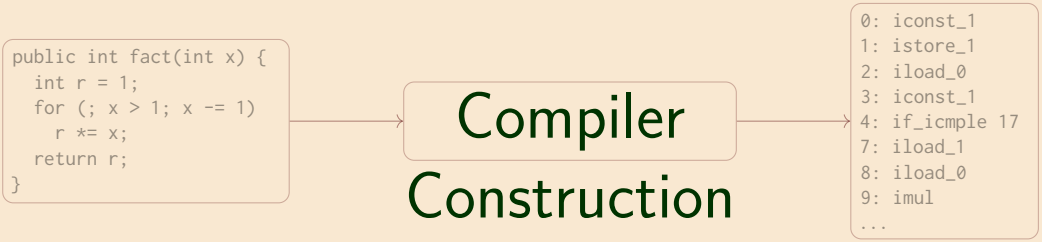


Compiler Construction

Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

Lent 2024

Why study compilers?

Why study compilers?

The Gap

Compiler structure

Course structure

(Largely uncontroversial)

Compilers are complex programs

You often need compilers
(whenever you write and run a program)

If you understand how compilers work,
you'll know what to expect
and how to get the best out of them

Why study compilers?

The Gap

Compiler structure

Course structure

Compiler-like programs are *everywhere*.

All sorts of programs can be viewed as compilers. *For example*,

query languages (GraphQL)
serialisation frameworks (Protobuf)
build systems (make)
game engine scripting (Lua)
financial contracts (MLFI)
music systems (Csound)
text editors (emacs)
hardware description languages (VHDL)
statistical computing environments (R)
browser engines (WebKit)
document processors (LaTeX)
continuous integration (GitHub Actions)
blockchain platforms (Solidity)
legal contracts (Catala)
text processors (sed, grep, ...)
interactive testing systems (expect)
compiler compilers (yacc)
wikis (MediaWiki)

(etc.)

Compilers are *interesting*

Why study compilers?

The Gap

Compiler structure

Course structure

Programming languages are semantically rich,
so programs that process language are rich, too

Compilers view a program from
many different perspectives

Compilers put semantics into practice

Compilers represent $70+$ years of research
A computer science success story!

Compilers involve self-application
(How might we compile a compiler?)

Lots of interesting questions: what does it mean for a compiler to be correct? what kind of optimizations are possible for a particular language? ... for a particular program? ... for particular inputs?


public int fact(int x) {
int r = 1;
for (; x > 1; x -= 1)
r *= x;
return r;
}
The Gap
? ? ? ?
0: iconst_1
1: istore_1
2: iload_0
3: iconst_1
4: if_icmple 17
7: iload_1
8: iload_0
9: imul
...

High-level ⟶ low-level

High-level language

Machine-independent
Complex syntax
Complex type system
Variables
Nested scope
Procedures, functions
Modules, objects
Cannot be run directly

? ? ? ?

Low-level language

Machine-specific
Simple syntax
Simple types
Memory, registers, words
Single flat scope
Can be run directly

Why study compilers?

The Gap

Compiler structure

Course structure

Java

```
class Fact {
 public static int fact(int x) {
  int result = 1;
  for (; x > 1; x -= 1) result *= x;
  return result;
 }
}
```

? ? ?

javac Fact.java
javap -c Fact.class

JVM bytecode

```
...
 0: iconst_1
 1: istore_1
 2: iload_0
 3: iconst_1
 4: if_icmple 17
 7: iload_1
 8: iload_0
 9: imul
10: istore_1
11: iinc 0, -1
14: goto 2
17: iload_1
18: ireturn
...
```

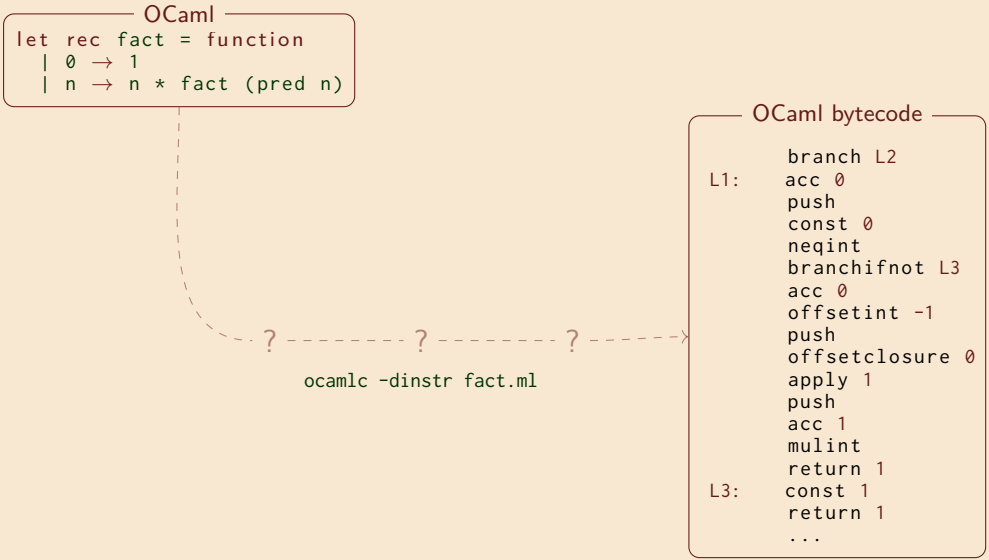
OCaml ⟶ OCaml bytecode

Why study compilers?

The Gap

Compiler structure

Course structure



C ⟶ assembly code

C

```
int fact(int x)
{
  int result = 1;
  for (; x > 1; x -= 1) result *= x;
  return result;
}
```

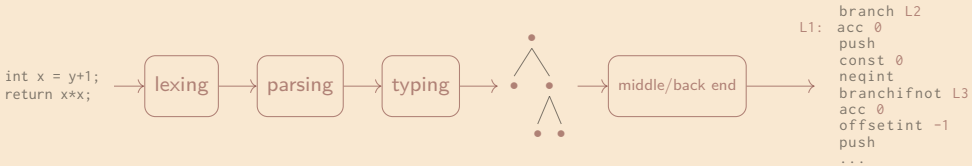
? ? ?

gcc -S fact.c

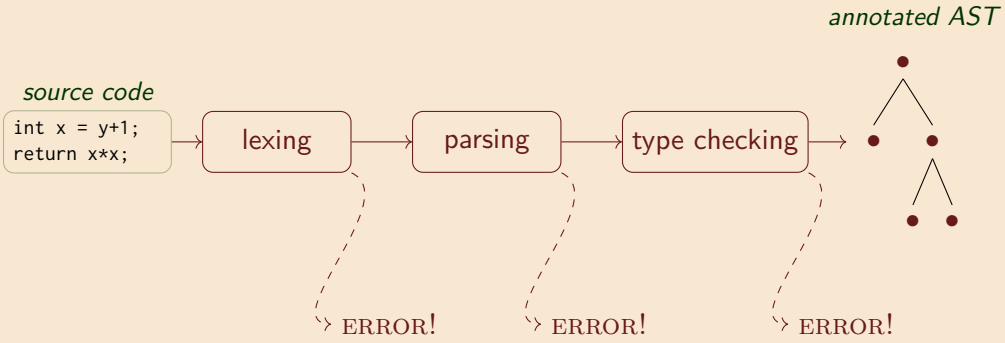
assembly code

```
fact(int):
    cmp     edi, 1
    mov     eax, 1
    jle     .L4
.L3:
    imul    eax, edi
    sub     edi, 1
    cmp     edi, 1
    jne     .L3
    rep ret
.L4:
    rep ret
```

Structure of a compiler

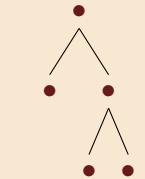


The front end



(*All* error-checking happens in the front end)

The middle & back ends



annotated AST

middle end

generic optimizations

retargetable representation

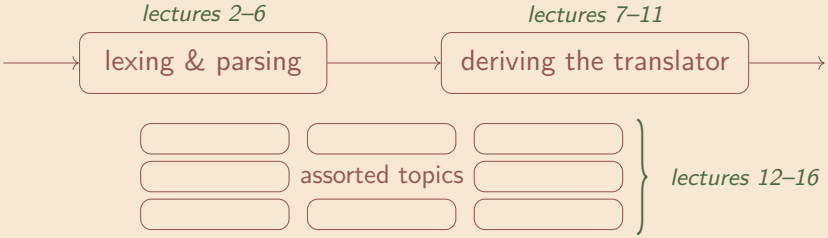
back end

machine-specific optimizations

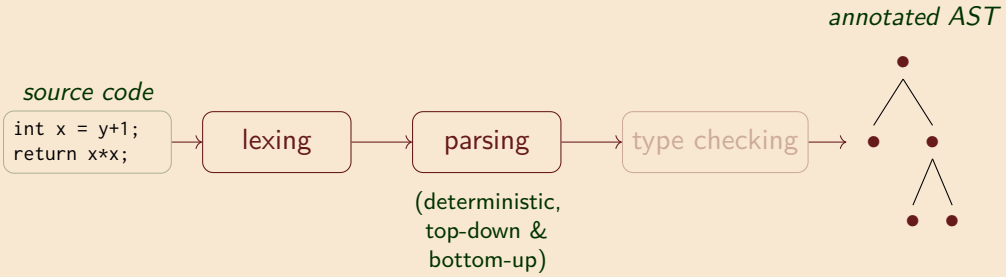
assembly code

```
        branch L2
L1:     acc 0
        push
        const 0
        neqint
        branchifnot L3
        acc 0
        offsetint -1
        push
        offsetclosure 0
        apply 1
        push
        acc 1
        mulint
        return 1
L3:     const 1
        return 1
        ...
```

Structure of this course

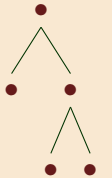


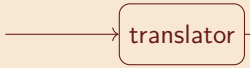
Lectures 2–6: lexing & parsing



We'll derive a compiler for Slang (simple language)

Slang AST





Jargon VM bytecode

```
   MK_CLOSURE(f, 0)
   MK_CLOSURE(L0, 0)
   APPLY
   HALT
L0:
   PUSH STACK_UNIT
   UNARY READ
   LOOKUP STACK_LOCATION -2
   APPLY
   RETURN
```

Plan:

1. start with an interpreter
2. perform principled refinements
3. derive a compiler

Lectures 12–16: assorted topics

- Linking
- OS interface
- Stacks *vs* registers
- Calling conventions
- Generating assembly code
- Objects & inheritance
- Memory management
- Bootstrapping
- Optimisation
- Exceptions

Next time: lexing