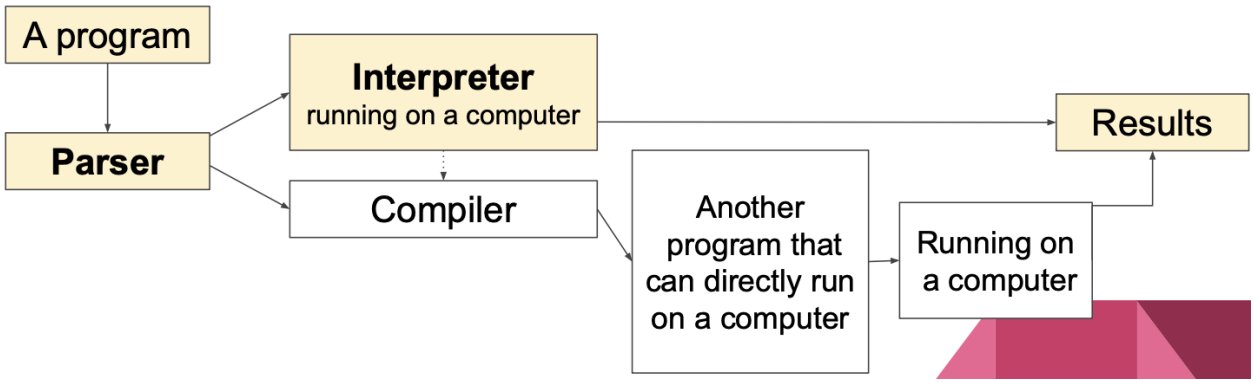
L4. Modeling Languages

유형	강의
복습 여부	☑
Status	Done

Interpreter vs Compiler

- Interpreter : steps through the source code line by line, figuring out what it's doing as it goes
- Compiler : figures out everything a program will do, turns it into "machine code", then saves that to be executed later



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

> Intepreter is a program, source code is data

Write the program with a programming language

- Syntax
- Semantics = some behaviors associated with each syntax
- Numerous useful libraries
- A collection of idioms that programmers of that language use

  idioms ≠ library

  idioms = programming patterns

Which one is most significant to learn PLT?

Semantics

그중에서도 Interpreter semantics for modeling languages

Programming languages already have semantics. By using the existing semantics, we can implement our own interpreters.

Syntax

Although semantics are the most significant, we need syntax.

Modeling syntax

- **Concrete Syntax** ('expression')
  - 3 + 4 (infix)
  - 3 4 + (postfix)
  - (+ 3 4) (parenthesized prefix)

→ Each of these notations is in use by at least one programming language to represent the same semantic, ‘the sum of 3 and 4’.

→ But dealing with different kinds of syntax is challenging.

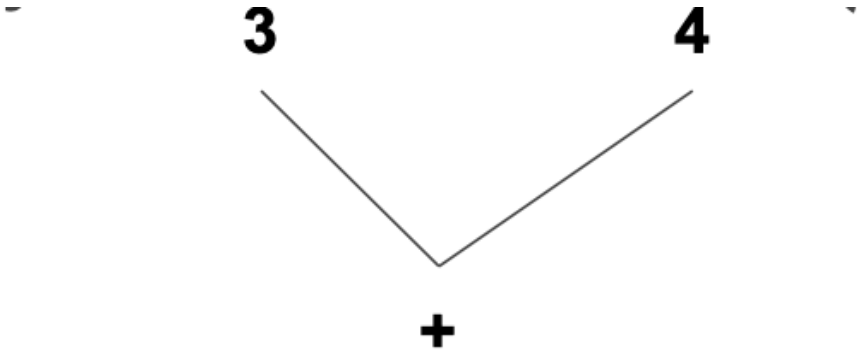
Can we have a general form of these syntax?

Why don’t we convert concrete syntax into one general abstract syntax?

If we can implement different different parsers to convert concrete syntax into one abstract syntax, we only need one interpreter to produce a result.

In addition, AST is helpful to analyze the expressions from the concrete syntax.

- **Abstract syntax** in a tree form



‘Representation’ with the ‘right data definition’ in Racket (add (num 3) (num 4))

Ex1. Model a new language, ‘AE’ that supports addition and subtraction

```
(define-type AE
  [num (n number?)]
  [add (lhs AE?) (rhs AE?)]
  [sub (lhs AE?) (rhs AE?)])
; 'AE' in (lhs AE?) can be either the instance of 'num', 'add', 'sub' variant

; (+ (- 2 3) 5) ==> 4
(define ae1 (add (sub (num 2) (num 3)) (num 5)))
(add? ae1)
(sub? ae1)
(add-lhs ae1)
```

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

Parser

a component in an interpreter or compiler

Identifies what kinds of program code 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.

Backus-Naur Form (BNF)

captures both the concrete syntax and a default abstract syntax!

⟨expr⟩ ::= (⟨expr⟩+⟨expr⟩)
| (⟨expr⟩ - ⟨expr⟩)
| ⟨num⟩
⟨num⟩ ::= 1, 42, 17, …

- <expr> ::= (element 1) | (element 2) | (element 3)

  Meta-variable, Non-terminal that can be replaced by one of the elements on the right-hand side

- <...>

  Literal syntax

- 1, 42, 17, …

  Actual values, Terminal that can not be replaced by others

⟨expr⟩ ::= (⟨expr⟩+⟨expr⟩)
| (⟨expr⟩ - ⟨expr⟩)
| ⟨num⟩

Each meta variable, such as ⟨expr⟩, defines a set

⟨num⟩ ::= 1, 42, 17, …

The set 〈num〉 is the set of all numbers.

2 ∈ 〈num〉
298 ∈ 〈num〉

To make an example 〈num〉, pick an element from it.

Example

〈expr〉 ::= (〈expr〉+〈expr〉) addition
| (〈expr〉 - 〈expr〉) subtraction
| 〈num〉 number

To make an example 〈expr〉:

1. Choose one case in the grammar
2. Pick an example for each meta variable
3. Combine the examples with literal text

예시 1

1. 〈num〉
2. 7 ∈ 〈num〉
3. 7 ∈ 〈num〉

예시 2

1. (〈expr〉 + 〈expr〉)
2. 8 ∈ 〈num〉 ⊆ 〈expr〉
3. (8 + 8) ∈ 〈expr〉

Example

⟨expr⟩ ::= ⟨expr⟩⟨expr⟩
| (⟨expr⟩ + ⟨expr⟩ = ⟨expr⟩)
| ⟨id⟩

⟨id⟩ ::= God, life, good, I, sorry, ␣(← space), great, HGU, ISEL, s, es, love, forgive, ?, PL, like, me, again,you, .(← dot/period), ,(← comma), Can, do, still, join, Do, is, am, C, D, J, L, N, P, S, U, W, Y

Generate any expressions by using the above syntax.

<expr>

(<expr> + <expr> = <expr>)

(<id> + <id> = <expr><expr>)

(Sorry + PL = <id><id>)

(Sorry + PL = forgiveme)

(Sorry + PL = <id><expr><expr>)

(Sorry + PL = forgive me)

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 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))
```