Stack Languages

March 4, 2014

Riccardo Pucella

Until now

We’ve studied basic interpreters

- Simple expression languages
- Functional languages as an obvious generalization

BUT

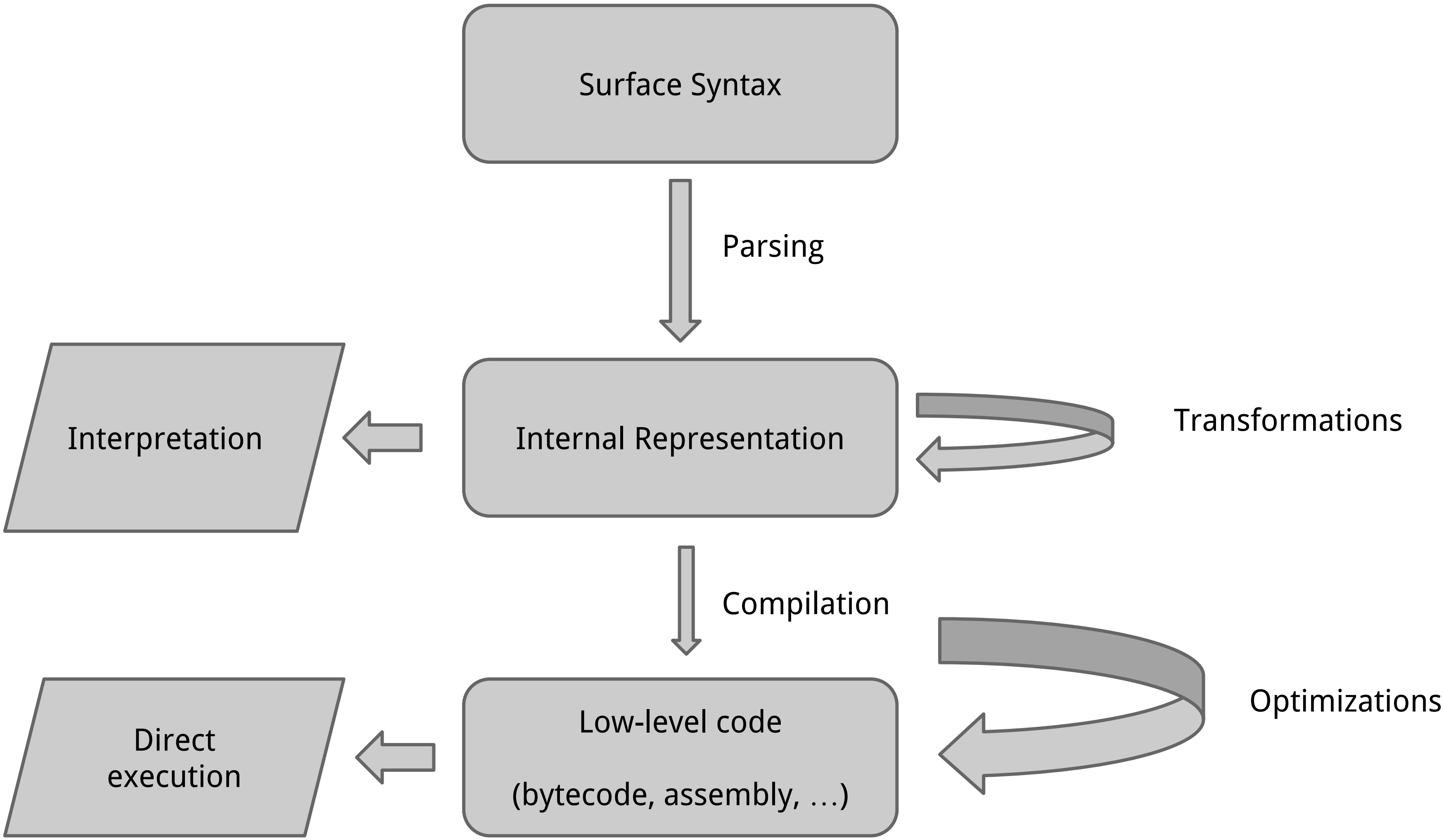
Our interpreters are doing a lot of work!

Two (potential) problems

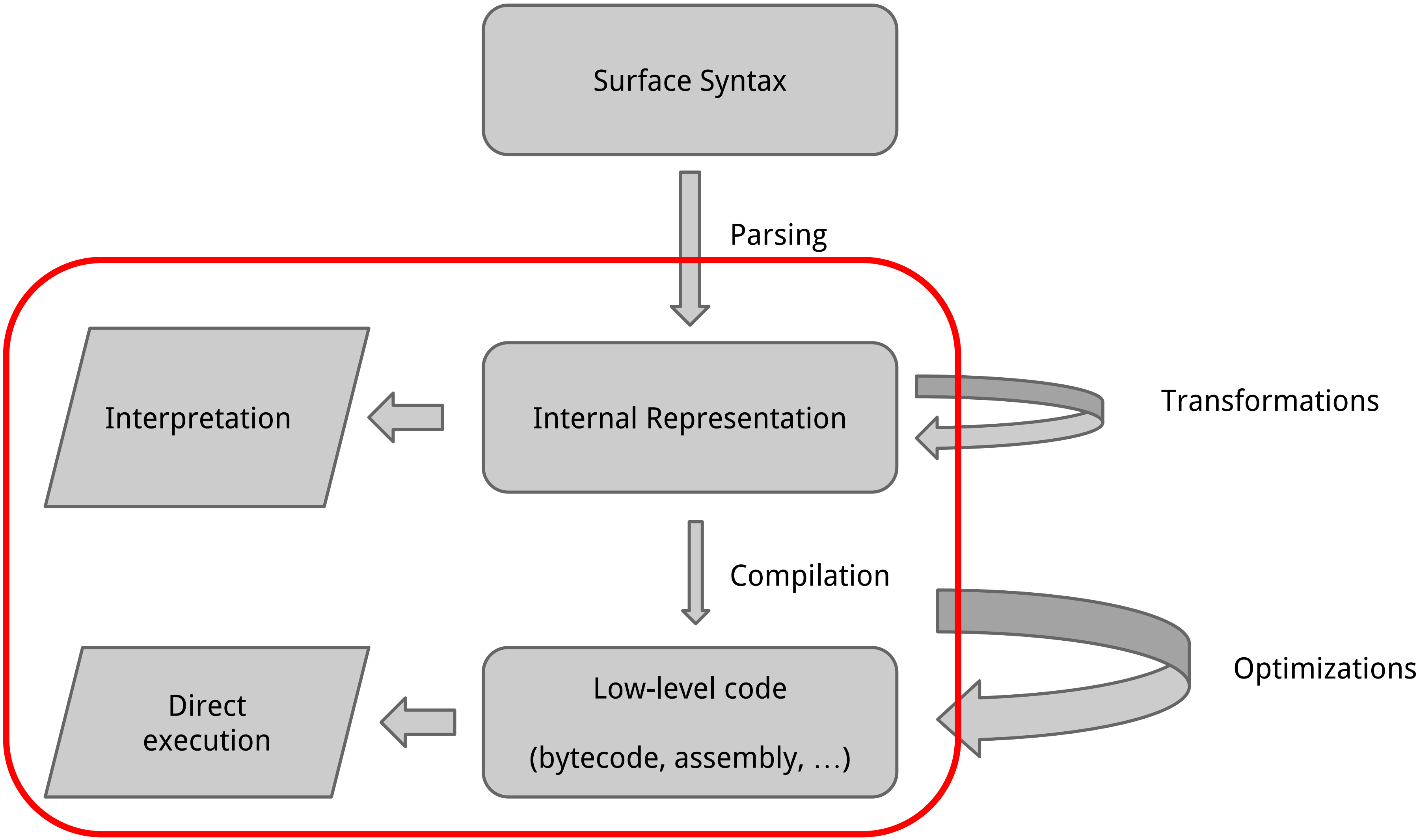
- Interpreters are inefficient
  - same code interpreted over and over again
    - functions, loops
  - lookup names in environment

- Execution infrastructure is heavy
  - primitives are complex
  - require a nontrivial infrastructure to support the interpreter
  - closures, environments, recursive evaluator

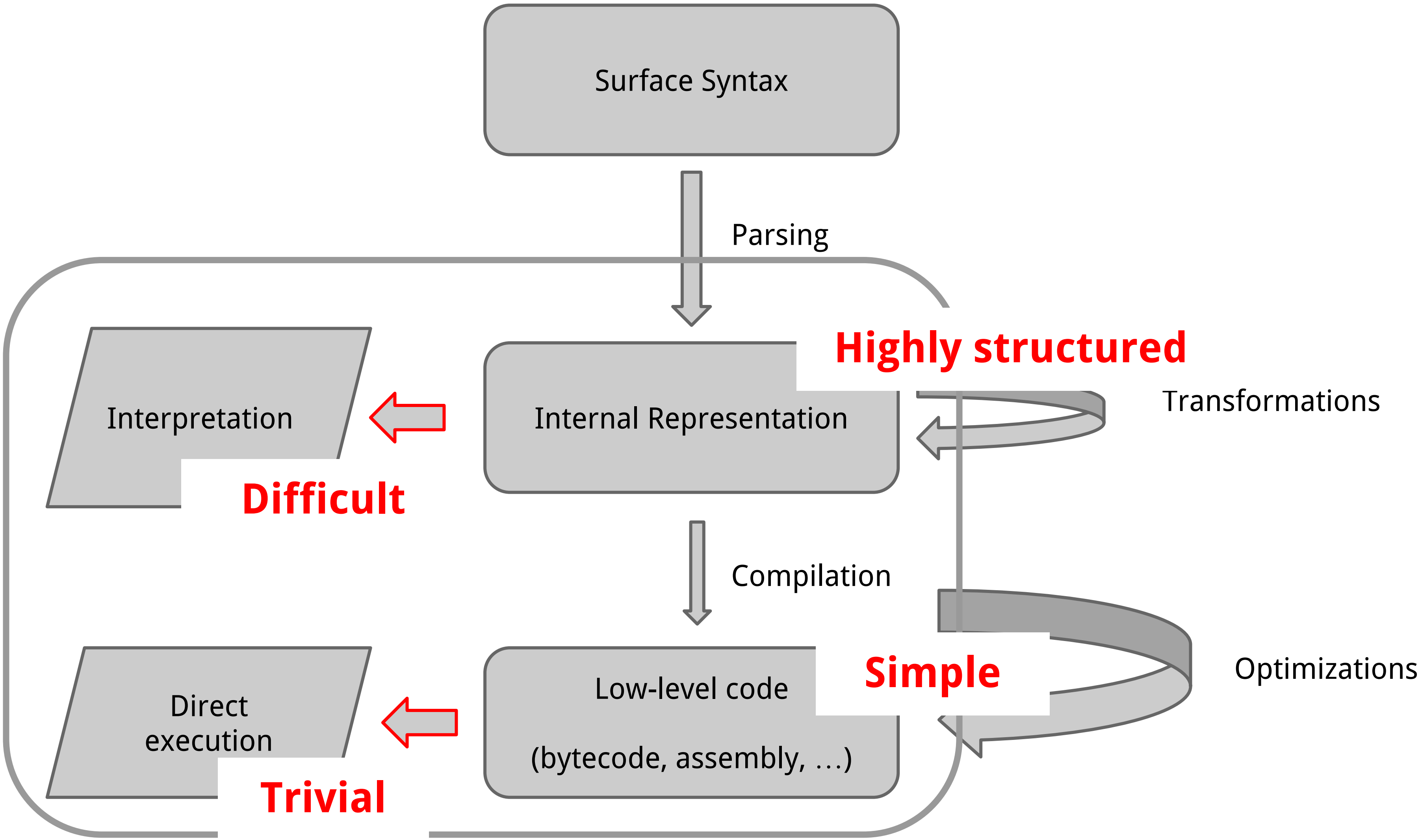
The structure of language execution



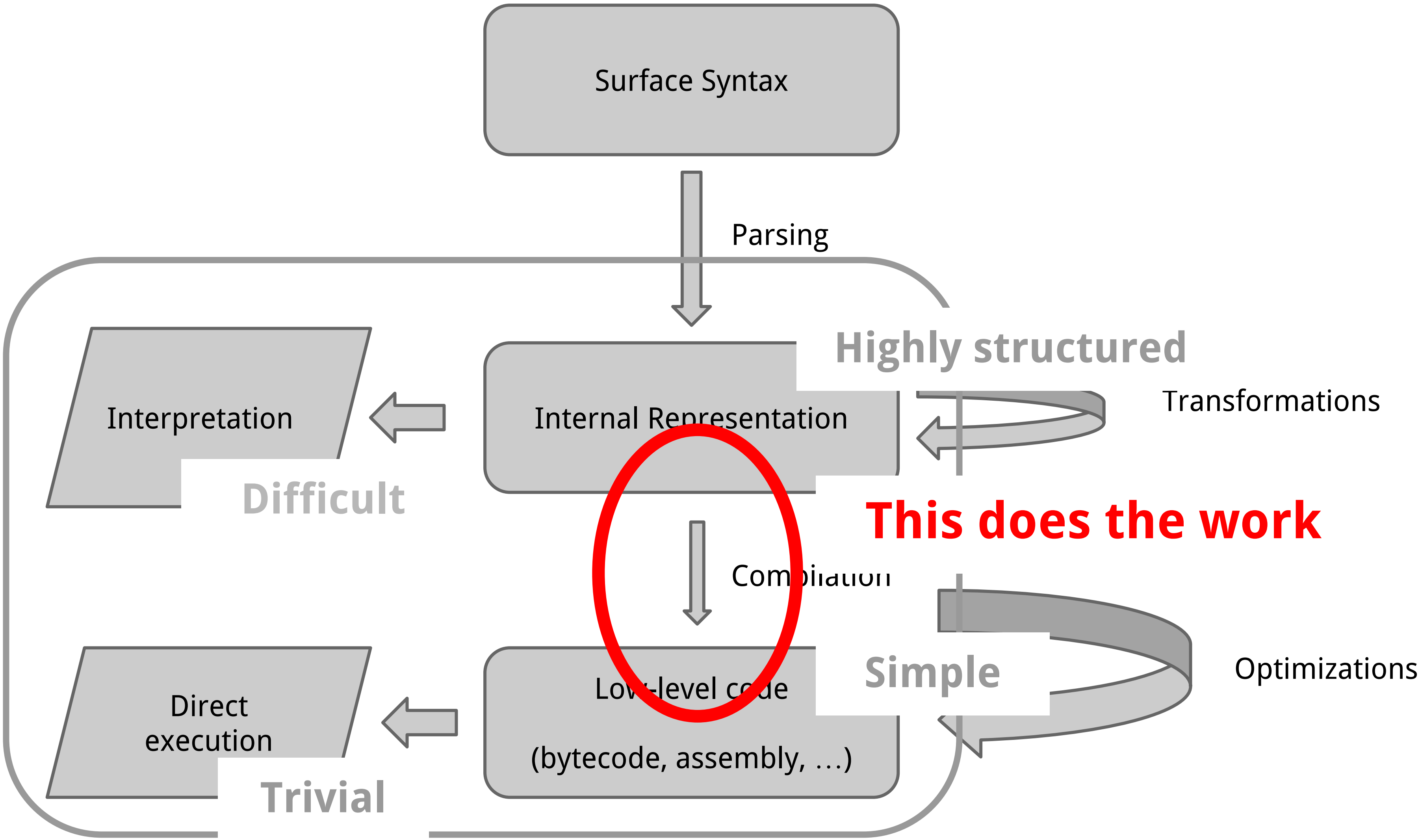
The structure of language execution



The structure of language execution



The structure of language execution



Stack Languages

- Simple yet interesting
- Easy to compile for
    - Basis of the Java Virtual Machine
- Easy to execute
    - Basis of embedded languages (FORTH)
- Can be compiled further

A FORTH-like language

Execution model:

- Program is a sequence of instructions
- Instructions manipulate the stack
  - expect their arguments on the stack
  - leave result on the stack

That’s it!

  - <mumbles something about control flow>

A FORTH-like language

Execution model:

(words)

- Program is a sequence of instructions
- Instructions manipulate the stack
    - expect their arguments on the stack
    - leave result on the stack

That's it!

- <mumbles something about control flow>

Literals

Literals just push themselves onto the stack

```
stack> 10
10
stack> 20
20 10
stack> 30
30 20 10
stack> nil
[] 30 20 10
```

(Only integers and lists)

Arithmetic operations

Arithmetic operations   +, *, -, mod

- pop two arguments on the stack
- push result onto the stack

```
stack> 10
10
stack> 20
20 10
stack> +
30
stack> 40
40 30
stack> *
1200
```

Chaining words

Sequences of words executed from left to right — updating the stack with each word

```
stack> 10 20 + 40
40 30 1200
stack> * +
2400
```

Stack operations

Operations to manipulate the stack

dup, drop, swap, over, rot

```
stack> 10 20 30 40
40 30 20 10
stack> swap
30 40 20 10
stack> over
40 30 40 20 10
stack> drop
30 40 20 10
stack> rot
20 30 40 10
```

Definitions

At the shell, introduced by `:` *`defined-word`*

```
stack> : square dup *
stack> 10
10
stack> square
100
stack> square
10000
stack> : sum-of-squares square swap square +
stack> 20
20 10000
stack> sum-of-squares
100000400
```

Conditionals

Conditionals are one example of *control flow*

```
IF execute if true THEN
IF execute if true ELSE execute if false THEN
```

Based on value on top of stack (0 is false)

```
stack> 0 if 10 else 20 then
20
stack> if 10 else 20 then
10
stack> 30 0= if 10 else 20 then
20 10
```

Recursion

Nothing special: just a word defined in terms of itself

```
stack> : -1 1 swap -
stack> 20
20
stack> -1
19
stack> : fact dup 0= if drop 1 else dup -1 fact * then
stack> 3 fact
6
stack> fact
720
```

Lists

An addition to our language

- though it’s easy to add to FORTH
- nil, cons, head, tail, nil=

```
stack> nil 1 cons
[1]
stack> 2 cons
[2,1]
stack> dup
[2,1] [2,1]
stack> head
2 [2,1]
stack> drop tail
[1]
```

Lists

An addition to our language

- though it's easy to add to FORTH
- nil, cons, head, tail, nil=

```
stack> nil 1 c
[1]
stack> 2 cons
[2,1]
stack> dup
[2,1] [2,1]
stack> head
2 [2,1]
stack> drop tail
[1]
```

```
stack> : split dup tail swap head
Definition split added to environment
stack> nil 2 cons 1 cons
[1,2]
stack> split
1 [2]
```

Recursion on lists

```
stack> : length dup nil= if drop 0 else tail length 1 +
then
Definition length added to environment
stack> nil 2 cons 1 cons
[1,2]
stack> length
2
stack> nil 3 cons 1 cons
[1,3] 2
stack> swap cons
[2,1,3]
stack> length
3
```

Exercises

Internal representation

```
datatype sentence = SEmpty
                  | SSequence of word * sentence
                  | SIf of sentence * sentence * sentence

         and word = WInt of int
                  | WPrim of (value list -> value list)
                  | WDefined of string

datatype value = VInt of int
                 | VList of value list
```

Internal representation

```
datatype sentence = SEmpty
```

```
datat
```

Parsing is trivial:

```
decl ::= T_COLON T_WORD sentence
         sentence

sentence ::= T_IF sentence T_THEN sentence
             T_IF sentence T_ELSE sentence T_THEN sentence
             word sentence
             word
             <empty>

word ::= T_INT
         T_WORD
```

Execution function

```
fun execute env I.SEmpty stack = stack
  | execute env (I.SSequence (I.WInt i,ws)) stack =
      execute env ws ((I.VInt i)::stack)
  | execute env (I.SSequence (I.WPrim prim, ws)) stack =
      execute env ws (prim stack)
  | execute env (I.SSequence (I.WDefined w, ws)) stack = let
      val stack = execute env (lookup w env) stack
    in    execute env ws stack  end
  | execute env (I.SIf (trueS,falseS,thenS)) (v::stack) = let
      val stack = execute env (case v
                                of I.VInt 0 => falseS
                                 | _ => trueS) stack
    in   execute env thenS stack   end
  | execute _ _ _ = evalError "cannot execute"
```

Execution function

```
fun execute env I.SEmpty stack = stack
  | execu
      e
  | exe
      e
  | exe
      v
    in
  | exe                                                       t
      v

    in
  | exe
```

Sample primitive:

```
(“+”, SSequence (WPrim primAdd, SEmpty))
```

```
  fun primAdd ((I.VInt i)::(I.VInt j)::stack) =
                   (I.VInt (i+j))::stack
    | primAdd _ = evalError "primAdd"
```

Next time

We “simplify” our interpreter:

— eliminate the use of “expensive” recursion in `execute`

— eliminate the use of lists to represent word definitions

Think about how you might go about doing the above