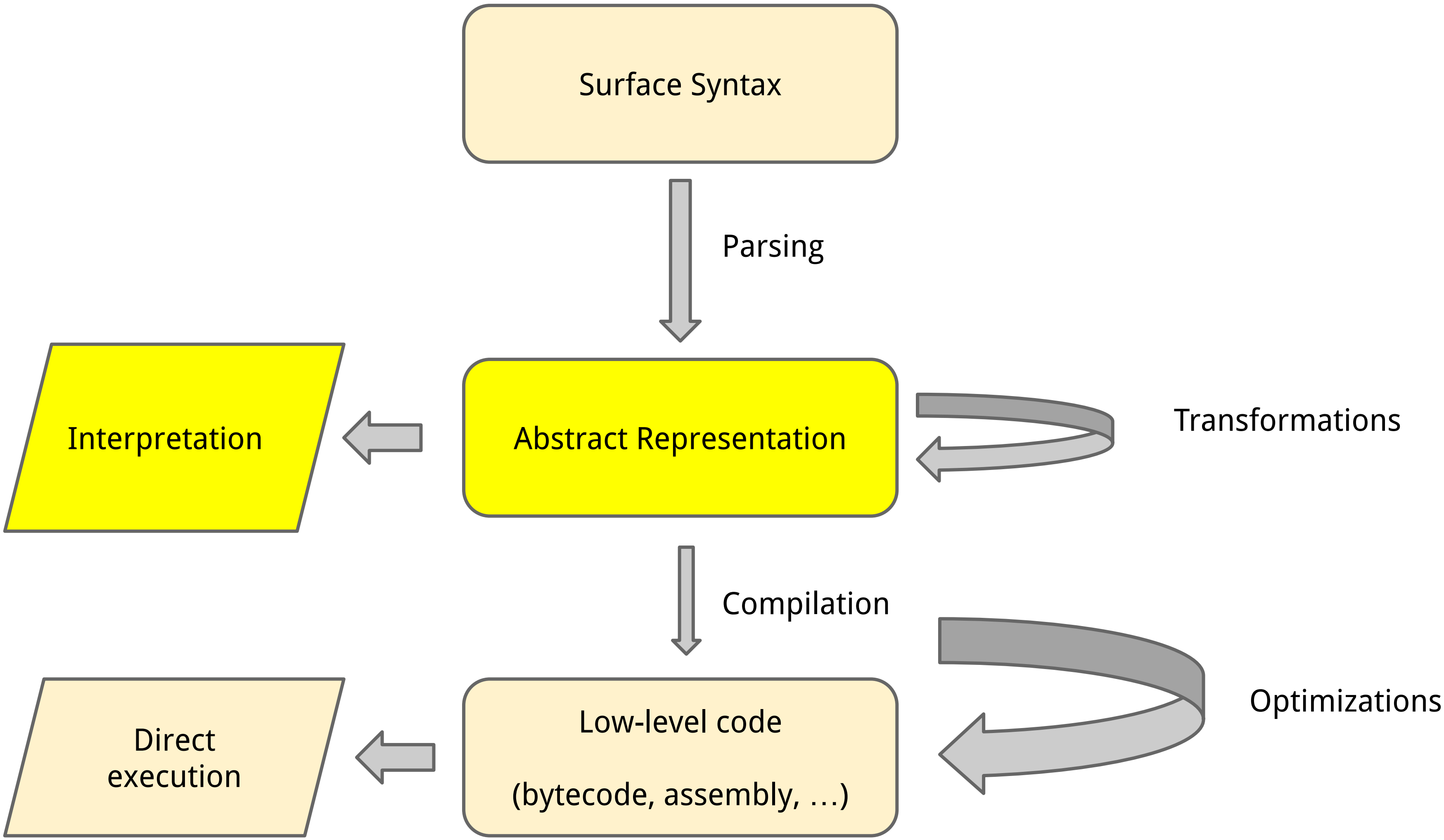
Compilation (I)

April 16, 2020

Riccardo Pucella

Interpretation

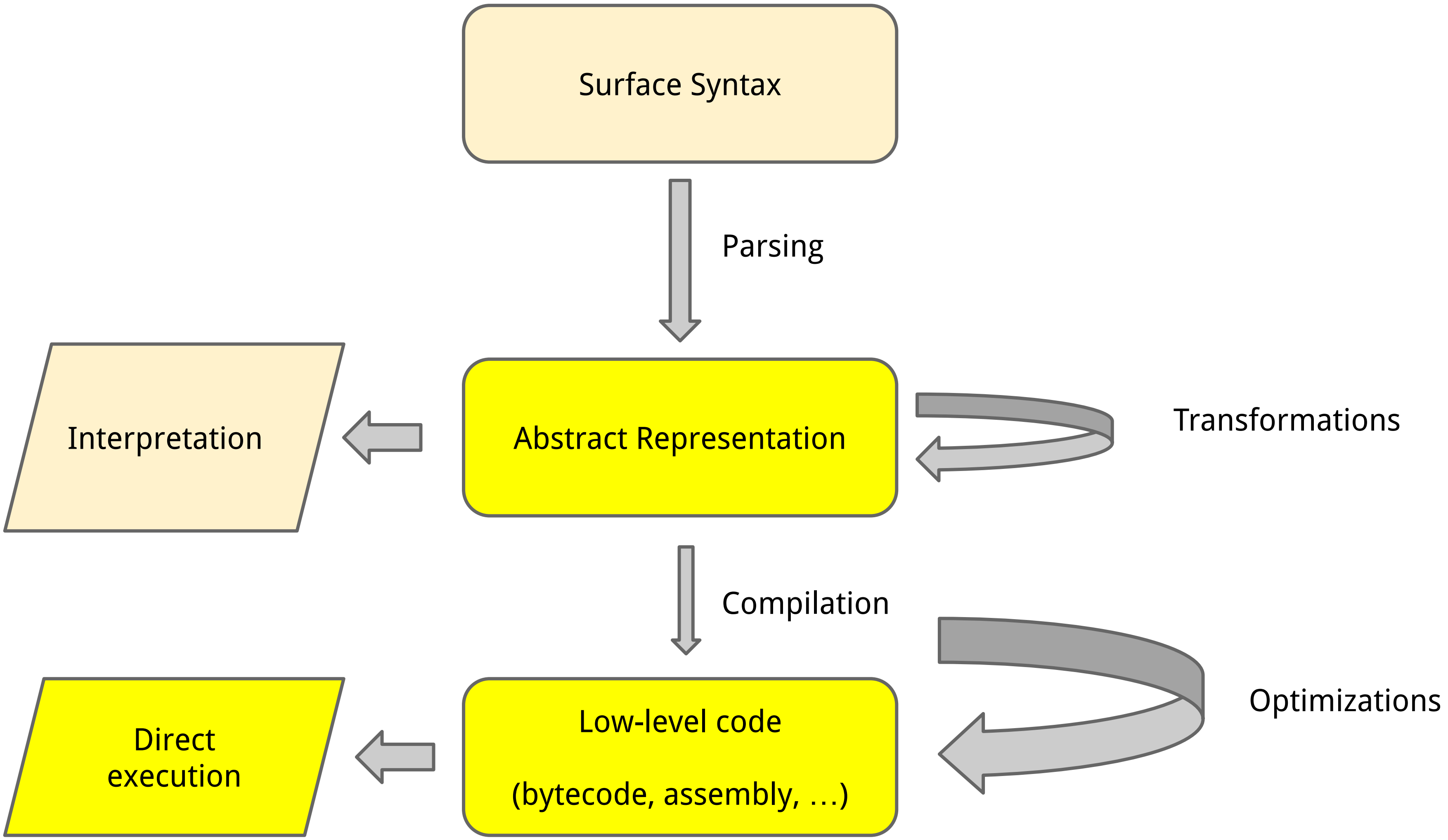


Interpretation

- Abstract representation is high level
- Abstract representation is recursive

- Recursively traverse the abstract representation, apply evaluation rules
- Need a lot of infrastructure to evaluate (environments, recursive calls, etc)

Compilation



Compilation

Convert abstract representation into a more low-level representation

- flat (non-recursive)
- easier to execute / less infrastructure

Often assembly/machine language

- either for a real CPU or a virtual one
- JVM, Web Assembly

Types of virtual machines

Two models of machines:

- register-based
  - Use registers to hold intermediate data
- stack-based
  - Use stack to hold intermediate data

We develop a simple stack-based virtual machine for FUNC

A simple stack machine

There is a stack — initially empty

A program is a sequence of instructions:

- `PUSH(n)` - push `n` on the stack
- `ADD`

  pop 2 items off the stack, push sum
- `MULT`

  pop 2 items off the stack, push product

After execution, result is item on top of stack

Examples

$(3 \times 2) + 1 \Rightarrow$

```
PUSH(1) PUSH(3) PUSH(2) MULT ADD
```

```
PUSH(3) PUSH(2) MULT PUSH(1) ADD
```

$(3 \times 2) + (4 \times 5) \Rightarrow$

```
PUSH(3) PUSH(2) MULT PUSH(4) PUSH(5) MULT ADD
```

Our virtual machine

Program:

- array of instructions (opcodes)

Storage:

- An environment variable ENV
- A program counter variable PC
  - (address of next instruction to execute)
- A stack holding:
  - values, addresses, environments

Execution:

- execute instruction at the PC
- may modify the stack
- update the PC

Instructions

STOP	stop and return value on top of stack
PUSH-INT(i)	push i on the stack
PUSH-ADDR(a)	push a on the stack
PUSH-ENV	push ENV on the stack
JUMP	pop a and jump to a
JUMP-TRUE	pop a and v and jump to a if v <> 0
CLOSURE	pop a and push closure (a, ENV)
OPEN	pop (a, e), set ENV = e, push a
ENV	pop e, set ENV = e
ADD-ENV(n)	pop v, add (n, v) to ENV
LOOKUP(n)	lookup (n, v) in ENV and push v
PRIM-CALL(i, op)	pop i values, call op, push result
NOP	do nothing
SWAP	swap top two values on the stack
COPY(n)	push the (n+1)th stack value on the stack

Example: arithmetic

$(1 + 2) * 3 \quad \Rightarrow$

```
000 PUSH-INT(2)
001 PUSH-INT(1)
002 PRIM-CALL(2, oper_plus)
003 PUSH-INT(3)
004 PRIM-CALL(2, oper_times)
005 STOP
```

Example: summation

Roughly:

```
((fun s (n result) (if (= n 0) result (s (+ n -1) (+ n result)))) 200 0)
```

```
000  PUSH-ENV
001  PUSH-INT(0)
002  PUSH-INT(200)

003  ADD-ENV(n)
004  ADD-ENV(result)
005  LOOKUP(n)
006  PUSH-INT(0)
007  PRIM-CALL(2, oper_equal)
008  PUSH-ADDR(020)
009  JUMP-TRUE
010  LOOKUP(n)
011  LOOKUP(result)
012  PRIM-CALL(2, oper_plus)
013  LOOKUP(n)
014  PUSH-INT(-1)
015  PRIM-CALL(2, oper_plus)
016  COPY(2)
017  ENV
018  PUSH-ADDR(003)
019  JUMP

020  LOOKUP(result)
021  STOP
```

How do we compile?

- Define a function C
  - taking an expression E
  - producing a sequence of opcodes

- When executing the produced opcodes:
  - if starting stack is S
  - after execution, stack looks like r :: S
  - r = the result of evaluating E

Compiling integers

C[`EInteger(n)`] ⇒

```
PUSH-INT(n)
```

Compiling Booleans

C[`EBoolean(true)`] ⇒

```
PUSH-INT(1)
```

C[`EBoolean(false)`] ⇒

```
PUSH-INT(0)
```

Compiling identifiers

C[`EId(n)`] ⇒

```
LOOKUP(n)
```

Compiling conditionals

C[`EIf(c, t, e)`] ⇒

```
  C[ c ]
  PUSH-ADDR(@then)
  JUMP-TRUE
  C[ e ]
  PUSH-ADDR(@done)
  JUMP
@then:
  C[ t ]
@done:
  NOP
```

Compiling functions

C[`EFunction(self, [p1, p2, ...], body)`] ⇒

```
  PUSH-ADDR(@after)
  JUMP
@fun:
  PUSH-ADDR(@fun)
  CLOSURE
  ADD-ENV(self)
  ADD-ENV(p1)
  ADD-ENV(p2)
  ...
  C[ body ]
  SWAP
  JUMP
```

```
@after:
  PUSH-ADDR(@fun)
  CLOSURE
```

```
Change code to match
sample demo
```

Compiling applications

C[`EApply(f, [e1, e2, ...])`] ⇒

```
  PUSH-ENV
  PUSH-ADDR(@return)
  ...
  C[ e2 ]
  C[ e1 ]
  C[ f ]
  OPEN
  JUMP
@return:
  SWAP
  ENV
```

Virtual machine runtime

What do we need to run the compiled code?

- An implementation of environments
- An implementation of closures
- Initial code for primitives:

```
000  PRIM-CALL(2, oper_plus)
001  SWAP
002  JUMP
003  PRIM-CALL(2, oper_times)
004  SWAP
005  JUMP
```

- Initial environment mapping primitives to closures with the above addresses

Next time

- Simplify the virtual machine
- Implement it in C-like language
- Optimize the code
- Talk about memory management