L5. Modeling Languages 2 (Parsing and Interpreting Arithmetic)

유형	강의
복습 여부	☑
Status	Done

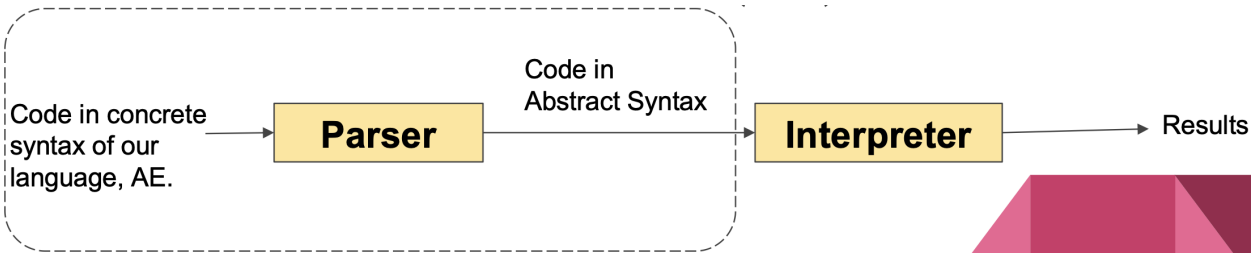
Parser

A parser is a component in an interpreter or compiler.

Identifies what kinds of program it is examining.

Converts concrete syntax (what we type) into abstract syntax.

To clearly specify the concrete syntax of the language, 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))
```

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

; {+ {- 2 1} 3} => 4
(define ae1 (add (sub (num 2) (num 1)) (num 3)))
; ae1 is an instance of add instance, not sub instance
(sub? ae1)
; get the left-hand side of ae1 instance
(add-lhs ae1)
; get the right-hand side of ae1 instance
(add-rhs ae1)
; get the right-hand side of left-hand side of ae1 instance
(sub-rhs (add-lhs ae1))
```

```
#f
(sub (num 2) (num 1))
(num 3)
(num 1)
```

```
;; [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)")
```

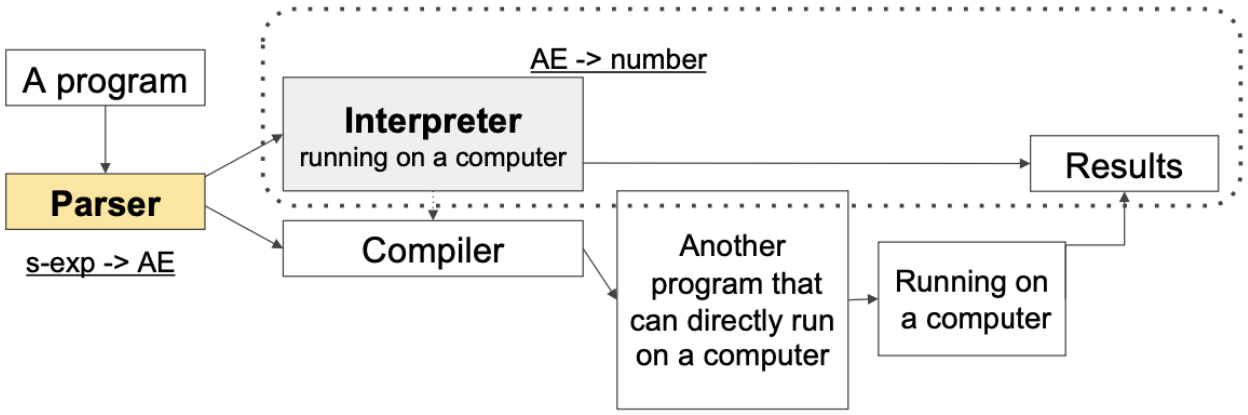
```
good (parse '3) at line 33
  expected: (num 3)
  given: (num 3)

good (parse '(+ 3 4)) at line 34
  expected: (add (num 3) (num 4))
  given: (add (num 3) (num 4))

good (parse '(+ (- 3 4) 7)) at line 35
  expected: (add (sub (num 3) (num 4)) (num 7))
  given: (add (sub (num 3) (num 4)) (num 7))

good (parse '(- 5 1 2)) at line 36
  expected: "parse: bad syntax: (- 5 1 2)"
  given: "parse: bad syntax: (- 5 1 2)"
```

Interpreter




```
; [contract] 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))]
    ))

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

```
good (interp (parse '3)) at line 46
  expected: 3
  given: 3

good (interp (parse '(+ 3 4))) at line 47
  expected: 7
  given: 7

good (interp (parse '(+ (- 3 4) 7))) at line 48
  expected: 6
  given: 6
```

Practice more!

Can you implement an AE parser for syntax based on infix or postfix?

- Infix: (2 + (9 - 2))
- Postfix: (2 (9 2 -) +)