clausal functional defn: **fun** f pat_1 **=** exp_1
| f pat_2 **=** exp_2
| ...
| f pat_2 **=** exp_2

ML Tutorial 8

Pattern matching examples *(function definitions)*

```
fun length l =
    case l of
      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)
```

ML Tutorial 9

Types

basic types: `int, real, string, bool`

`3 : int,  true : bool,  "abc" : string`

function types: t_1 `->` t_2

`even: int -> bool`

product types: t_1 `*` t_2 `, unit`

`(3,true): int * bool,  (): unit`

record types: `{`lab_1 `:` t_1`, ... ,` lab_n `:` t_n`}`

`car: {make : string, year : int}`

type operators: t `list` *(for example)*

`[1,2,3] : int list`

ML Tutorial 10

Type abbreviations

```
type tycon = ty
```

examples:

```
type point = real * real
type line = point * point
type car = {make: string, year: int}
```

```
type tyvar tycon = ty
```

examples:

```
type ‘a pair = ‘a * ‘a
type point = real pair
```

ML Tutorial 11

Datatypes

$$\texttt{\textbf{datatype}}\ \textit{tycon} = \textit{con}_1\ \texttt{\textbf{of}}\ \textit{ty}_1 \mid \ldots \mid \textit{con}_n\ \texttt{\textbf{of}}\ \textit{ty}_n$$

This is a *tagged union* of variant types ty_1 through ty_n. The tags are the *data constructors* con_1 through con_n.

The data constructors can be used both in expressions to build values, and in patterns to deconstruct values and discriminate variants.

The "**`of`** *`ty`*" can be omitted, giving a nullary constructor.

Datatypes can be *recursive*.

```
datatype intlist = Nil | Cons of int * intlist
```

ML Tutorial 12

Datatype example

```
datatype btree = LEAF
               | NODE of int * btree * btree

fun depth LEAF = 0
  | depth (NODE(_,t1,t2)) = max(depth t1, depth t2)

fun insert(LEAF,k) = NODE(k,LEAF,LEAF)
  | insert(NODE(i,t1,t2),k) =
      if k > i then NODE(i,t1,insert(t2,k))
      else if k < i then NODE(i,insert(t1,k),t2)
      else NODE(i,t1,t2)

(* in-order traversal of btrees *)
fun inord LEAF = []
  | inord(NODE(i,t1,t2)) =
      inord(t1) @ (i :: inord(t2))
```

ML Tutorial 13

Representing programs as datatypes

```
type id = string

datatype binop = PLUS | MINUS | TIMES | DIV

datatype stm = SEQ of stm * stm
             | ASSIGN of id * exp
             | PRINT of exp list

and exp = VAR of id
        | CONST of int
        | BINOP of binop * exp * exp
        | ESEQ of stm * exp

val prog =
    SEQ(ASSIGN(”a”,BINOP(PLUS,CONST 5,CONST 3)),
        PRINT[VAR “a”])
```

ML Tutorial 14

Computing properties of programs: size

```
fun sizeS (SEQ(s1,s2)) = sizeS s1 + sizeS s2
  | sizeS (ASSIGN(i,e)) = 2 + sizeE e
  | sizeS (PRINT es) = 1 + sizeEL es

and sizeE (BINOP(_,e1,e2)) = sizeE e1 + sizeE e2 + 2
  | sizeE (ESEQ(s,e)) = sizeS s + sizeE e
  | sizeE _ = 1

and sizeEL [] = 0
  | sizeEL (e::es) = sizeE e + sizeEL es


sizeS prog ⇒ 8
```