Semantics (= Evaluation) / PLAI Chapter 3

An important feature of BNF specifications: we can use the derivations to specify **meaning**

Given a grammar

1
```
ae  := NUMBER
       | ae1 + ae2
       | ae1 - ae2
ae1: ae
ae2: ae
```

We can define an **evaluator** function

2
```
a. eval(NUMBER) = NUMBER ; <-- special rule:
                           ; moves syntax into a
                             value
b. eval(ae1 + ae2) = eval(ae1) + eval(ae2)
c. eval(ae1 - ae2) = eval(ae1) - eval(ae2)
```

Semantics (= Evaluation) / PLAI Chapter 3

An important feature of BNF specifications: we can use the derivations to specify **meaning**

Given a grammar

```
ae := NUMBER
      | ae1 + ae2
      | ae1 - ae2
ae1: ae
ae2: ae
```

We can define an **evaluator** function

```
a. eval(NUMBER) = NUMBER ; <-- special rule:
                         ; moves syntax into a
                           value
b. eval(ae1 + ae2) = eval(ae1) + eval(ae2)
c. eval(ae1 - ae2) = eval(ae1) - eval(ae2)
```

1. meaning in our context is "running" a program (or "interpreting", "compiling", but we will use "evaluating")).
2. The text (and many others) use the convention of $\langle A \rangle$ to denote meta-variables, i.e. variables whose values are pieces of program.

- Note the completely different roles of the two ”+”s and ”-”s. In fact, it might have been more correct to write:

3

```
a. eval(NUMBER) = NUMBER
b. eval(ae1 "+" ae2) = eval(ae1) + eval(ae2)
c. eval(ae1 "-" ae2) = eval(ae1) - eval(ae2)
```

- When we write:
  eval(1 - 2 + 3) = ... = 1 - 2 + 3
  we have two expressions, but one stands for an input syntax, and one stands for a ‘real’ mathematical expression.

- Note the completely different roles of the two "+"s and "-"s. In fact, it might have been more correct to write:

```
a. eval(NUMBER) = NUMBER
b. eval(ae1 "+" ae2) = eval(ae1) + eval(ae2)
c. eval(ae1 "-" ae2) = eval(ae1) - eval(ae2)
```

- When we write:
  eval(1 - 2 + 3) = ... = 1 - 2 + 3
  we have two expressions, but one stands for an input syntax, and one stands for a 'real' mathematical expression.

1. - Note that there's a similar kind of informality in our **syntax** BNF specifications, where we assume that FOO (foo) refers to some terminal (non-terminal).
2. An alternative popular notation for eval(X) is [[X]]:
   a. [[NUMBER]] = NUMBER b. [[ae1 + ae2]] = [[ae1]] + [[ae2]] c. [[ae1 - ae2]] = [[ae1]] - [[ae2]]

Ambiguity

4 `eval(1 - 2 + 3) = ?`

Depending on the way the expression is parsed, we get either 2 or -4:

5
```
eval(1 - 2 + 3) = eval(1 - 2) + eval(3)
                                        ; [b]
                = eval(1) - eval(2) + eval(3)
                                        ; [c]
                = 1 - 2 + 3             ; [a,a,a]
                = 2
```

```
eval(1 - 2 + 3) = eval(1) - eval(2 + 3)
                                        ; [c]
                = eval(1) - (eval(2) + eval(3))
                                        ; [a]
                = 1 - (2 + 3)           ; [a,a,a]
                = -4
```

- We need parens around a sub-expression only in one case, why?
- With ‘eval’ the parse tree matters, so ambiguity must be eliminated.

```
eval(1 - 2 + 3) = eval(1) - eval(2 + 3)
                                        ; [c]
                = eval(1) - (eval(2) + eval(3))
                                        ; [a]
                = 1 - (2 + 3)           ; [a,a,a]
                = -4
```

- We need parens around a sub-expression only in one case, why?
- With 'eval' the parse tree matters, so ambiguity must be eliminated.

1. In a case of a computer implementation, the syntax on the left is (as always) an AE syntax, and the 'real' expression on the right is an expression in whatever language we use to implement our AE language.

Suppose we want to parse numbers ourselves

7

```
number: DIGIT number

eval(0) = 0
eval(1) = 1
eval(2) = 2
...
eval(9) = 9

eval(DIGIT) = DIGIT
eval(DIGIT number) = 10*eval(DIGIT) + eval(number)
```

- This grammar is unambiguous.

8
```
number: DIGIT | DIGIT number
```

- But the corresponding eval rule is wrong:

9
```
eval(123) = 10*eval(1) + eval(23)
          = 10*1 + 10*eval(2) + eval(3)
          = 10*1 + 10*2 + 3
          = 33
```

- Changing the order of the last rule works much better:

10
```
number: DIGIT | number DIGIT
```

- and then:

11
```
eval(number DIGIT) = 10*eval(number) +
    eval(DIGIT)
```

- What are potential problems (think implementation language versus implemented language).

- Changing the order of the last rule works much better:

```
10  number: DIGIT | number DIGIT
```

- and then:

```
11  eval(number DIGIT) = 10*eval(number) +
        eval(DIGIT)
```

- What are potential problems (think implementation language versus implemented language).

1. Think of an average language that does not give you bignums, making the above rules fail when the numbers are too big. In Racket, we happen to be using an integer representation for the syntax of integers, and both are unlimited. But what if we wanted to write a Racket compiler in C or a C compiler in Racket? What about a C compiler in C, where the compiler runs on a 64 bit machine, and the result needs to run on a 32 bit machine?

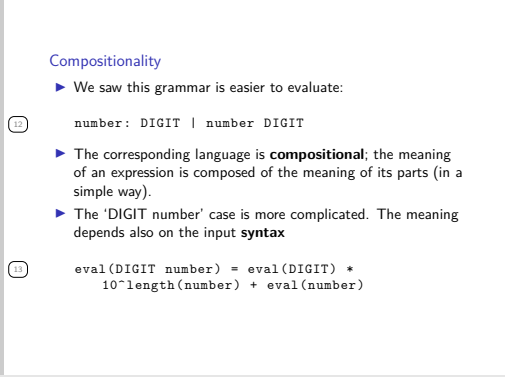
Compositionality

- We saw this grammar is easier to evaluate:

12
```
number: DIGIT | number DIGIT
```

- The corresponding language is **compositional**; the meaning of an expression is composed of the meaning of its parts (in a simple way).
- The 'DIGIT number' case is more complicated. The meaning depends also on the input **syntax**

13
```
eval(DIGIT number) = eval(DIGIT) *
    10^length(number) + eval(number)
```



1. This this case this can be tolerable, since the meaning of the expression is still made out of its parts – but imperative programming (when you use side effects) is much more problematic since it is not compositional (at least not in the obvious sense).
2. This is compared to functional programming, where the meaning of an expression is a combination of the meanings of its subexpressions. For example, every sub-expression in a functional program has some known meaning, and these all make up the meaning of the expression that contains them – but in an imperative program we can have a part of the code be 'x++' – and that doesn't have a meaning by itself, at least not one that contributes to the meaning of the whole program in a direct way.

Implementing an Evaluator

- For simple languages, a merged parser and evaluator is possible

14

```
(define (eval sx)
  (let ([rec (lambda (fn)
               (eval (fn (s-exp->list sx))))])
    (cond
      [(s-exp-match? `NUMBER sx)
       (s-exp->number sx)]
      [(s-exp-match? `(+ ANY ANY) sx)
       (+ (rec second) (rec third))]
      [(s-exp-match? `(- ANY ANY) sx)
       (- (rec second) (rec third))]
      [else (error 'eval (to-string sx))])))
```

Abstract and Concrete Syntax

- For more complex languages, evaluating in the parser is not usually practical.
- So we split into (at least) two layers
- By using the abstract syntax tree (AST) as an API, we can independently change the (concrete) syntax and the semantics.
- We can also e.g. provide better error checking/messages.


```
(eval `{+ 1 {- 3 "a"}})
```

Abstract and Concrete Syntax

- For more complex languages, evaluating in the parser is not usually practical.
- So we split into (at least) two layers
- By using the abstract syntax tree (AST) as an API, we can independently change the (concrete) syntax and the semantics.
- We can also e.g. provide better error checking/messages.

```
(eval `{+ 1 {- 3 "a"}})
```

1. We'll see that typechecking actually also works on the abstract syntax.
2. this is like the distinction between XML syntax and well-formed XML syntax.

Implementing The AE Language

- Back to our 'eval' – this will be its (obvious) type:

16
```
(AE -> Number)
;; consumes an AE and computes the
   corresponding number
```

- which leads to some obvious test cases:

17
```
(test (eval (parse-sx `3))                3)
(test (eval (parse-sx `{+ 3 4}))          7)
(test (eval (parse-sx `{+ {- 3 4} 7}))    6)
```

- We're less interested in testing the parser (except perhaps for debugging).

18
```
(test (parse-sx `{+ {- 3 4} 7})
      (Add (Sub (Num 3) (Num 4)) (Num 7)))
```

- The structure of the recursive 'eval' code follows the recursive structure of its input.

19
```
(define (eval expr)
  (type-case AE expr
    [(Num n)    ....]
    [(Add l r) (.... (eval l) .... (eval r))]
    [(Sub l r) (.... (eval l) .... (eval r))]))
```

- In this case, filling in the gaps is very simple

20
```
(define (eval expr)
  (type-case AE expr
    [(Num n)   n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]))
```

- We can further combine ‘eval’ and ‘parse’ into a single ‘run’ function that evaluates an AE s-expr.

21
```
(define (run sx)
  (eval (parse-sx sx)))
```

Putting the pieces together


```
#| BNF for the AE language:
   <AE> ::= <num>
          | { + <AE> <AE> }
          | { - <AE> <AE> }
          | { * <AE> <AE> }
          | { / <AE> <AE> }
|#
```

23

```
;; AE abstract syntax trees
(define-type AE
  [(Num val : Number)]
  [(Add l : AE) (r : AE)]
  [(Sub l : AE) (r : AE)]
  [(Mul l : AE) (r : AE)]
  [(Div l : AE) (r : AE)])
```

```
(define (parse-sx sx)
  (let ([rec (lambda (fn)
               (parse-sx (fn (s-exp->list sx))))])
    (cond
      [(s-exp-match? `NUMBER sx)
       (Num (s-exp->number sx))]
      [(s-exp-match? `(+ ANY ANY) sx)
       (Add (rec second) (rec third))]
      [(s-exp-match? `(- ANY ANY) sx)
       (Sub (rec second) (rec third))]
      [(s-exp-match? `(* ANY ANY) sx)
       (Mul (rec second) (rec third))]
      [(s-exp-match? `(/ ANY ANY) sx)
       (Div (rec second) (rec third))]
      [else (error 'parse-sx (to-string sx))])))
```

```
(define (eval expr)
  (type-case AE expr
    [(Num n)   n]
    [(Add l r) (+ (eval l) (eval r))]
    [(Sub l r) (- (eval l) (eval r))]
    [(Mul l r) (* (eval l) (eval r))]
    [(Div l r) (/ (eval l) (eval r))]))

(define (run sx)
  (eval (parse-sx sx)))

(test (run `3)  3)
(test (run `{+ 3 4})  7)
(test (run `{+ {- 3 4} 7})  6)
```