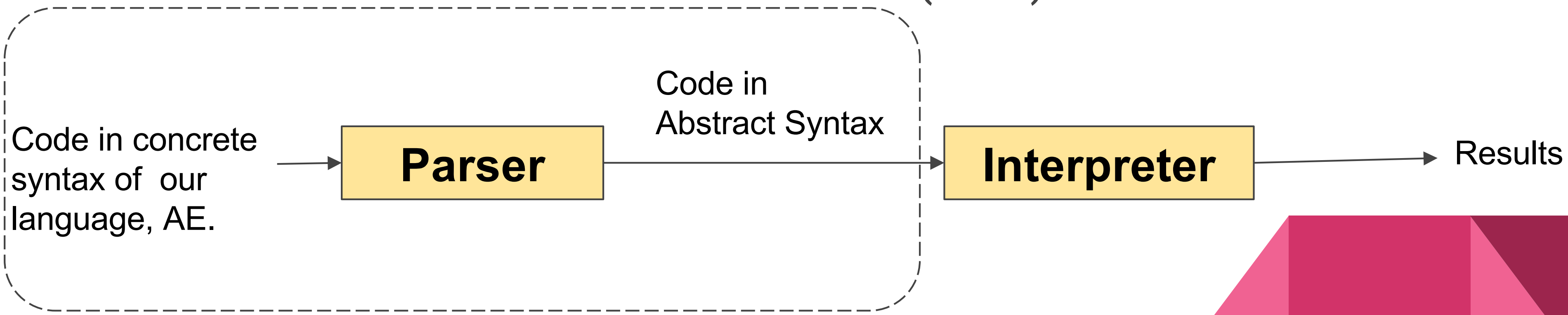
ITP20005
Modeling Languages (2)
(Parsing and Interpreting Arithmetic)

Lecture05
JC

Parser

Parser

- A parser is a component in an interpreter or compiler.
  - Identifies what kinds of program it is examining, and
  - Converts concrete syntax into abstract syntax.
- To do this, we need a clear specification of the concrete syntax of the language!!
  - How to specify???
    - We use Backus-Naur Form (BNF)



Example: A Grammar for Arithmetic Expressions

- Example syntax of new arithmetic expressions (AE) we want to use.
  {+ {- 3 4 } 7}
- Specify in BNF

```
<AE> ::= <num>
       | {+ <AE> <AE>}
       | {- <AE> <AE>}
```

- Abstract syntax representation (tree) in Racket

```
(define-type AE
  [num (n number?)]
  [add (lhs AE?)
       (rhs AE?)]
  [sub (lhs AE?)
       (rhs AE?)])
```

*** Example usages based on AE.**

```
(define ae1 (add (sub (num 3) (num 4)) (num 7)))
(sub? ae1)            ; Checking type

; retrieving expressions
(add-rhs ae1)
(sub-rhs (add-lhs ae1))
```

BNF captures both the concrete syntax and a default abstract syntax!

(That is why BNF has been used in definitions of languages.
Let's see some examples...)

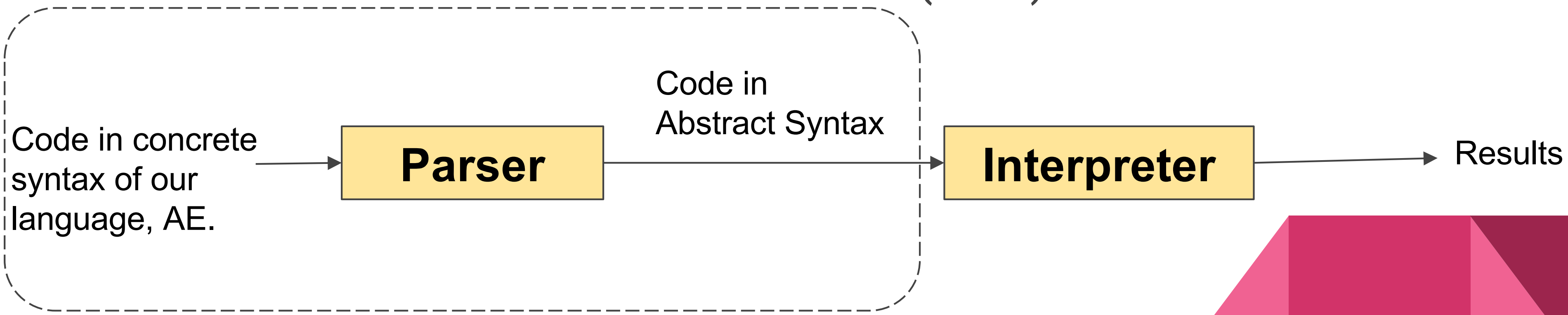
https://users-cs.au.dk/amoeller/RegAut/JavaBNF.html

https://cs.wmich.edu/~gupta/teaching/cs4850/sumII06/The%20syntax%20of%20C%20in%20Backus-Naur%20form.htm

https://docs.python.org/3/reference/grammar.html

Parser

- A parser is a component in an interpreter or compiler.
  - Identifies what kinds of program it is examining, and
  - Converts concrete syntax (what we type) into abstract syntax.
- To do this, we need a clear specification of the concrete syntax of the language!!
  - How to specify???
    - We use Backus-Naur Form (BNF)



Parser for Arithmetic Expressions

```
;; parse : sexp -> AE
;; to convert s-expressions into AEs in abstract syntax

;; tests
(test (parse '3) (num 3))
(test (parse '{+ 3 4}) (add (num 3) (num 4)))
(test (parse '{- 4 3}) (sub (num 4) (num 3)))
(test (parse '{+ {+ 4 3} {- 4 3}}) (add (add (num 4) (num 3)) (sub (num 4) (num 3))))
```

* sexp: sub-expression which is just source code

Parser for Arithmetic Expressions

```
;; parse : sexp -> AE
;; to convert s-expressions into AEs in abstract syntax
(define (parse sexp)
        (cond
                [(number? sexp) (num sexp)]
                [(eq? (first sexp) '+) (add (parse (second sexp))
                                                        (parse (third
sexp)))]
                [(eq? (first sexp) '-) (sub (parse (second sexp))
                                                        (parse (third
sexp)))]
                ))

(test (parse '3) (num 3))
(parse '{+ 3 4}) ;; our code must start with a single quote to deal with them as symbols!
(test (parse '{+ 3 4}) (add (num 3) (num 4)))

;; how about this?
```

Parser for Arithmetic Expressions

```
;; parse : sexp -> AE
;; to convert s-expressions into AEs
(define (parse sexp)
        (cond
                [(number? sexp) (num sexp)]
                [(and (= 3 (length sexp))
                          (eq? (first sexp) '+))
                 (add (parse (second sexp))
                          (parse (third sexp)))]
                [(and (= 3 (length sexp))
                          (eq? (first sexp) '-))
                 (sub (parse (second sexp))
                          (parse (third sexp)))]
                [else (error 'parse "bad syntax: ~a" sexp)]))
```

Parser for Arithmetic Expressions

```
(define (parse sexp)
    (cond
        [(number? sexp) (num sexp)]
        [(and (= 3 (length sexp))           (eq? (first sexp) '+))
         (add (parse (second sexp))    (parse (third sexp)))]
        [(and (= 3 (length sexp))           (eq? (first sexp) '-))
         (sub (parse (second sexp))         (parse (third sexp)))]
        [else (error 'parse "bad syntax: ~a" sexp)]))

(test (parse '3) (num 3))
(test (parse '{+ 3 4}) (add (num 3) (num 4)))
(test (parse '{+ {- 3 4} 7}) (add (sub (num 3) (num 4)) (num 7)))
(test/exn (parse '{- 5 1 2}) "parse: bad syntax: (- 5 1 2)")
```

Complete implementation for the parser

```
#lang plai
;; type definition for AE
(define-type AE
  [num (n number?)]
  [add (lhs AE?)
       (rhs AE?)]
  [sub (lhs AE?)
       (rhs AE?)])
```

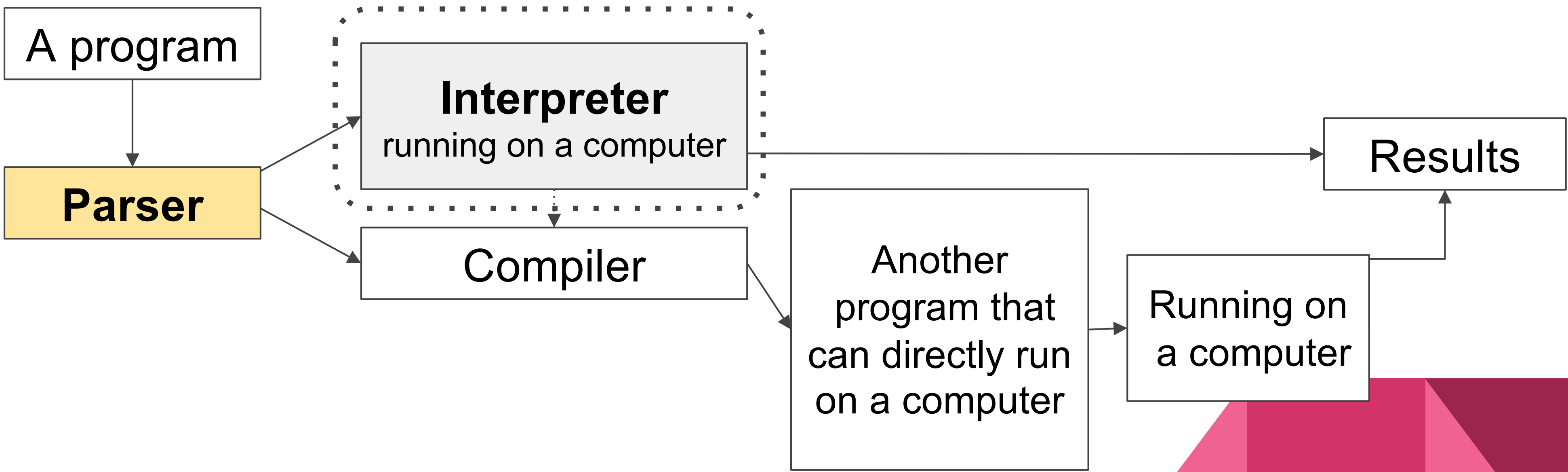
```
;; [contract] parse: sexp -> AE
;; [purpose] to convert s-expressions into AEs
(define (parse sexp)
  (cond
   [(number? sexp) (num sexp)]
   [(and (= 3 (length sexp))
         (eq? (first sexp) '+))
    (add (parse (second sexp))
         (parse (third sexp)))]
   [(and (= 3 (length sexp))
         (eq? (first sexp) '-))
    (sub(parse(second sexp))
        (parse(third sexp)))]
   [else (error 'parse "bad syntax:~a" sexp)]))

(test (parse '3) (num 3))
(test (parse '[+ 3 4]) (add (num 3) (num 4)))
(test (parse '{+ {- 3 4} 7}) (add (sub (num 3) (num 4)) (num 7)))

(test/exn (parse '{- 5 1 2}) "parse: bad syntax: (- 5 1 2)")
```

Big Picture (modeling languages)

- Just write an interpreter to explain a language.
- By writing an interpreter, we can understand the language!
- Interpreter can be converted into a compiler!!!



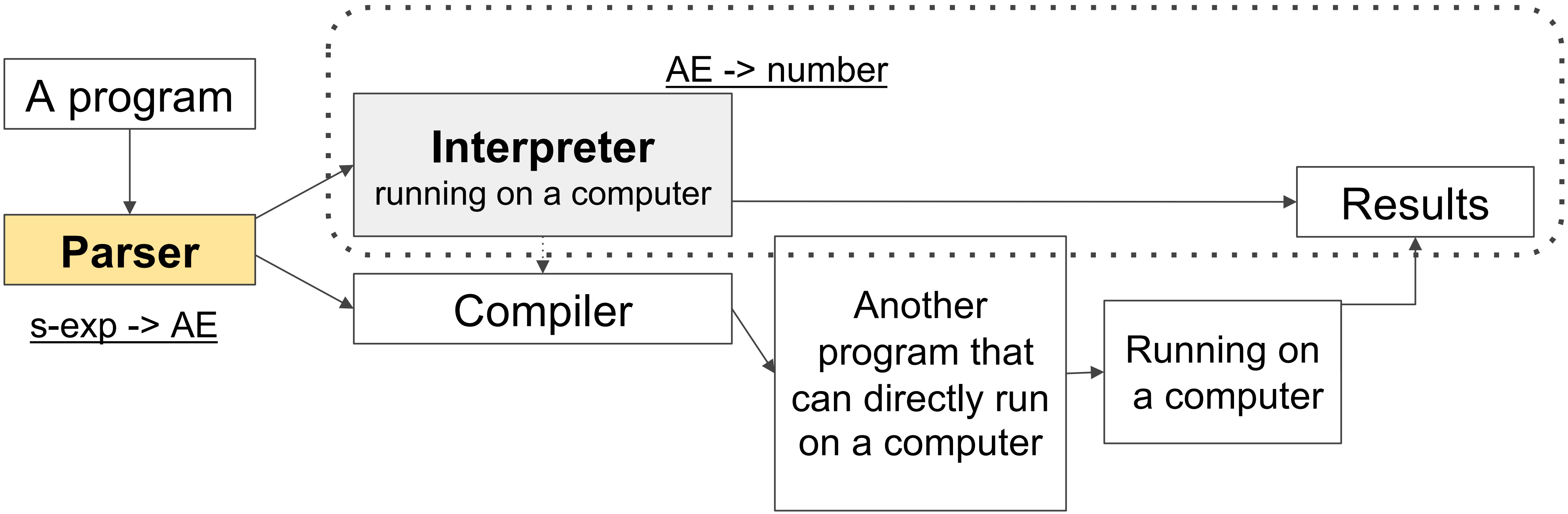
An Interpreter

for Arithmetic Expressions (AE)

An Interpreter
for Arithmetic Expressions (AE)

;; [contract] interp: AE -> number
;; [Purpose] consumes an AE and compute the corresponding number.

Big Picture (modeling languages)



Type Deconstruction for the AE interpreter

- Type Deconstruction is an important technique to easily implement an interpreter to deal with code in abstract syntax semantically.

(type-case *type-id expr*
[*variant_id*$_1$ (*field_id*$_{11}$ ...) expr$_1$]
...
[*variant_id*$_m$ (*field_id*$_{m1}$ ...) expr$_m$]

; interp: AE -> number

Type Deconstruction for the AE interpreter

(type-case *type-id expr*
    [*variant_id*$_1$ (*field_id*$_{11}$ …) expr$_1$]
    …
    [*variant_id*$_m$ (*field_id*$_{m1}$ …) expr$_m$]

```
; interp: AE -> number
(define (interp an-ae)
        (type-case AE an-ae
                ;; n is recognized as actual number for computers
                [num (n) n]
                ;; add is recognized as a real behavior to sum two AEs.
                [add (l r) (+ (interp l) (interp r))]
                ;; sub is recognized as a real behavior to subtract two AEs.
                [sub (l r)      (- (interp l) (interp r))]))
```

Type Deconstruction for the AE interpreter

```
(type-case type-id expr
        [variant_id1   (field_id11 ...) expr1]
        ...
        [variant_idm   (field_idm1 ...) exprm]
```

```
; ... : AE -> ...
(define (... an-ae)
    (type-case AE an-ae
        [num (n) ...]
        [add (l r) ...]
        [sub (l r)      ...]))
```

Template for AE

Type Deconstruction for the AE interpreter

- Do we need type-case? Why?

```
; interp: AE -> number
(define (interp an-ae)
     (cond
          [(num? an-ae) (num-n an-ae)]
          [(add? an-ae) (+ (interp (add-lhs an-ae))
                                         (interp (add-
rhs an-ae)))]
          [(sub? an-ae) (- (interp (sub-lhs an-ae))
                                         (interp (sub-rhs
an-ae)))]))
```

Type Deconstruction for the AE interpreter

- Do we need type-case? Why?

```
; interp: AE -> number
(define (interp an-ae)
    (type-case AE an-ae
            [num (n) n]
            [add (l r) (+ (interp l) (interp r))]
            [sub (l r) (- (interp l) (interp r))]))
```

Recall...the Design Recipe for functions

- Contract (Signature)

  ; area-of-ring: number number -> number
- Purpose

  ; to compute the area of a ring whose radius is
  ; outer and whose hole has a radius of inner
- Tests

  (test (area-of-ring 5 3) 50.24)
- Header

  (define (area-of-ring outer inner)
- Body

```
(- (area-of-disk outer)
   (area-of-disk inner)))
```

How to design an interpreter

- Determine the data representation
  - define-type (e.g., *AE*)
- Write tests
  - test (e.g., *(test (interp (parse '{+ 1 2})) 3)*)
- Create a template for the implementation
  - type-case for an interpreter
- Finish implementation case-by-case
- Run tests

Interpreter for Arithmetic Expressions

```
; interp : AE -> number
; to get results from AE
(define (interp an-ae)
        (type-case AE an-ae)
                [num (n) n]
                [add (1 r) (+ (interp l) (interp r))]
                [sub (1 r) (- (interp l) (interp r))]))

(test (interp (parse '3)) 3)
(test (interp (parse '{+ 3 4})) 7)
(test (interp (parse '{+ {- 3 4} 7})) 6)
```

We just implemented a program that consumes programs!

Practice more!

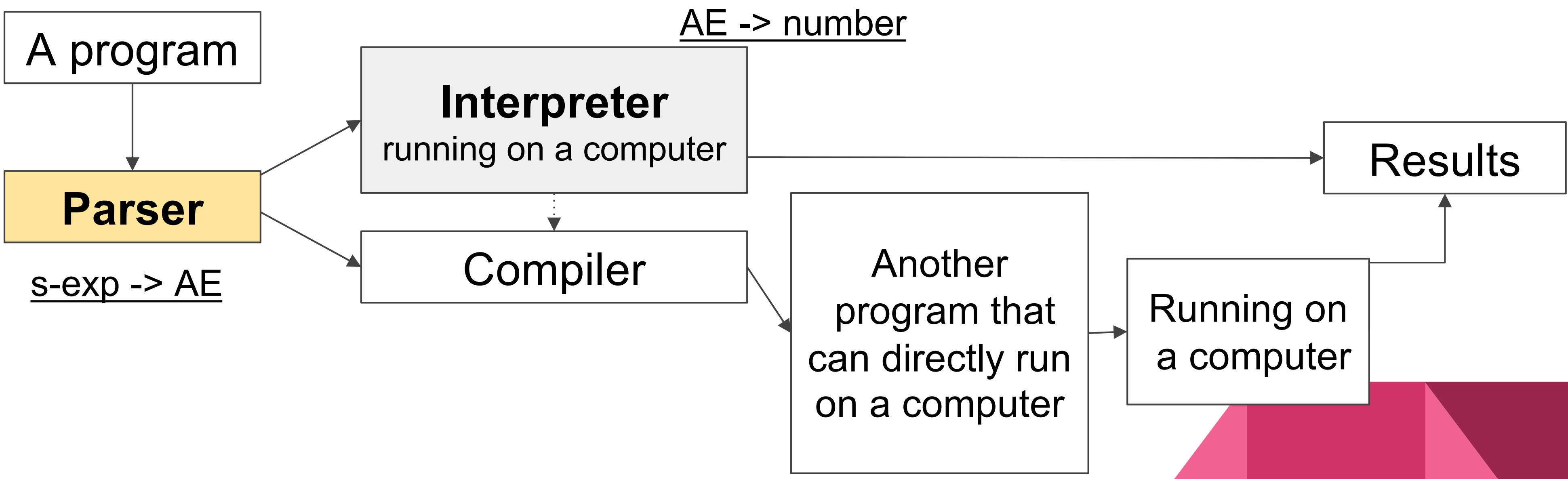
- Can you implement an AE parser for syntax based on infix or postfix?
  - Infix: (2 + (9 - 2))
  - Postfix: (2 (9 2 -) +)

Perhaps, we can even write programs that generate programs!

(Do you know what it is?)

Big Picture (modeling languages)

- Just write an interpreter to explain a language.
- By writing an interpreter, we can understand the language!
- Interpreter can be converted into a compiler!!!



Topics we cover and schedule (tentative)

- Racket tutorials (L2,3)
- Modeling languages (L4,5)
- Interpreting arithmetic (L5)
- Language principles
    - Substitution (L6-7)
    - Function (L8)
    - Deferring Substitution (L9)
    - First-class Functions (L10-L12)
    - Laziness (L13,14)
    - Recursion (L15,16)
    - Mutable data structures (L17,18,19,20)
    - Variables (L21,22)
    - Continuations (L23-26)
- Guest Video Lecture (L27)

TODO

Read Chapter 3. Substitution

http://cs.brown.edu/~sk/Publications/Books/ProgLangs/2007-04-26/plai-2007-04-26.pdf

2nd edition:

http://cs.brown.edu/courses/cs173/2012/book/From_Substitution_to_Environments.html

JC

jcnam@handong.edu

https://lifove.github.io

* Slides are from Prof. Sukyoung Ryu's PL class in 2018 Spring or created by JC based on the main text book.