Let us recall the grammar Prakash presented in a previous class.

\[
\begin{align*}
\text{Exp} & = \text{Term} + \text{Exp} \mid \text{Term} \\
\text{Term} & = \text{Primary} \ast \text{Term} \mid \text{Primary} \\
\text{Primary} & = \text{Constant} \mid (\text{Exp}) \\
\text{Constant} & = \text{A} \mid \text{B} \mid \ldots \mid \text{Z}
\end{align*}
\]

Unlike Prakash’s notes, I grouped the constants together in a separate category. This is not of great importance but is simply done to emphasize certain aspects of a continuation passing style (CPS) parser.

The key aspect of CPS is the use of functions as control flow. As you saw in previous lectures, they serve to capture what is to be done next. If an instance of a recursive call needs to do several different things, it would provide a new function as continuation and thus take control to where to go next. In the case of parsers, two things can happen at any given step. The rule may succeed and so we move forward the token stream. The rule may also fail and thus we need to backtrack and try another one. This leads us to having two different continuations: a success continuation and a failure continuation.

What happens exactly when a rule succeeds? It consumes tokens from the token stream, modifying it. It also returns an expression which is the parsed tree. In order to make more sense of those, let us make the token stream and the parsed tree more precise.

The token stream is the output of the tokenizer (or lexer). A stream can be viewed as an infinite list (you could define the output of the tokenizer as a regular list if you wanted). You don’t need to know the details of how to define streams. For this lecture, we only assume that given a stream of tokens, we have two functions `next : tokens \rightarrow tok` and `rest : tokens \rightarrow tokens` which give respectively the head and the tail of the stream, respectively. Since, the input passed to the tokenizer is most likely finite, if we are passed the end of the input, we assume `next` returns always the end of input symbol: `EOI`. The individual tokens are the following

\[
\begin{align*}
type \text{cnst} & = \text{A} \mid \text{B} \mid \ldots \mid \text{Z} \\
type \text{tok} & = \text{Plus} \mid \text{Times} \mid \text{LParen} \mid \text{RParen} \mid \text{Ctok of cnst} \mid \text{EOI}
\end{align*}
\]

which represent respectively the characters `+`, `*`, `(`, `)`, A, B, ..., Z, and `\0`. We could also have used characters explicitly instead of tokens, if we wanted.
Now, the expression tree is defined as a recursive data type.

```plaintext
type exp = Sum of exp * exp | Prod of exp * exp | Cnst of cnst
```

The end result is thus a tree whose leaves are constants and inner node sums or products. The expression tree is simpler than the grammar because we do not need anymore information about parentheses or precedence since the result is simply a tree.

Coming back to our success continuation, we see that as the caller of the continuation will have a different token stream, and will have computed an expression. Hence, it needs to provide both of them as inputs to the success continuation. Its type is thus `exp → tokens → 'a`. We recall that the return type of continuations need to be polymorphic as it must only depend on the initial continuation.

For the failure continuation, it represents backtracking. It doesn’t have to return any computed result and wants us to forget its usage of the token stream (so the next rule starts at the right place). It will thus simply take unit `()` as input, and return again an `'a`.

We will have a function for each category in our grammar, each of them has type `tokens → (exp → tokens → 'a) → (unit → 'a) → 'a`

We will start by writing the function for primaries. We will build the function interactively and denote missing code to be filled out by `?`.

```plaintext
let rec parsePrim tokens succ fail = ?
```

Looking back at our grammar, we see that the first rule that is applied is the constant rule. We will thus call the function `parseConstant` which simply pattern matches on the next token. If `parseConstant` succeeds, we do not need to do anything afterward in our rule for primary. Hence, we will simply use the success continuation we were given.

```plaintext
let rec parsePrim tokens succ fail = parseConstant tokens succ ?
```

If `parseConstant` fails, we want to try our other rule. This rule tests whether the next token is an opening parenthesis and then calls `parseExp`. If either of those steps fail, we do not have any more rule to use and thus we simply call the given failure continuation. Our new failure continuation is thus

```plaintext
fun () -> if next tokens = LParen then
      parseExp (rest tokens) ? fail
    else fail ()
```

Now, what happens if `parseExp` succeeds? We need to verify that there is a closing parenthesis. We thus need a success continuation for it. We recall that a success continuation takes an expression `e` and a token stream `tokens'`. If we succeed, we simply are done with this rule. We can simply return the expression we were given (as we do not care about the parentheses anymore) and we simply strip out the token stream from the closing parenthesis we just read. If we fail, we do not have any other rule we can try. Hence, the continuation is
fun e tokens’ -> if next tokens = RParen then
    succ e (rest tokens’)
  else fail ()

Putting all of them together gives us the following resulting function.

let rec parsePrim tokens succ fail =
  let succ’ = fun e tokens’ ->
    if next tokens = RParen then
      succ e (rest tokens’)
    else fail ()
  let fail’ = fun () ->
    if next tokens = LParen then
      parseExp (rest tokens) succ’ fail
    else fail ()
  parseConstant tokens succ fail’

Now let us write parseTerm. First, we call parsePrim.

and parseTerm tokens succ fail = parsePrim tokens ? ?

At any point, if the rule fails, we try the next one which simply calls parsePrim. This is the failure continuation.

let fail’ = fun () -> parseTerm tokens succ fail

If it succeeds, we test whether the next token is Times. If it is, we do a recursive call. Otherwise, we fail.

let succ’ = fun e tokens’ ->
  if next tokens’ = Times then
    parseTerm (rest tokens’) ? fail’
  else fail’ ()

Now, the recursive call will return an expression e’, but this is not the expression we want to return. What we want to return is Prod (e, e’). Hence we write a new success continuation to build the expression.

let succ’ =
  fun e tokens’ ->
    let succ’’ = (fun e’ toks -> succ (Prod (e, e’)) toks)
    if next tokens’ = Times then
      parseTerm (rest tokens’) succ’’ fail’
    else fail’ ()

The resulting function is thus

and parseTerm tokens succ fail =
  let fail’ = fun () -> parsePrim tokens succ fail
  let succ’ =
fun e tokens' ->
  let succ'' = fun e' toks -> succ (Prod (e, e')) toks
  if next tokens' = Times then
      parseTerm (rest tokens') succ'' fail'
  else fail' ()
  parsePrim tokens succ' fail'

Since expressions have the same structure than terms, parseExp is basically the same as parseTerm.

and parseExp tokens succ fail =
  let fail' = fun () -> parseTerm tokens succ fail
  let succ' =
      fun e tokens' ->
        let succ'' = fun e' toks -> succ (Sum (e, e')) toks
        if next tokens' = Plus then
            parseExp (rest tokens') succ'' fail'
        else fail' ()
        parseTerm tokens succ' fail'

  This concludes the definition of our program. Now, as initial continuations, one could use something like the following.

let parse tokens =
  let succ = fun e tokens' ->
      if next tokens' = EOI then e
      else failwith "Input not all consumed."
  let fail = fun () -> failwith "Incorrect input"
  parseExp tokens succ fail