Our goal here: Implementing a parser in pl for simple arithmetic expression language defined by the following grammar:

```
  <AE> ::= <num>
         | { + <AE> <AE> }
         | { - <AE> <AE> }
```

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

- **Simple Parsing -- Implementing a "parser"**

Regardless of what the **syntax actually looks like**, we want to parse it as soon as possible -- converting the **concrete syntax** to an **abstract syntax** tree.

No matter how we write our syntax:
- 3+4   (infix),
- 3 4 +   (postfix),
- +(3,4) (prefix with operands in parens),
- (+ 3 4) (parenthesized prefix),

In all of the above, we have the same **abstract syntax** – an operator "+" with two operands "3" and "4" (semantically, we mean the same thing -- adding the number 3 and the number 4). The essence of this (i.e., the **abstract syntax**) is basically a **tree structure** with an addition operation as the root and two leaves holding the two numerals.

With the right data definition, we can describe this in **Racket** -- as the expression (Add (Num 3) (Num 4)) where `Add' and `Num' are **constructors** of a tree type for syntax, or in a **C-like** language, it could be something like Add(Num(3),Num(4)).

Similarly, the expression

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

will be described in Racket as the expression:

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

Important note: "expression" was used in two *different* ways in the above -- each way corresponds to a different language.

Defining a new data type for the task:

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

```
  (define-type AE
     [Num Number]
     [Add AE AE]
     [Sub AE AE])
```

- Note -- **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.

[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).]

To make things at a very simple level, we will use the above fact through a double-level approach:

- **Read into a list form** – we first "parse" our language into an intermediate representation – a Racket list -- this is mostly done by a modified version of Racket's `read' that uses curly braces "{}"s instead of round parens "()"s.
  [This part will be given to us for free – we will be using a function that reads a string and outputs an Sexpr (a Racket number or list of elements)]
- **Parse into an AST data type** – 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)**.

The latter is achieved by the following simple recursive function:

```
 (: parse-sexpr : Sexpr -> AE)
  ;; to convert s-expressions into AEs
  (define (parse-sexpr sexpr)
    (cond [(number? sexpr) (Num sexpr)]
          [(and (list? sexpr) (= 3 (length sexpr)))
           (let ([make-node
                  (cond [(equal? '+ (first sexpr)) Add]
                        [(equal? '- (first sexpr)) Sub]
                        [else (error 'parse-sexpr "don't
know about ~s"
                                     (first sexpr))])])
             (make-node (parse-sexpr (second sexpr))
                        (parse-sexpr (third sexpr))))]
          [else (error 'parse-sexpr "bad syntax in ~s"
sexpr)]))
```

Equivalently, we can write (using the special form `` `match' ``):

```
  (: parse-sexpr : Sexpr -> AE)
  ;; to convert s-expressions into AEs
  (define (parse-sexpr sexpr)
    (cond [(number? sexpr) (Num sexpr)]
          [(and (list? sexpr) (= 3 (length sexpr)))
           (let ([make-node
                  (match (first sexpr)
                    ['+ Add]
                    ['- Sub]
                    [else (error 'parse-sexpr "don't know
about ~s"
                                 (first sexpr))])
                  #| the above is the same as:
                  (cond [(equal? '+ (first sexpr)) Add]
                        [(equal? '- (first sexpr)) Sub]
                        [else (error 'parse-sexpr "don't
know about ~s"
                                     (first sexpr))])
                  |#])
             (make-node (parse-sexpr (second sexpr))
                        (parse-sexpr (third sexpr))))]
          [else (error 'parse-sexpr "bad syntax in ~s"
sexpr)]))
```

This function is pretty simple, but as our languages grow, they will become more verbose and more difficult to write. So, instead, we use `match' in a more profound way - as what it does is to match a value and bind new identifiers to different parts (try it with "Check Syntax"). Re-writing the above code using `match':

```
  (: parse-sexpr : Sexpr -> AE)
  ;; to convert s-expressions into AEs
  (define (parse-sexpr sexpr)
    (match sexpr
      [(number: n) (Num n)]
      [(list '+ left right)
       (Add (parse-sexpr left) (parse-sexpr right))]
      [(list '- left right)
       (Sub (parse-sexpr left) (parse-sexpr right))]
      [else (error 'parse-sexpr "bad syntax in ~s"
sexpr)]))
```

To make things less confusing, we will combine this with the function that parses a string into a sexpr so we can use strings to represent our programs:

```
  (: parse : String -> AE)
  ;; parses a string containing an AE expression to an AE
  (define (parse str)
    (parse-sexpr (string->sexpr str)))
```

- **The `match' Form**

The syntax for `match' is

```
  (match value
    [pattern result-expr]
    ...)
```

The value is matched against each pattern, possibly binding names in the process, and if a pattern matches it evaluates the result expression.

The simplest form of a pattern is simply an identifier -- it always matches and binds that identifier to the value:

```
  (match (list 1 2 3)
    [x x]) ; evaluates to the list
```

Another simple pattern is **a quoted symbol**, which matches that symbol.
For example:

```
  (match var
     ['x "yes"]
     [else "no"])
```

will evaluate to "yes" if \`var' is the symbol \`x', and to "no" otherwise. Note that \`else' is not a keyword here -- it happens to be a pattern that always succeeds, so it behaves like an else clause except that it binds \`else' to the unmatched-so-far value.

Many patterns **look like** function application -- but don't confuse them with applications. A \`(list x y z)' pattern matches a list of exactly three items and binds the three identifiers; or if the "arguments" are themselves patterns, \`match' will descend into the values and match them too. This means that the patterns can be nested:

```
  (match (list 1 2 3)
    [(list x y z) (+ x y z)]) ; evaluates to 6
  (match '((1) (2) 3)
    [(list (list x) (list y) z) (+ x y z)]) ; evaluates to
6
```

There is also a \`cons' pattern that matches a non-empty list and then matches the first part against the head for the list and the second part against the tail of the list.

In a \`list' pattern, you can use \`...' to specify that the previous pattern is repeated zero or more times, and bound names get bound to the list of respective matching. One simple consequent is that the \`(list hd tl ...)' pattern is exactly the same as \`(cons hd tl)', but being able to repeat an arbitrary pattern is very useful:

```
  (match '((1 2) (3 4) (5 6) (7 8))
    [(list (list x y) ...) (append x y)])
  ; evaluates to (1 3 5 7 2 4 6 8)
```

A few more useful patterns:

```
  id                 -- matches anything, binds `id' to it
  _                  -- matches anything, but does not bind
  (number: n)        -- matches any number and binds it to `n'
  (symbol: s)        -- same for symbols
  (string: s)        -- strings
  (sexpr: s)         -- S-expressions (needed sometimes for
Typed Racket)
  (and pat1 pat2) -- matches both patterns
  (or pat1 pat2)  -- matches either pattern (careful with
bindings)
```

The patterns are tried one by one *in-order*, and if no pattern matches the value, an error is raised.

Note that `...' in a `list' pattern can follow *any* pattern, including all of the above, and including nested list patterns.

Here are a few examples -- you can try them out with "**#lang pl untyped**" (for our purposes pl –typed- will suffice) at the top of the definitions window. This:

```
  (match x
    [(list (symbol: syms) ...) syms])
```

matches `x' against a pattern that accepts only a list of symbols, and binds `syms' to those symbols. And here's an example that matches a list of any number of lists, where each of the sub-lists begins with a symbol and then has any number of numbers. Note how the `n' and `s' bindings get values for a list of all symbols and a list of lists of the numbers:

```
  > (define (foo x)
      (match x
        [(list (list (symbol: s) (number: n) ...) ...)
         (list 'symbols: s 'numbers: n)]))
  > (foo (list (list 'x 1 2 3) (list 'y 4 5)))
  '(symbols: (x y) numbers: ((1 2 3) (4 5)))
```

Here is a quick example for how `or' is used with two literal alternatives, how `and' is used to name a specific piece of data, and how `or' is used with a binding:

```
> (define (foo x)
    (match x
      [(list (or 1 2 3)) 'single]
      [(list (and x (list 1 _)) 2) x]
      [(or (list 1 x) (list 2 x)) x]))
> (foo (list 3))
'single
> (foo (list (list 1 99) 2))
'(1 99)
> (foo (list 1 10))
10
> (foo (list 2 10))
10
```