# CS4613 Lecture 2

David Bremner

March 21, 2023

# BNF, Grammars, the Simple AE Language

- We want to investigate programming languages, and we want to do that **using** a programming language.
- The first thing when we design a language is to specify the syntax.
- For this we use extended BNF (Backus-Naur Form). We'll use a version compatible with `http://docs.racket-lang.org/brag`

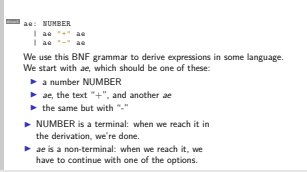
driver1

```
ae: NUMBER
  | ae "+" ae
  | ae "-" ae
```

We use this BNF grammar to derive expressions in some language. We start with *ae*, which should be one of these:

- a number NUMBER
- *ae*, the text "+", and another *ae*
- the same but with "-"

- NUMBER is a terminal: when we reach it in the derivation, we're done.
- *ae* is a non-terminal: when we reach it, we have to continue with one of the options.



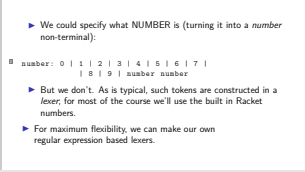

1. Explain the different parts. Specifically, this is a mixture of low-level (concrete) syntax definition with parsing.

- We could specify what NUMBER is (turning it into a *number* non-terminal):

```
number: 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
            | 8 | 9 | number number
```

- But we don't. As is typical, such tokens are constructed in a *lexer*; for most of the course we'll use the built in Racket numbers.

- For maximum flexibility, we can make our own regular expression based lexers.




1. This makes life a lot easier, and we get free stuff like floats, rationals etc.

For completeness, here is the lexer for *ae*

lexer

```
(define (tokenize ip)
  (define my-lexer
    (lexer-src-pos
     [(:+ numeric)
      (token 'NUMBER (string->number lexeme))]
     [(:or "+" "-" "*" "/" "(" ")" )
      (token lexeme lexeme)]
     [whitespace
      (token 'WHITESPACE lexeme #:skip? #t)]
     [(eof)
      (void)]))
  (define (next-token) (my-lexer ip))
  next-token)
```

# Using our parser:

driver1

```
(require "grammar1.rkt")
(require "lexer.rkt")

(syntax->datum (parse
                (tokenize-string "1 + 2 - 3")))
```

We can use our BNF to prove that “1-2+3” is a valid ae expression:

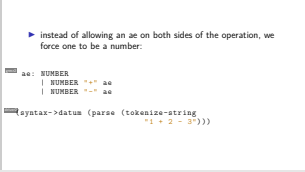
```
ae
ae + ae                  ; (2)
ae + NUMBER              ; (1)
ae - ae + NUMBER         ; (3)
ae - ae + 3              ; NUMBER
NUMBER - ae + 3          ; (1)
NUMBER - NUMBER + 3      ; (1)
1 - NUMBER + 3           ; NUMBER
1 - 2 + 3                ; NUMBER
```

- We can visualize the derivation using a tree, with the rules used at every node.
- These specifications suffer from being *ambiguous*: an expression can be derived in multiple ways (which means we don't know how to evaluate it).

- instead of allowing an ae on both sides of the operation, we force one to be a number:

grmr2
```
ae: NUMBER
    | NUMBER "+" ae
    | NUMBER "-" ae
```

driver2
```
(syntax->datum (parse (tokenize-string
                                "1 + 2 - 3")))
```




1. Now there is a single way to derive any expression, and it is always associating operations to the right: an expression like "1+2+3" can only be derived as "1+(2+3)".

- To change this to left-association, we would use this:

grm3
```
ae: NUMBER
    | ae "+" NUMBER
    | ae "-" NUMBER
```

driver3
```
(syntax->datum (parse (tokenize-string
                                "1 + 2 - 3")))
```

- Suppose that our AE syntax has addition and multiplication:

```
ae: NUMBER
  | ae "+" ae
  | ae "*"ae
```

- We can fix precedence by adding new non-terminals – say one for "factors":

```
ae: NUMBER
  | ae "+" ae
  | fac

fac: NUMBER
  | fac "*" fac
```

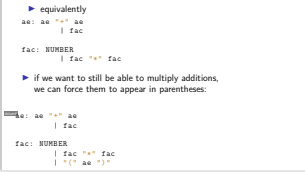
- equivalently

```
ae: ae "+" ae
        | fac

fac: NUMBER
        | fac "*" fac
```

- if we want to still be able to multiply additions, we can force them to appear in parentheses:

driver4

```
ae: ae "+" ae
        | fac

fac: NUMBER
        | fac "*" fac
        | "(" ae ")"
```




1. Now we must parse any AE expression as additions of multiplications (or numbers). First, note that if ae goes to ⟨fac⟩ and that goes to NUMBER, then there is no need for an ae to go to a NUMBER,

driver4

```
(parse-string "1 + 2 * 3")
(parse-string "1 * 2 + 3")
(parse-string "(1 + 2) * (3 + 4)")
```

- Next, note that AE is still ambiguous about additions, which can be fixed by forcing the left hand side of an addition to be a factor:

```
ae: fac "+" ae
      | fac

fac: NUMBER
      | fac "*" fac
      | "(" ae ")"
```

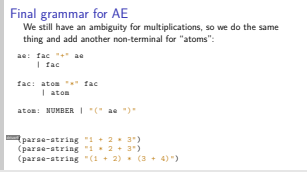
# Final grammar for AE

We still have an ambiguity for multiplications, so we do the same thing and add another non-terminal for “atoms”:

```
ae: fac "+" ae
  | fac

fac: atom "*" fac
   | atom

atom: NUMBER | "(" ae ")"
```

driver5

```
(parse-string "1 + 2 * 3")
(parse-string "1 * 2 + 3")
(parse-string "(1 + 2) * (3 + 4)")
```




1. And you can try to derive several expressions to be convinced that derivation is always deterministic now.
   But as you can see, this is exactly the cosmetics that we want to avoid – it will lead us to things that might be interesting, but unrelated to the principles behind programming languages. It will also become much much worse when we have a real language rather such a tiny one.

# A slightly more complex BNF example

grammar

```
json: number | string | array  | object

number: NUMBER
string: STRING
array: "[" [json ("," json)*] "]"
object: "{" [kvpair ("," kvpair)*] "}"
kvpair: STRING ":" json
```

The scanner is also a bit more complex, we have to parse quoted strings. I leave it for the curious to read...

lexer

driver

```
(syntax->datum
 (parse
  (tokenize (open-input-string #<<EOF
[
 {"thread": "0000000000031e50",
    "timestamp": 1358008026,
    "date_relative": "35 mins. ago",
    "matched": 1, "total": 1, "authors":
    "Debian Bug Tracking System",
    "subject": "Processed: tagging as
    pending bugs that are closed by
    packages in NEW", "tags": ["inbox",
    "inbox::debian", "unread"]},
 {"thread": "0000000000031e55",
    "timestamp": 1358006945,
```

# That's enough of that...

- We will declare the whole business of parsing **complicated** syntax to be tangential to this course.
- Is there a good solution? We can do what Racket does – always use fully parenthesized expressions:

```
ae: NUMBER
      | ( ae + ae )
      | ( ae - ae )
```

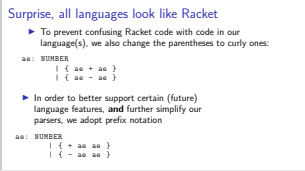
# Surprise, all languages look like Racket

- To prevent confusing Racket code with code in our language(s), we also change the parentheses to curly ones:

```
ae: NUMBER
  | { ae + ae }
  | { ae - ae }
```

- In order to better support certain (future) language features, **and** further simplify our parsers, we adopt prefix notation

```
ae: NUMBER
  | { + ae ae }
  | { - ae ae }
```




1. (Remember that in a sense, Racket code is written in a form of already-parsed syntax...)

# Concrete and Abstract syntax

```
3+4       (infix),
3 4 +     (postfix),
+(3,4)    (prefix with args in parens),
(+ 3 4)   (parenthesized prefix),
```

We can represent the tree as

```
(Add (Num 3) (Num 4))
```

Similarly, the expression

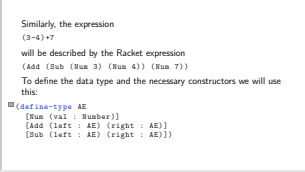
```
(3-4)+7
```

will be described by the Racket expression

```
(Add (Sub (Num 3) (Num 4)) (Num 7))
```

To define the data type and the necessary constructors we will use this:

29
```
(define-type AE
  [Num (val : Number)]
  [Add (left : AE) (right : AE)]
  [Sub (left : AE) (right : AE)])
```


Similarly, the expression

```
(3-4)+7
```

will be described by the Racket expression

```
(Add (Sub (Num 3) (Num 4)) (Num 7))
```

To define the data type and the necessary constructors we will use this:

```
(define-type AE
  [Num (val : Number)]
  [Add (left : AE) (right : AE)]
  [Sub (left : AE) (right : AE)])
```

1. Important note: "expression" was used in two **different** ways in the above – each way corresponds to a different language.
2. Racket follows the tradition of Lisp which makes syntax issues almost negligible – the language we use is almost as if we are using the parse tree directly. Actually, it is a very simple syntax for parse trees, one that makes parsing extremely easy.
3. This has an interesting historical reason... Some Lisp history – M-expressions vs. S-expressions, and the fact that we write code that is isomorphic to an AST. Later we will see some of the advantages that we get by doing this. See also "The Evolution of Lisp", section 3.5.1 (especially the last sentence).

# Two level parsing

We can replace the usual tokenizing with the (more powerful) `read`

30
```
(read)
;; "{+ 1 2}"
```

Then we write our own ‘parse’ function that will parse the resulting list into an instance of the AE type – an abstract syntax tree (AST).

# Recursive parser

```
(define (parse-sexpr sxp)
  (cond
    [(s-exp-number? sxp)
     (Num (s-exp->number sxp))]
    [(and (s-exp-list? sxp)
          (= 3 (length(s-exp->list sxp))))
     (let* ([lst (s-exp->list sxp)]
            [op (s-exp->symbol (first lst))]
            [l (parse-sexpr (second lst))]
            [r (parse-sexpr (third lst))])
       (case op
         [(+) (Add l r)]
         [(-) (Sub l r)]
         [else
          (error 'parse (to-string op))]))]
    [else (error 'parse (to-string sxp))]))
```

We can simplify the parser and make it more extensible by using `s-exp-match?`

32

```
(define (parse-sx sx)
  (local
      [(define (rec 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))]
      [else (error 'parse-sx (to-string sx))])))
```

We can combine our parser with the function that parses a string into a sexpr

33
```
;; parses a string containing an AE expression to an AE
(define (read-ae)
  (parse-sx (read)))
```

For our tests it will be more convenient to enter s-expressions directly, since we get indentation etc... for free.