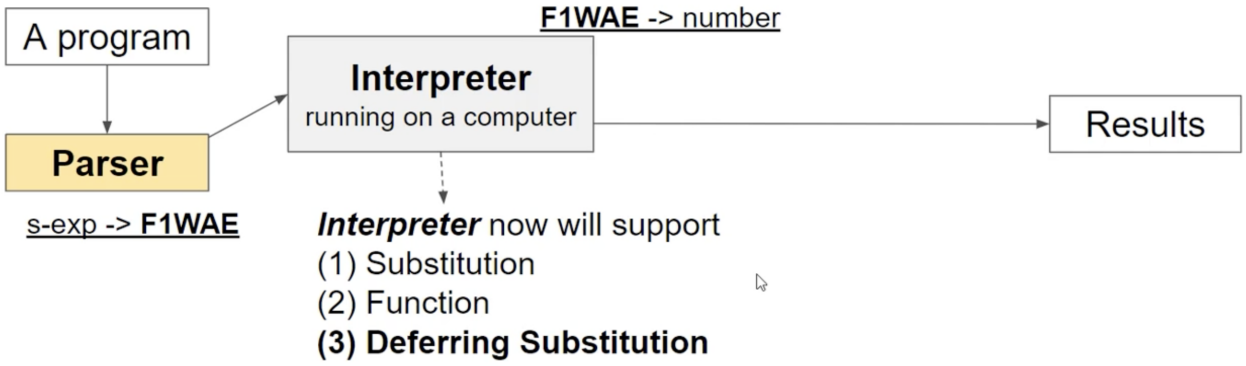
L9. Deferring Substitution

유형	강의
복습 여부	☑
Status	Done



Deferring Substitution

Let's improve the time complexity of the substitution we implemented in the previous lectures.

AE → WAE → **F1WAE with better substitution**

```
(interp (parse '{with {x 1}           [ ]
                      {+ {with {x 2} x}
                         x}}))
⇒
(interp (parse '{+ {with {x 2} x}     [x=1]
                   x}))
⇒
(+      (interp (parse '{with {x 2} x}     [x=1] ))
    (interp (parse 'x   [x=1])))
⇒
(+ (interp (parse 'x  [x=2 x=1])) (interp (parse 'x [x=1])))
⇒ (+ 2 1)
```

cf. cache is a data structure that looks like a linked list

WAE

```
; WAE
(define-type WAE
  [num (n number?)]
  [add (lhs WAE?) (rhs WAE?)]
  [sub (lhs WAE?) (rhs WAE?)]
  [with (name symbol?) (named-expr WAE?) (body WAE?)]
  [id (name symbol?)])
```

DefrdSub

```
; DefrdSub
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value number?) (saved DefrdSub?)])
; mtSub : mt stands for 'empty' cache (repository)
; aSub : non-empty cache, a pair of an identifier and a value for subsitution and the next pair
```

```
; example instance
(aSub 'x 1 (aSub 'y 4 (aSub 'x 2 (mtSub))))
```

WAE Parser

```
; [contract] parse : sexp -> WAE
; [purpose] to convert s-expression into WAE
(define (parse sexp)
  (match sexp
    [(? number?) (num sexp)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'with (list i v) e) (with i (parse v) (parse e))]
    [(? symbol?) (id sexp)]
    [else (error 'parse "bad syntax:~a" sexp)]))
```

```
(parse '{with {x 1} {+ {with {x 2} x} x}} )
(parse '{with {x 1} {with {y 2} {+ y x}}} )
```

```
(with 'x (num 1) (add (with 'x (num 2) (id 'x)) (id 'x)))
(with 'x (num 1) (with 'y (num 2) (add (id 'y) (id 'x))))
```

WAE Interpreter with DefrdSub

to maintain our cache, we need to provide an empty cache as a second parameter. While interpreting our source code, this cache will be updated.

```
(interp (parse '{with {x 1}
                      {with {y 2}
                         {+ 100 {+ 99 {+ 98 … {+ y x} … }}}})(mtSub))
⇒
(interp (parse '{with {y 2}
                      {+ 100 {+ 99 {+ 98 … {+ y x} … }}}})
        (aSub 'x 1 (mtSub)))
⇒
(interp (parse '{+ 100 {+ 99 {+ 98 … {+ y x} … }}})
        (aSub 'y 2 (aSub 'x 1 (mtSub))))
⇒ …
⇒
(interp (parse 'y) (aSub 'y 2 (aSub 'x 1 (mtSub))))
```

For deferred substitution, we need a helper function to lookup a value of the id, s from ds.

```
; [contract] interp : WAE DefrdSub -> number
(define (interp wae ds)
  (type-case WAE wae
    [num (n) n]
    [add (l r) (+ (interp l ds) (interp r ds))]
    [sub (l r) (- (interp l ds) (interp r ds))]
    [with (i v e) (interp e (aSub i (interp v ds) ds))]
    [id (s) (lookup s ds)]))

; [contract] lookup : symbol DefrdSub -> number
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free identifier")]
    [aSub (i v saved) (if (symbol=? i name)
                          v
                          (lookup name saved))]))
```

```
(test (lookup 'x (aSub 'x 1 (mtSub))) 1)
(test (lookup 'y (aSub 'x 1 (aSub 'y 4 (mtSub)))) 4)
(test (interp (with 'x (num 1) (add (with 'x (num 2) (id 'x)) (id 'x))) (mtSub)) 3)
(test (interp (with 'x (num 1) (with 'y (num 2) (add (id 'y) (id 'x)))) (mtSub)) 3)
```

```
good (lookup 'x (aSub 'x 1 (mtSub))) at line 53
  expected: 1
  given: 1

good (lookup 'y (aSub 'x 1 (aSub 'y 4 (mtSub)))) at line 54
  expected: 4
  given: 4

good (interp (with 'x (num 1) (add (with 'x (num 2) (id 'x)) (id 'x))) (mtSub)) at line 55
  expected: 3
  given: 3

good (interp (with 'x (num 1) (with 'y (num 2) (add (id 'y) (id 'x)))) (mtSub)) at line 56
  expected: 3
  given: 3
```

Scope

Static Scope

- the scope of an identifier’s binding is a syntactically delimited region.
- binding of a variable can be determined by program text and is independent of the run-time function call stack.
- the compiler first searches in the current block, then in global variables, then in successively smaller scopes.
- it’s common in most programming languages.

Ex1. C Programming Supports static scoping

```
// A C program to demonstrate static scoping.
#include<stdio.h>

int x = 10;

// Called by g()
int f()
{
return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
```

```
int x = 20;
return f();
}

int main()
{
printf("%d", g());
printf("\n");
return 0;
}
```

The output : 10

The value returned by f() is not dependent on who is calling it. f() always returns the value of global variable x.

Dynamic Scope

- the scope of an identifier’s binding is the entire remainder of the execution during which that binding is in effect.
- each identifier has a global stack of bindings and the occurrence of an identifier is searched in the most recent binding.
    - each time a new function is executed, a new scope is pushed onto the stack.
- the compiler first searches the current block and then successively all the calling functions.
- it’s very uncommon in the familiar programming languages.

Ex2. a pseudo code to demonstrate dynamic scoping

```
int x = 10

// Called by g()
int f()
{
   return x;
}

// g() has its own variable
// named as x and calls f()
int g()
{
   int x = 20;
```

```
   return f();
}

main()
{
  printf(g());
}
```

The output : 20

Ex3. **Perl** supports both static and dynamic scoping.

Perl's keyword "my" defines a statically scoped local variable, while the keyword "local" defines a dynamically scoped local variable.

```
# A perl code to demonstrate static scoping

$x = 50;

sub fun2
{
return $x;
}

sub fun1
{
# Since 'my' is used, x uses static scoping.
my $x = 10;
my $y = fun2();
return $y;
}

print fun1();
```

The output : 50

```
# A perl code to demonstrate dynamic scoping

$x = 50;

sub fun2
{
return $x;
}

sub fun1
```

```
{
# Since 'local' is used, x uses dynamic scoping.
local $x = 10; # temporarily updates global variable $x
my $y = fun2();
return $y;
}

print fun1();
```

The output : 10

Static Scoping vs Dynamic Scoping

	Static Scoping	Dynamic Scoping
Pros	easy to understand / faster execution / debugging less time-consuming because the scope of variables can be determined at compile-time	more flexible / easier to write code that is reusable because variables can be accessed from anywhere.
Cons	can be harder to write code that is reusable, because variables can only be accessed within their defined scope.	can lead to slower execution / harder to reason about the scope of variables in a program because it is determined at runtime.

F1WAE

```
(define-type FunDef
  [fundef (fun-name symbol?)
          (arg-name symbol?)
          (body F1WAE?)])

(define-type F1WAE
  [num (n number?)]
  [add (lhs F1WAE?) (rhs F1WAE?)]
  [sub (lhs F1WAE?) (rhs F1WAE?)]
  [with (name symbol?) (named-expr F1WAE?) (body F1WAE?)]
  [id (name symbol?)]
  [app (ftn symbol?) (arg F1WAE?)])
```

DefrdSub

```
; DefrdSub
(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?) (value number?) (saved DefrdSub?)])
; mtSub : mt stands for 'empty' cache (repository)
; aSub : non-empty cache, a pair of an identifier and a value for subsitution and the next pair
```

```
; example instance
(aSub 'x 1 (aSub 'y 4 (aSub 'x 2 (mtSub))))
```

F1WAE Parser

```
; [contract] parse-fd : sexp -> FunDef
(define (parse-fd sexp)
  (match sexp
    [(list 'deffun (list f x) b) (fundef f x (parse b))]))

; [contract] parse : sexp -> F1WAE
(define (parse sexp)
  (match sexp
    [(? number?) (num sexp)]
    [(list '+ l r) (add (parse l) (parse r))]
    [(list '- l r) (sub (parse l) (parse r))]
    [(list 'with (list i v) e) (with i (parse v) (parse e))]
    [(? symbol?) (id sexp)]
    [(list f a) (app f (parse a))]
    [else (error 'parse "bad syntax: ~a" sexp)]))
```

```
(parse '{f 1})
(parse-fd '{deffun {f x} {+ x 3}})
```

```
(app 'f (num 1))
(fundef 'f 'x (add (id 'x) (num 3)))
```

F1WAE Interpreter with Defrdsub

```
; [contract] interp : F1WAE list-of-FuncDef DefrdSub -> number
(define (interp f1wae fundefs ds)
```

```
  (type-case F1WAE f1wae
    [num (n) n]
    [add (l r) (+ (interp l fundefs ds) (interp r fundefs ds))]
    [sub (l r) (- (interp l fundefs ds) (interp r fundefs ds))]
    [with (i v e) (interp e fundefs (aSub i (interp v fundefs ds) ds))]
    [id (s) (lookup s ds)]
    [app (f a) (local
                 [(define a_fundef (lookup-fundef f fundefs))]
                 (interp (fundef-body a_fundef)
                         fundefs
                         (aSub (fundef-arg-name a_fundef)
                               (interp a fundefs ds)
                               (mtSub))))]))

; [contract] lookup : symbol DefrdSub -> number
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free identifier")]
    [aSub (i v saved) (if (symbol=? i name)
                          v
                          (lookup name saved))]))

; [contract] lookup-fundef : symbol list-of-FunDef -> FunDef
(define (lookup-fundef name fundefs)
  (cond
    [(empty? fundefs)
     (error 'lookup-fundef "unknown function")]
    [else
     (if (symbol=? name (fundef-fun-name (first fundefs)))
         (first fundefs)
         (lookup-fundef name (rest fundefs)))]))
```

```
; (parse '{f 1})
; (parse-fd '{deffun {f x} {+ x 3}})
(test (interp (app 'f (num 1)) (list (fundef 'f 'x (add (id 'x) (num 3)))) (mtSub)) 4)
```

```
good (interp (app 'f (num 1)) (list (fundef 'f 'x (add (id 'x) (num 3)))) (mtSub)) at line 76
  expected: 4
  given: 4
```

Why we used ‘mtSub’ instead of ‘ds’ in

```
(aSub (fundef-arg-name a_fundef) (interp a fundefs ds) (mtSub))
```

Each function definition creates new independent scope for it.