4th Week Group activity

상태	완료
강의	💻 Programming Language Theory

Topic 1 : Syntax error vs Semantic error

Discuss if a free identifier is a syntax error or semantic error?

	Syntax	Semantics
Meaning	The rules of any statement in the programming language	The meaning associated with any statement in the programming language
Error	It occurs when a statement is not valid according to the grammar of the programming language.	It occurs when a statement is syntactically valid but does not do what the programmer intended.
Examples	Ex. missing semicolons in C++, using undeclared variables in Java	Ex. when implementing division, the programmer forgot to consider the case when the divisor is 0

{with {x 5} {+ x {with {y x} x}}}

The syntax of 'with' is '{with {var expr} body}'. The above example does not violate the syntax of 'with'. But because the last 'x' is not bound in the surrounding context, '{with {y x} x} ', it's difficult to figure out where to look for the definition of 'x' and its value. Therefore, a free identifier is a semantic error.

Topic 2 : Syntactic sugar and desugaring

1. **Discuss what is syntactic sugar and desugaring.**

   Syntactic Sugar

   a syntax within a programming language designed to make things easier to read or to express, but does not add any new functionality to the language.

   ```
   # Using list comprehension (syntactic sugar)
   squared_numbers = [x**2 for x in range(1, 6)]
   print(squared_numbers)
   ```

   Desugaring

   the process of translating code written in syntactic sugar into an equivalent code that uses the more fundamental features of the language.

   - happens before interpreter
   - converting sugared expression into existing syntax

   ```
   # Equivalent code using a for loop (desugaring)
   squared_numbers = []
   for x in range(1, 6):
   ```

```
    squared_numbers.append(x**2)
print(squared_numbers)
```

2. **What is the advantage of using syntactic sugar?**

   It makes the language sweeter for human use.

   Things can be expressed more clearly, concisely and in an alternative style that some may prefer.

   Additionally, understanding how syntactic sugar is desugared helps us comprehend the underlying mechanisms.

3. **Provide five examples of syntactic sugar of your favorite languages.**

   Ex1. Compound Assignment Operators in Python

```
; x = x + 5
x += 5;
```

   Ex2. Ternary Operator in C

```
/*
bool passed;
if (mark >= 50) {
    passed = true;
} else {
    passed = false;
}
*/

bool passed = mark >= 50 ? true : false;
```

   Ex3. For Loops in C

```
/*
i = 0;
while (i < num_rows) {
  j = 0;
  while (j < num_cols) {
    matrix[i][j] = 1;
    j++;
  }
  i++;
}
*/

for (i = 0; i < num_rows; i++) {
  for (j = 0; j < num_cols; j++) {
    matrix[i][j] = 1;
  }
}
```

Advanced : Creating new syntactic sugar

Study textbook: http://cs.brown.edu/courses/cs173/2012/book/first-desugar.html

```
; Language 'ArithC' that supports number, addition and multiplication
(define-type ArithC
  [numC (n number?)]
  [plusC (l ArithC?) (r ArithC?)]
  [multC (l ArithC?) (r ArithC?)])
```

```
; Parser for 'ArithC'
; [contract] parse : sexp -> 'ArithC'
(define (parse sexp)
  (match sexp
    [(? number?) (numC sexp)]
    [(list '+ l r) (plusC (parse l) (parse r))]
    [(list '* l r) (multC (parse l) (parse r))]
    [else (error 'parse "bad syntax:~a" sexp)]))
```

```
; Interpreter for 'ArithC'
; [contract] interp : 'ArithC' -> number
(define (interp a)
  (type-case ArithC a
    [numC (n) n]
    [plusC (l r) (+ (interp l) (interp r))]
    [multC (l r) (* (interp l) (interp r))]))
```

```
(test (interp (parse '{+ 3 4})) 7)
(test (interp (parse '{* 5 4})) 20)
(test (interp (parse '{+ 6 {* 5 2}})) 16)
```

```
good (interp (parse '(+ 3 4))) at line 31
  expected: 7
  given: 7

good (interp (parse '(* 5 4))) at line 32
  expected: 20
  given: 20

good (interp (parse '(+ 6 (* 5 2)))) at line 33
  expected: 16
  given: 16
```

Let’s define a syntactic sugar for subtraction

```
; Language 'ArithS' that supports number, addition, subtraction, and multiplication
(define-type ArithS
  [numS (n number?)]
  [plusS (l ArithS?) (r ArithS?)]
  [minusS (l ArithS?) (r ArithS?)]
  [multS (l ArithS?) (r ArithS?)])
```

```
; Parser for 'ArithS'
; [contract] parse : sexp -> 'ArithS'
(define (parse sexp)
  (match sexp
    [(? number?) (numS sexp)]
    [(list '+ l r) (plusS (parse l) (parse r))]
    [(list '- l r) (minusS (parse l) (parse r))]
    [(list '* l r) (multS (parse l) (parse r))]
    [else (error 'parse "bad syntax:~a" sexp)]))
```

```
; Desugar function
; Desugaring allows us to simplify the surface language while still leveraging the capabilities of the core language.
; the subtraction operation is desugared into addition and multiplication by -1
; [contract] desugar : 'ArithS' -> 'ArithC'
(define (desugar as)
  (type-case ArithS as
    [numS (n) (numC n)]
    [plusS (l r) (plusC (desugar l) (desugar r))]
    [multS (l r) (multC (desugar l) (desugar r))]
    [minusS (l r) (plusC (desugar l) (multC (numC -1) (desugar r)))]))
```

```
(test (interp (desugar (parse '{- 30 10}))) 20)
(test (interp (desugar (parse '{+ 6 {- 5 2}}))) 9)
```

```
good (interp (desugar (parse '(- 30 10)))) at line 76
  expected: 20
  given: 20

good (interp (desugar (parse '(+ 6 (- 5 2))))) at line 77
  expected: 9
  given: 9
```