MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language

Eijiro Sumii
Tohoku University

Highlights

"<u>Simple</u> and <u>efficient</u> compiler for a minimal functional language"

- Only 2000 lines of OCaml
- Efficiency comparable to OCamlOpt and GCC for several applications
  - Ray tracing, Huffman encoding, etc.
- Used for undergraduates in Tokyo since 2001

Outline of This Talk

- Pedagogical background
- Design and implementation of MinCaml
- Efficiency

Computer Science for Undergraduates in Tokyo

- Liberal arts (1.5 yr)
  - English, German/Chinese/French/Spanish, mathematics, logic, physics, chemistry, ...
  - Computer literacy, CS introduction, Java programming, data structures
- CS major (2.5 yr) [~30 students/yr]
  - Algorithms, OS, architecture, ...
  - SPARC assembly, C, C++, Scheme, OCaml, Prolog

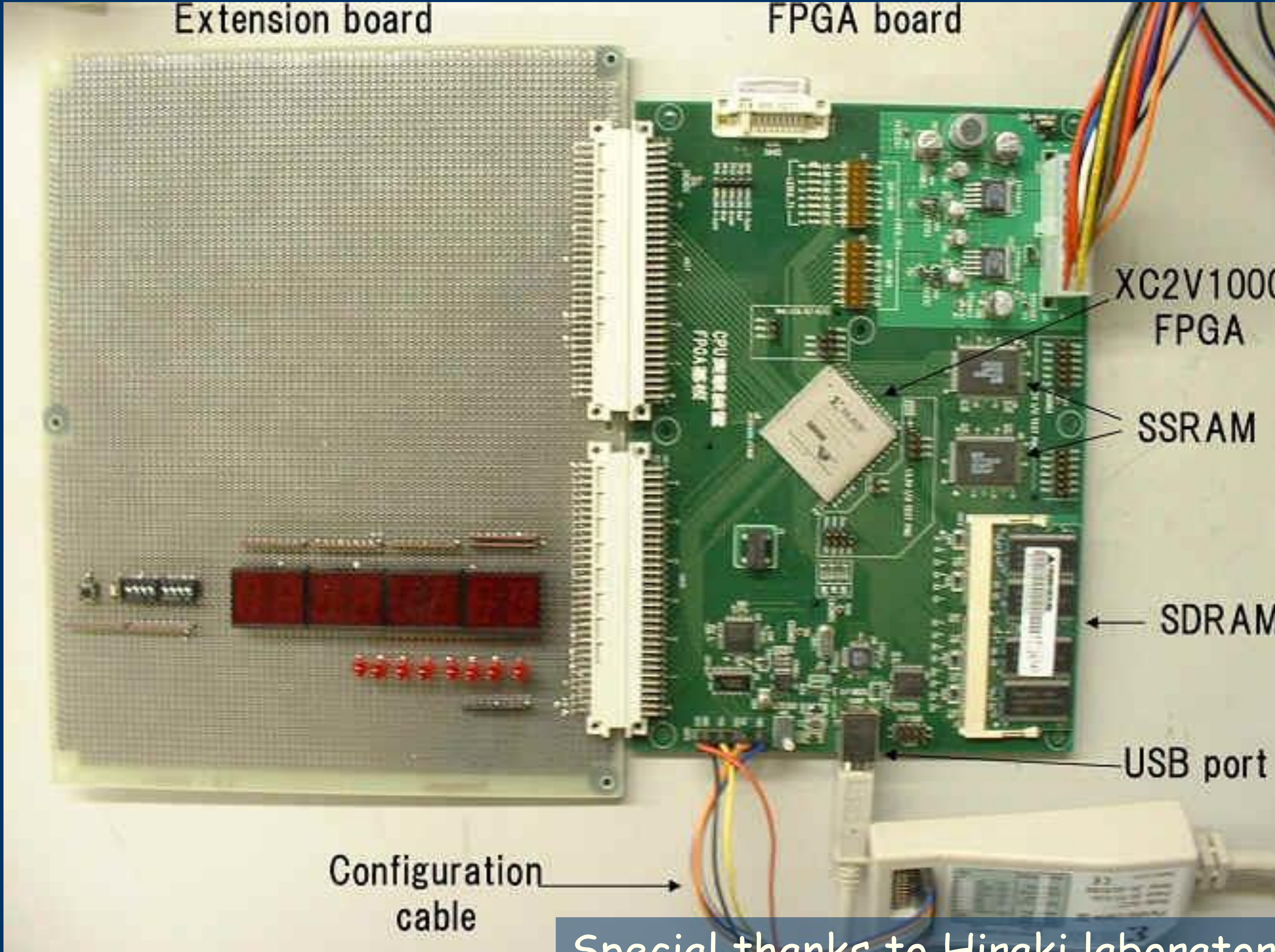
Programming Languages for CS Major in Tokyo

- PL labs (63 hr)
  - Mini-Scheme interpreter in Scheme,
  - Mini-ML interpreter in OCaml,
  - Othello/Reversi competition in OCaml, etc.
- Compiler lecture (21 hr)
  - Parsing, intermediate representations, register allocation, garbage collection, ...
- PL theory lectures (42 hr)
  - λ-calculus, semantics, type theory, ...

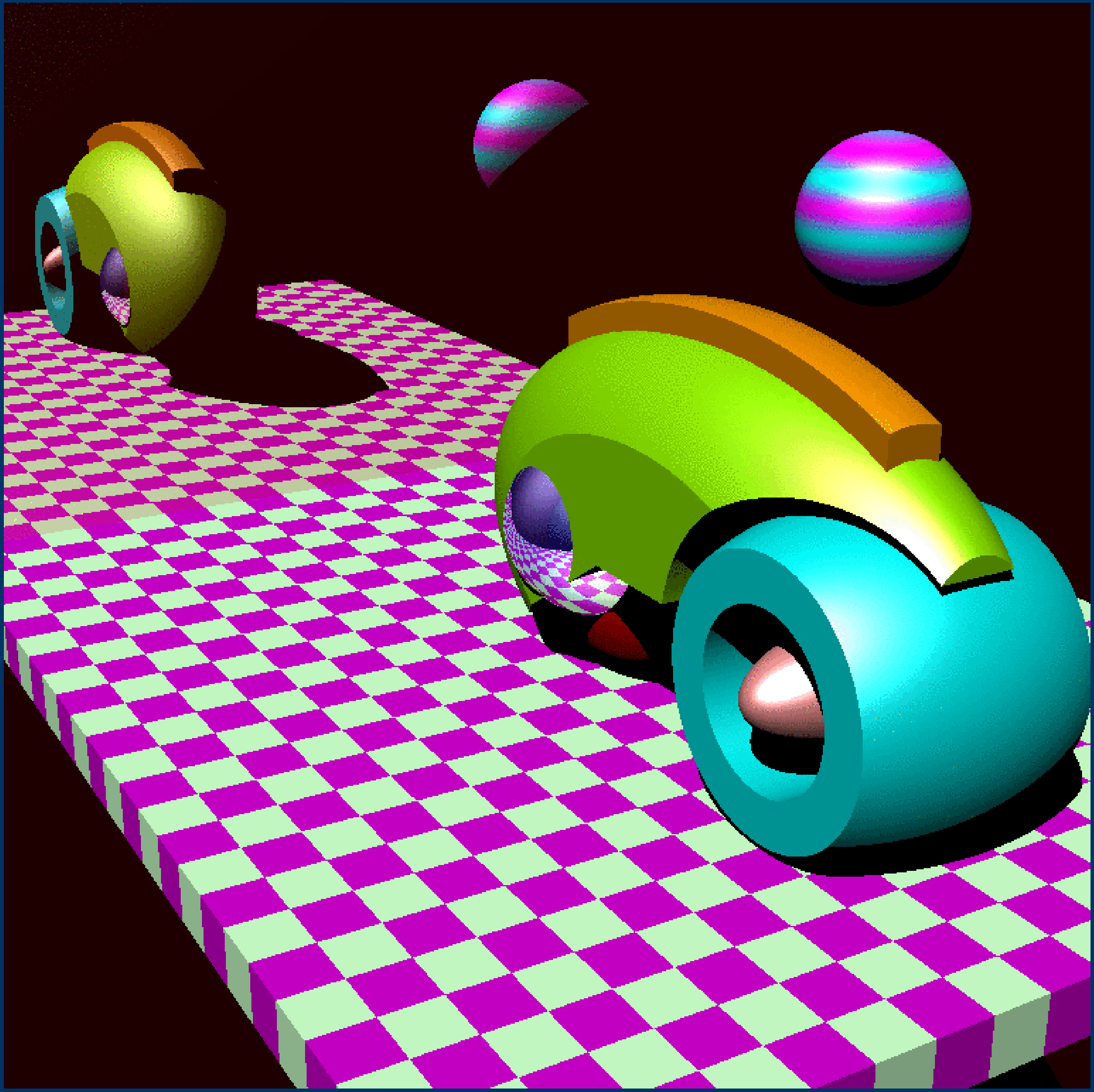
CPU/Compiler Labs (126 hr)

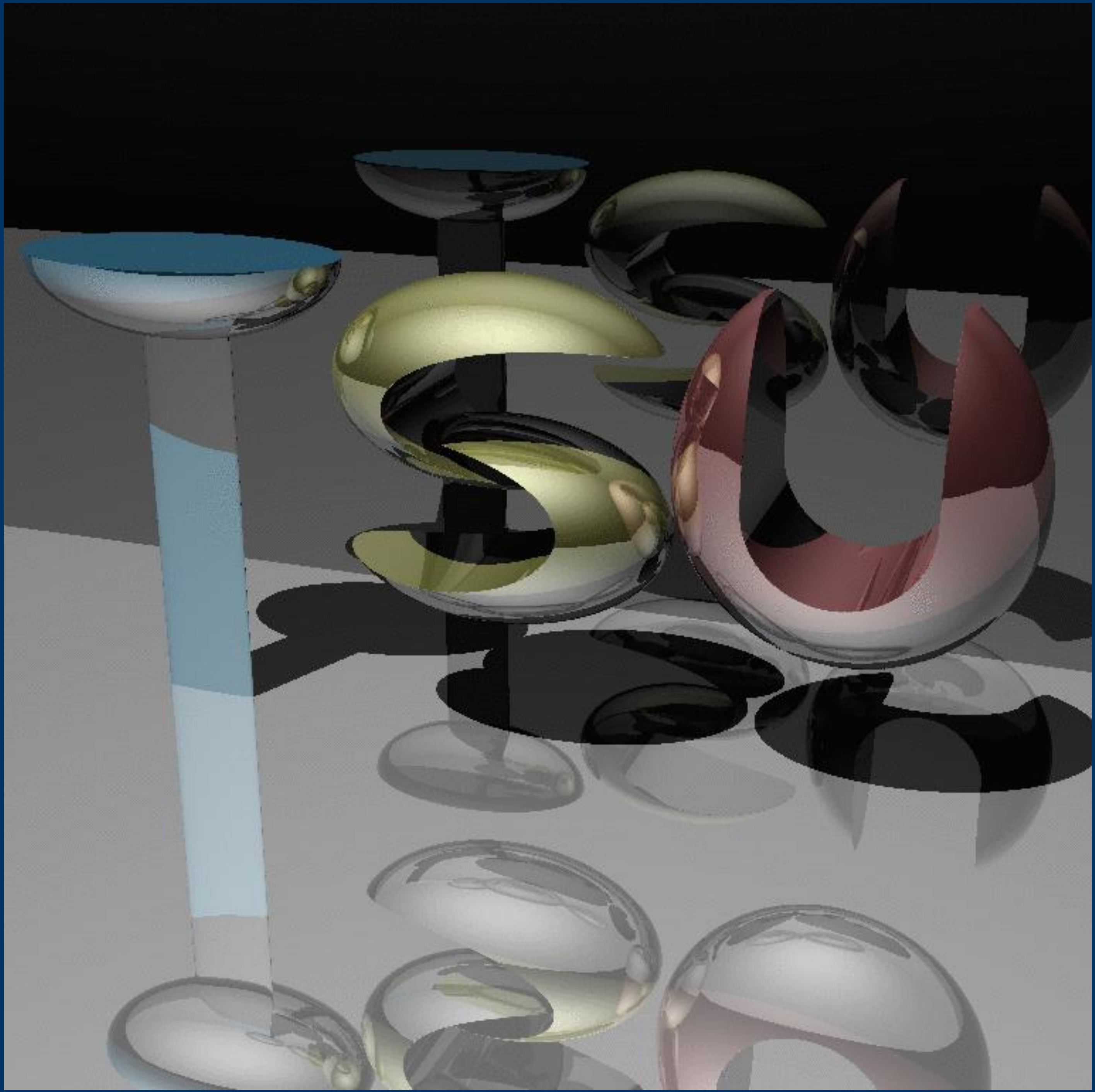
- CPU lab
  - Design and implement original CPUs by using VHDL and FPGA
- Compiler lab
  - Develop compilers for the original CPUs
  - ✓ MinCaml is used here!

$\Rightarrow$ Compete by the speed of ray tracing (5-6 students per group)



Special thanks to Hiraki laboratory





How is MinCaml Used?

- Students are given high-level descriptions of MinCaml
  - in Japanese and pseudo-code
- Each group is required to implement them
- Every student is required to solve small exercises
  - such as hand compilation

Outcome (1/2)

Students liked ML!

- Implemented polymorphism (like MLton), garbage collection, inter-procedural register allocation, etc. without being told
- Started a portal site (www.ocaml.jp) with Japanese translations of the OCaml manual without being told

Outcome (2/2)

"Outsiders" are also using MinCaml

- Somebody anonymous wrote a comprehensive commentary on MinCaml
- Ruby hackers organized an independent seminar to study MinCaml
- Prof. Asai is using MinCaml in Ochanomizu University

Outline of This Talk

- Pedagogical background
- Design and implementation of MinCaml
- Efficiency

Goals

- As simple as possible

but

- Able to efficiently execute non-trivial applications (such as ray tracing)

MinCaml: The Language

- Functional: no destructive update of <u>variables</u> (cf. SSA)
- Higher-order
- Call-by-value
- Impure
  - Input/output
  - Destructive update of <u>arrays</u>
- Implicitly typed
- Monomorphic

Syntax (1/2)

M, N (expressions) ::=

- c
- op(M_1, ..., M_n)
- if M then N_1 else N_2
- let x = M in N
- x
- let rec x y_1 ... y_n = M_1 in M_2
- M N_1 ... N_n (no partial application)
- ...

(cont.)

Syntax (2/2)

M, N (expressions) ::=

...

$(M_1, \ldots, M_n)$

let $(x_1, \ldots, x_n) = M$ in N (cf. $\#_i\ M$)

Array.create M N

M.(N)

$M_1.(M_2) \leftarrow M_3$

()

Literally implemented as ML data type Syntax.t

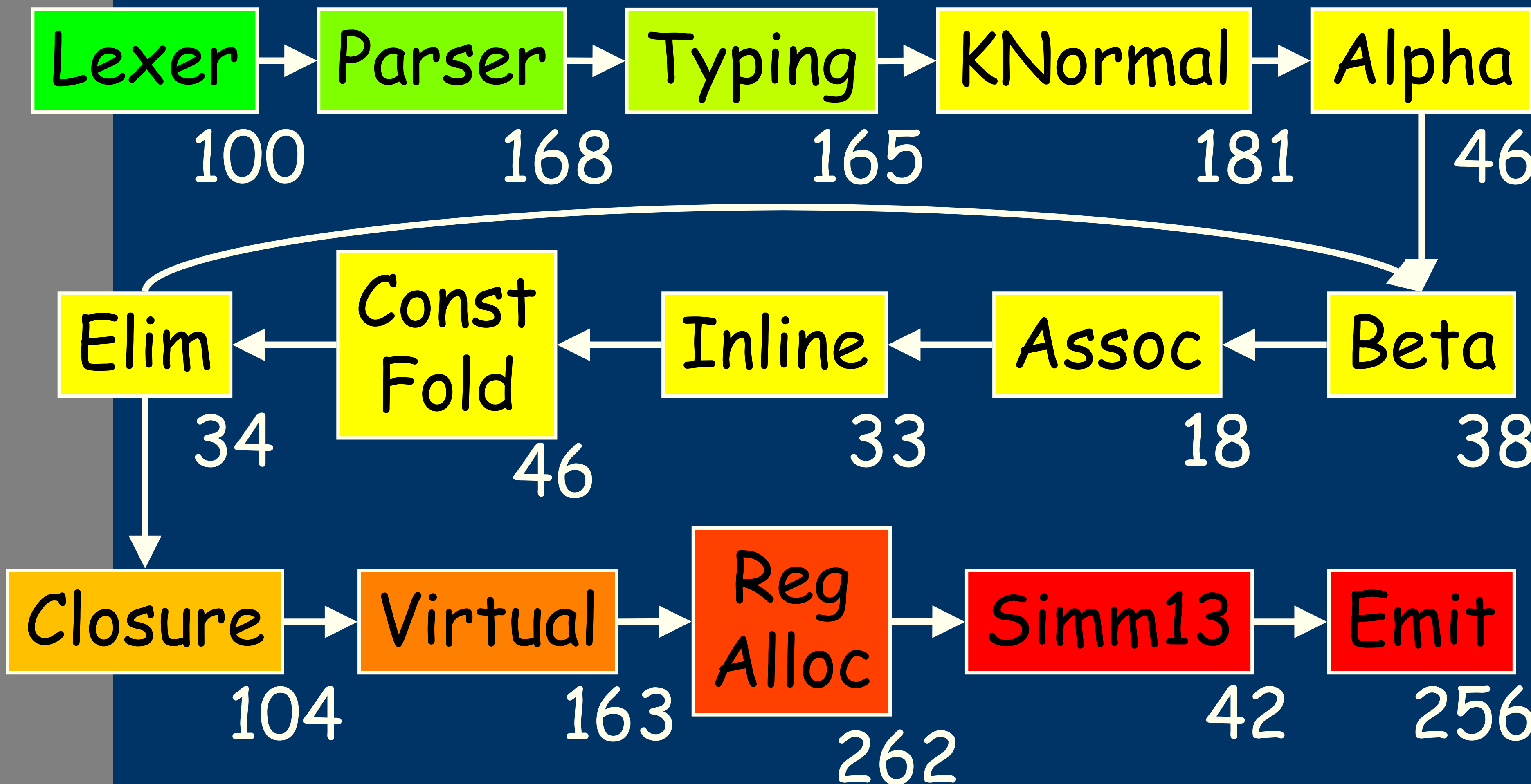
Everything else is omitted!

- Array boundary checking (easy)
- Garbage collection
- Data types and pattern matching
- Polymorphism
- Exceptions
- Objects

etc.

Optional homework
($\geq$ 2 compulsory from this year)

MinCaml: The Compiler



Lexing and Parsing

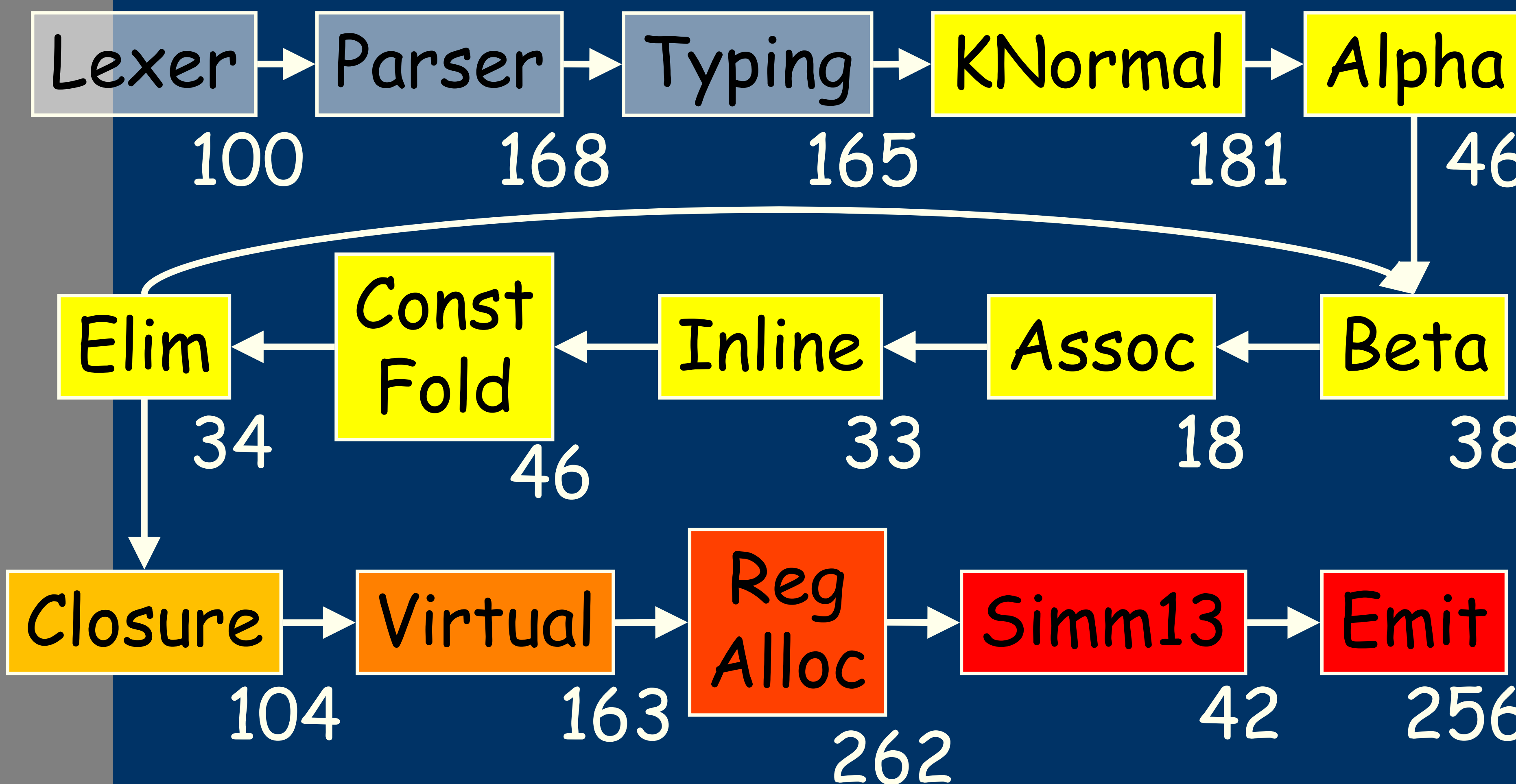
- Written in OCamlLex and OCamlYacc
- Given by the instructer
  - Algorithms are out of scope

  Cf. packrat parsing [Ford 2002]

Type Inference

- Based on standard unification using ML references
  - Let-polymorphic version is already taught in PL lab
- Free variables are treated as external functions (or arrays)
  - "Principal typing" [Jim 96] is automatically inferred

MinCaml: The Compiler



K-Normalization

$$a + b + c * d$$

$$\Downarrow$$

```
let tmp1 = a + b in
let tmp2 = c * d in
tmp1 + tmp2
```

- Nesting <u>is</u> allowed

  $$\text{let } x = (\text{let } y = M_1 \text{ in } M_2) \text{ in } M_3$$

  - Simplifies the normalization and inlining

  Cf. A-normalization by CPS

Syntax of K-Normal Form

$M, N ::=$

- c
- $op(x_1, \ldots, x_n)$
- if x then M_1 else M_2
- let $x = M$ in N
- x
- let rec $x\ y_1 \ldots y_n = M_1$ in M_2
- $x\ y_1 \ldots y_n$
- …

Implemented as KNormal.t

Algorithm of K-Normalization: Pseudo-Code Given to Students

$$K : \text{Syntax.t} \rightarrow \text{KNormal.t}$$

K(c) = c

K(op(M_1, ..., M_n)) =

let x_1 = K(M_1) in ... let x_n = K(M_n) in
op(x_1, ..., x_n)

K(if op(M_1, ..., M_n) then N_1 else N_2) =

let x_1 = K(M_1) in ... let x_n = K(M_n) in
if op(x_1, ..., x_n) then K(N_1) else K(N_2)

K(let x = M in N) = let x = K(M) in K(N)

K(x) = x

etc.

α-Conversion (Another Example of Pseudo-Code)

α : KNormal.t $\rightarrow$ Id.t Map.t $\rightarrow$ KNormal.t

$\alpha(c)\rho$ = c

$\alpha(op(x_1, \ldots, x_n))\rho$ = $op(\rho(x_1), \ldots, \rho(x_n))$

$\alpha(\text{if } x \text{ then } N_1 \text{ else } N_2)\rho$ =

  if $\rho(x)$ then $\alpha(N_1)\rho$ else $\alpha(N_2)\rho$

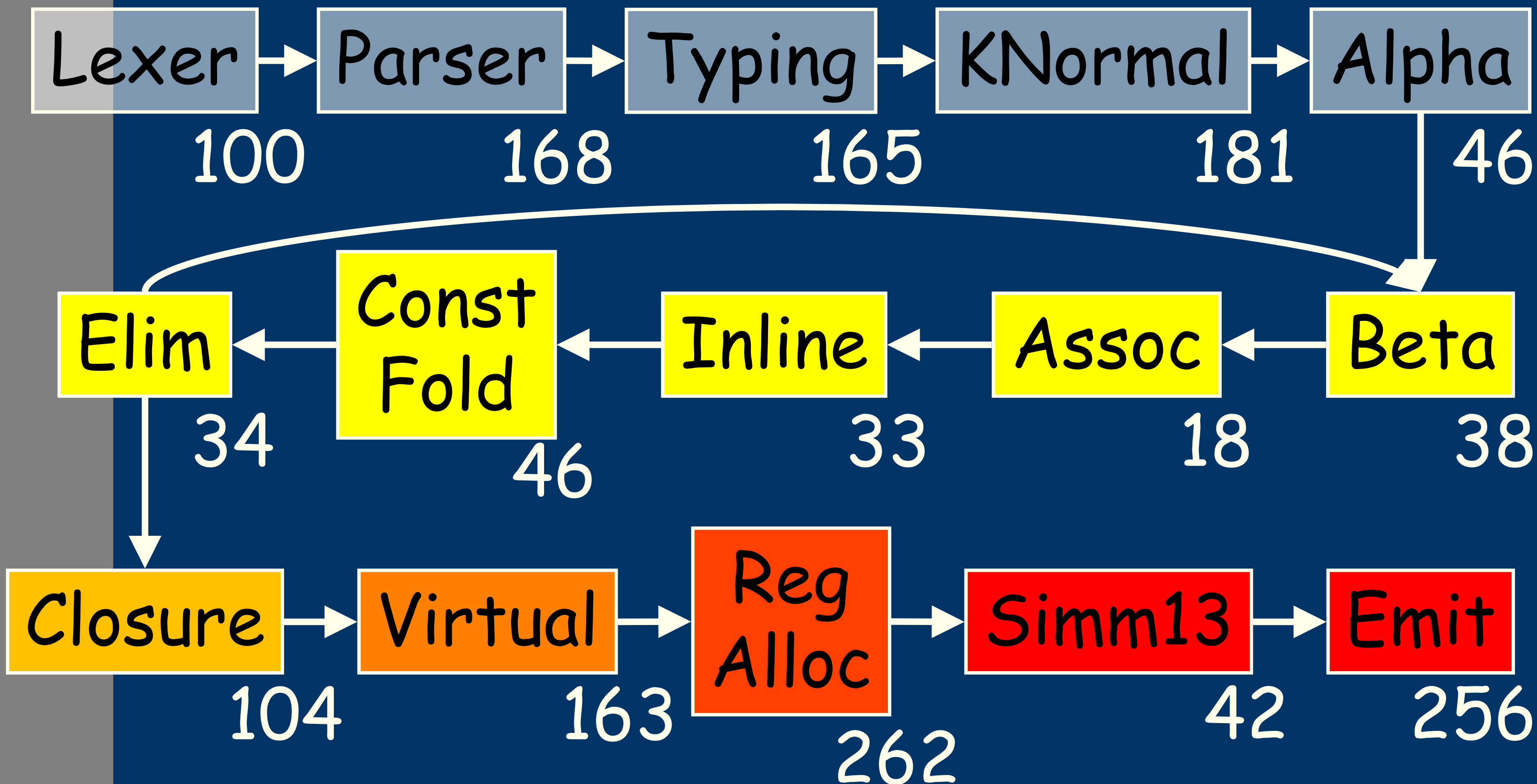
$\alpha(\text{let } x = M \text{ in } N)\rho$ = (x' fresh)

  let x' = $\alpha(M)\rho$ in $\alpha(N)\rho[x \rightarrow x']$

$\alpha(x)\rho$ = $\rho(x)$

etc.

MinCaml: The Compiler



β-Reduction

$$\boxed{\text{let } x = y \text{ in } M \quad \Rightarrow \quad [y/x]M}$$

- Pseudo-code (similar to previous examples) is left as an exercise

Nested "Let" Reduction

$$\text{let } y = (\text{let } x = M_1 \text{ in } M_2) \text{ in } M_3$$
$$\Downarrow$$
$$\text{let } x = M_1 \text{ in let } y = M_2 \text{ in } M_3$$

- Resembles A-normalization, but does not expand "if"

  $C[\text{if } M \text{ then } N_1 \text{ else } N_2]$
  $\Rightarrow \quad \text{if } x \text{ then } C[N_1] \text{ else } C[N_2]$

Inlining

Inlines all "small" functions

- Includes recursive ones
- "Small" = less than a constant size
  - User-specified by "-inline" option
- Repeat for a constant number of times
  - User-specified by "-iter" option

Constant Folding and Unused Variable Elimination

let x = 3 in let y = 7 in x + y

$\Downarrow$

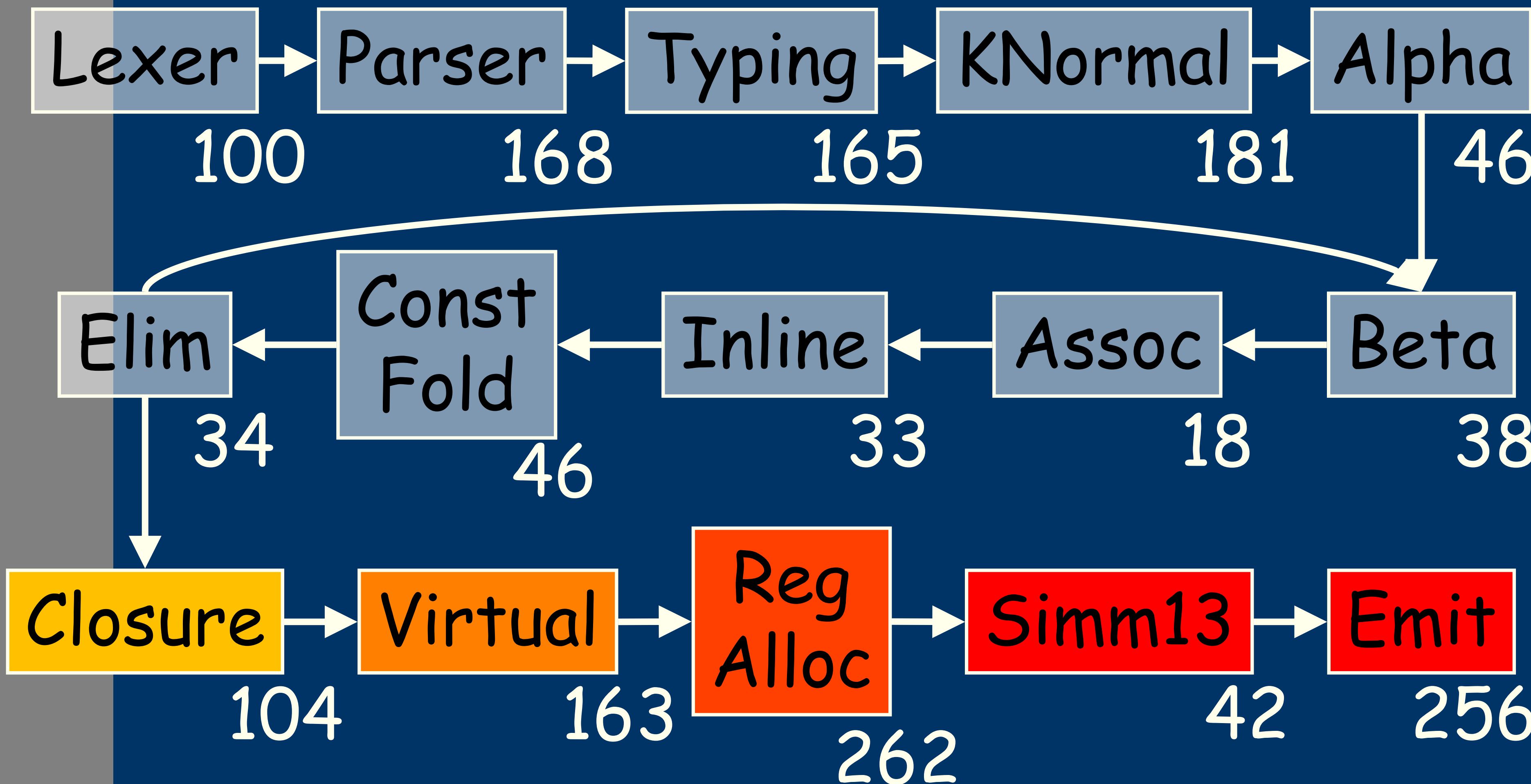
let x = 3 in let y = 7 in 10

$\Downarrow$

10

Effective after inlining

MinCaml: The Compiler



Closure Conversion

Local function definitions (let rec)
\+ function applications

$\Downarrow$

Top-level function definitions
\+
- Closure creations (make_closure)
- Closure applications (apply_closure)
- Known function calls (apply_direct)

Example 1: Closure Creation/Application

```
let x = 3 in
let rec f y = x + y in
f 7
```

$$\Downarrow$$

let rec f_{top} [x] y = x + y ;;

let x = 3 in
make_closure f = (f_{top}, [x]) in
apply_closure f 7

Example 2: Known Function Call

let rec f x = x + 3 in
(f, f 7)

$\Downarrow$

let rec f_{top} [] x = x + 3 ;;

make_closure f = (f_{top}, []) in
(f, apply_direct f 7)

Example 3: Unused Closure Elimination

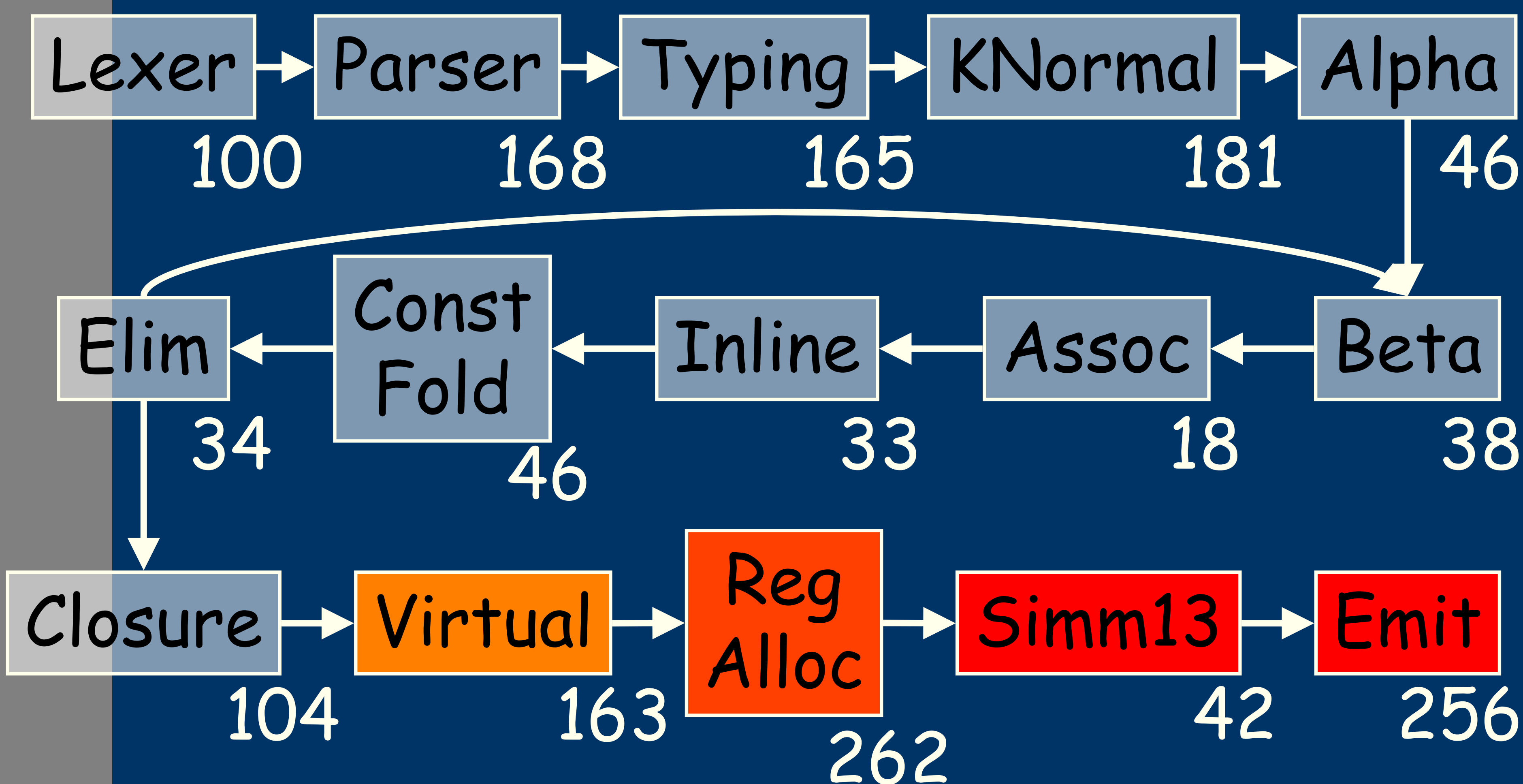
```
let rec f x = x + 3 in
f 7
```

$$\Downarrow$$

let rec f_{top} [] x = x + 3 ;;

```
apply_direct f 7
```

MinCaml: The Compiler



Virtual Machine Code Generation

SPARC assembly with:

- Infinite number of registers/variables
- Top-level function definitions and calls (call_closure, call_direct)
- Conditional expressions (if)

Tuple creations/accesses and closure creations are expanded to stores and loads

Register Allocation

Greedy algorithm with:

- Look-ahead for targeting

  let x = 3 in let y = 7 in f y x

  $\Rightarrow$ let r_2 = 3 in let r_1 = 7 in f r_1 r_2

- Backtracking for "early save"

  let x = 3 in

  ...; f (); ...; x + 7

  $\Rightarrow$ let r_1 = 3 in

  save(r_1, x); ...; f (); ...; restore(x, r_2); r_2 + 7

13-Bit Immediate Optimization

- Specific to SPARC
- “Inlining” or "constant folding" for integers from -4096 to 4095

```
set 123, %r1
add %r1, %r2, %r3
```

$\Downarrow$

```
add %r2, 123, %r3
```

Assembly Generation

Lengthy (256 lines)
but easy

- Tail call optimization
- Stack map computation
- Register shuffling
  - Somewhat tricky but short (11 lines)

Outline of This Talk

- Pedagogical background
- Design and implementation of MinCaml
- Efficiency

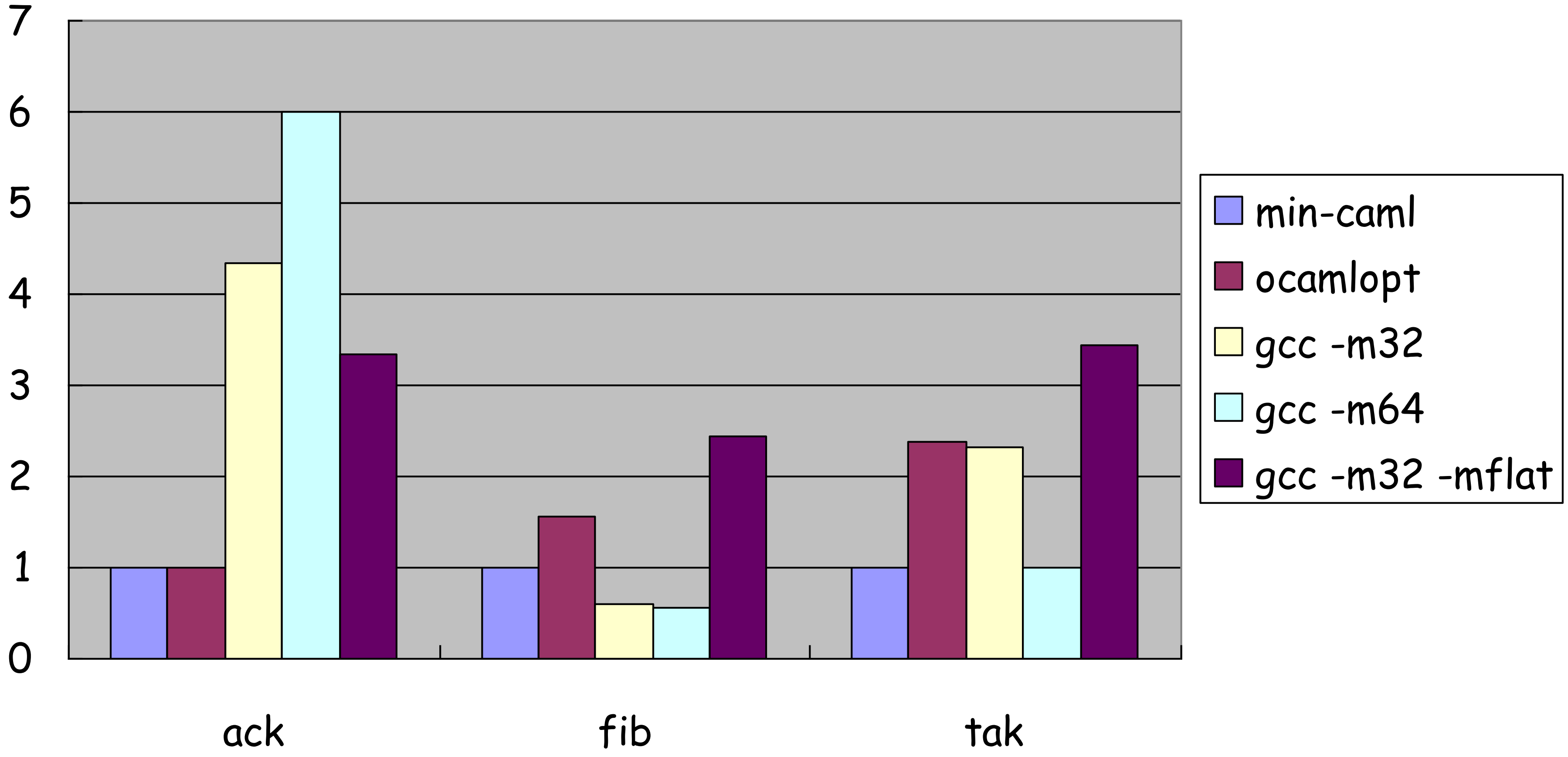
Environment

- Machine: Sun Fire V880
  - 4 Ultra SPARC III 1.2GHz
  - 8 GB main memory
  - Solaris 9
- Compilers:
  - MinCaml (32 bit, -iter 1000 -inline 100)
  - OCamlOpt 3.08.3 (32 bit, -unsafe -inline 100)
  - GCC 4.0.0 20050319 (32 bit and 64 bit, -O3)
  - GCC 3.4.3 (32 bit "flat model", -O3)

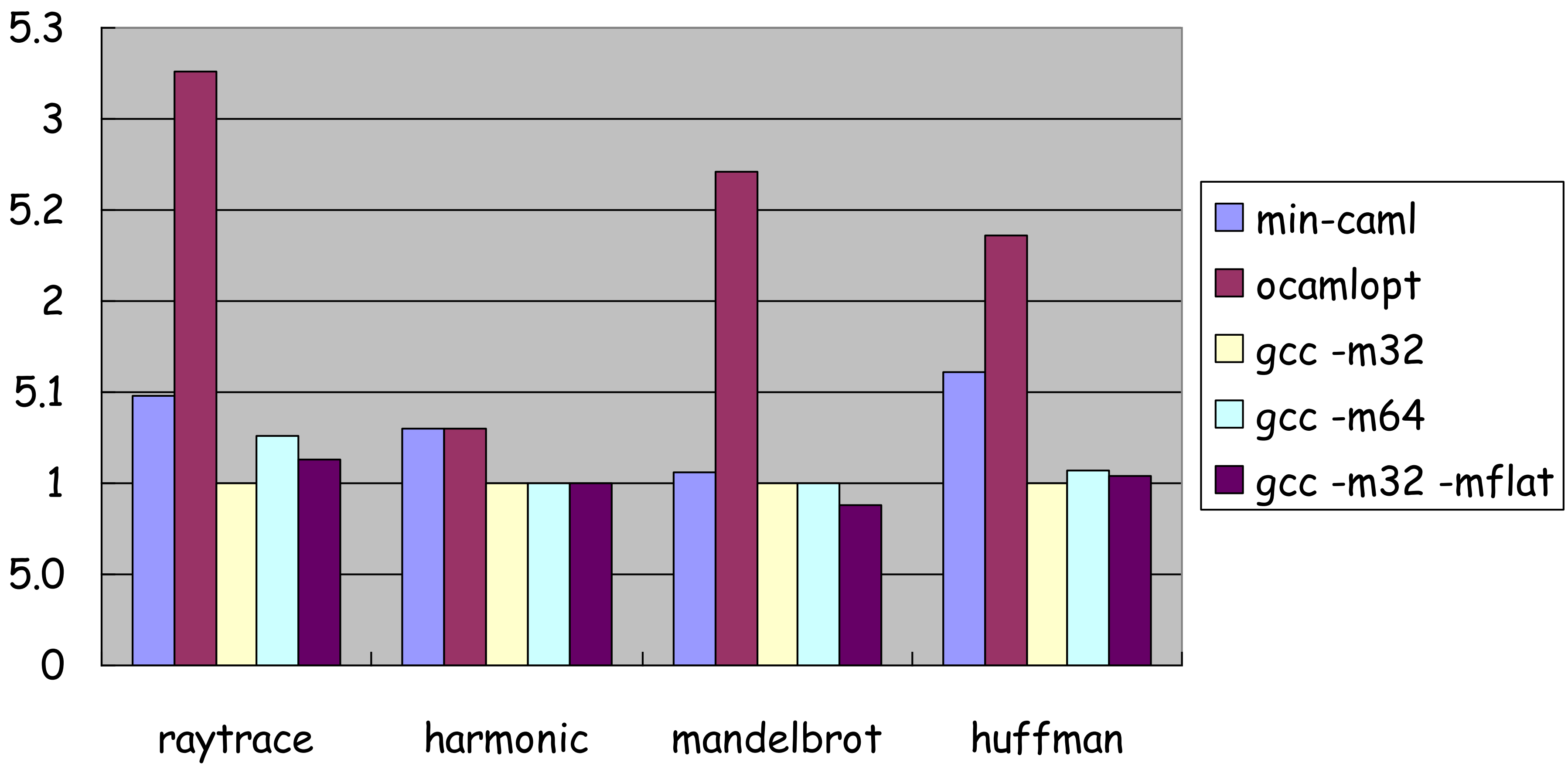
Applications

- Functional
  - Ackermann
  - Fibonacci
  - Takeuchi
- Imperative
  - Ray tracing
  - Harmonic function
  - Mandelbrot set
  - Huffman encoding

Execution Time of Functional Programs (min-caml = 1)



Execution Time of Imperative Programs
(gcc -m32 = 1)



Summary

"Simple and efficient compiler for a minimal functional language"

Future work:

- Improve the register allocation
  - By far more complex than other modules
  - Too slow at compile time
- Retarget to IA-32
  - 2-operand instructions (which are "destructive" by definition) and FPU stack

http://min-caml.sf.net/