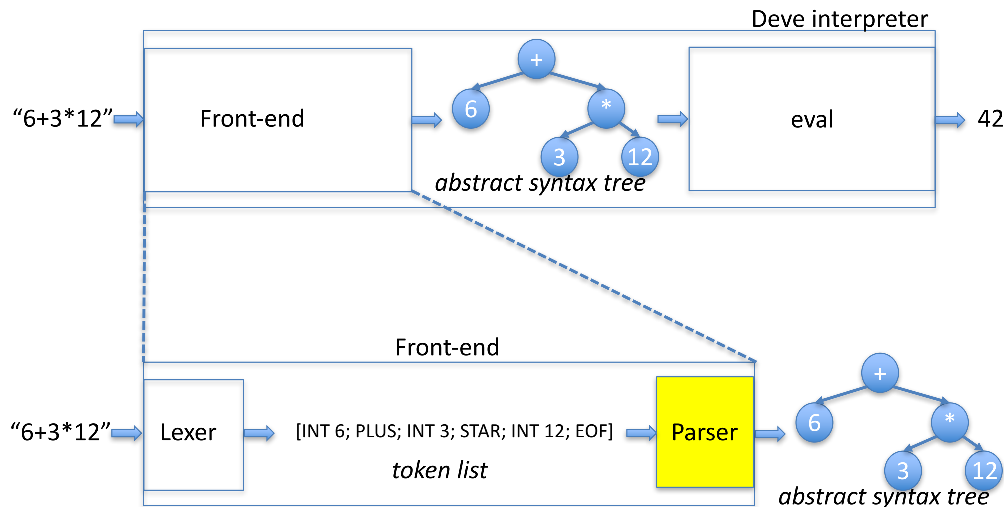
Parsing

Barış Aktemur
CS321 Programming Languages
Ozyegin University

Parser is the part of the interpreter/compiler that takes the tokenized input from the lexer (aka scanner) and builds an abstract syntax tree out of it. See the big picture below.



In order to speak about parsing, we first need to talk about "grammar". A grammar is the set of rules that define the **structure** of valid inputs in a language. Just like there are grammar rules for natural languages such as Turkish, English, Korean, etc., there are grammar rules for programming languages as well. The grammar of a language specifies only what the correct structure of an input is; it does not say anything about the "meaning" of the input. That is, the grammar is about the **syntactic** properties, it is not about **semantical** properties. For instance, the following sentence is grammar-wise just fine, although it has semantic problems.

The weather tomorrow was a green bazinga.

We define the grammar rules of a language using the *Backus-Naur Form* (BNF) notation. Here is the grammar of a simple language in BNF:

```
(A)  main ::= exp EOF
(B)  exp  ::= INT
(C)          | NAME
(D)          | exp PLUS exp
(E)          | exp STAR exp
(F)          | LET NAME EQ exp IN exp
(G)          | IF exp THEN exp ELSE exp
```

This notation as shown here may look like OCaml code to you. But it is not. Do NOT confuse it with executable code. It is just some notation for specifying grammar. Here, there are **terminal** symbols, written in UPPERCASE letters; and **non-terminals** written in lowercase letters. Terminals correspond to the tokens received from the lexer. The grammar has 7 rules, marked as (A), (B), etc. These rules are called **productions**. Finally, a grammar should have a **starting symbol**. In the grammar above, the starting symbol is *main*.

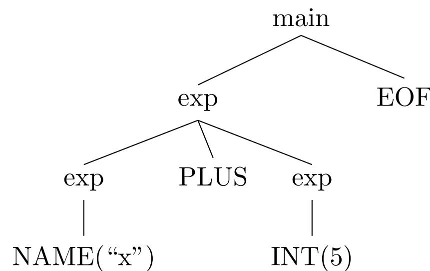
The rules tell us the following:

- A *main* is defined as an *exp* followed by the EOF token. (rule A)
- An *exp* is defined as any of the following:
  - the INT token, (rule B)
  - the NAME token, (rule C)
  - an *exp* followed by the PLUS token, followed by an *exp*, (rule D)
  - ...
  - the IF token followed by an *exp*, followed by the THEN token, followed by an *exp*, followed by the ELSE token, followed by an *exp*. (rule G)

Using these rules, we **derive** inputs. Derivation is carried out as follows:

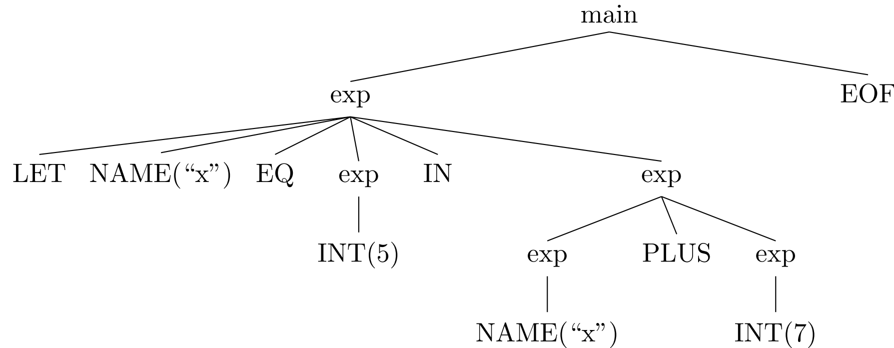
- Start with the starting symbol.
- At each step, pick a non-terminal, and expand it by applying one of the productions.
- Stop when no more non-terminals remain (i.e. all we have are terminals).
- The result is a grammatically correct input according to the specified grammar.

We can show the expansions (i.e. applications of the production rules) as a tree. Here is a simple example:



This is called *a parse tree*, or sometimes *a derivation tree*, or sometimes *a concrete syntax tree (CST)*. If you can derive a parse tree for a list of tokens, this means that token list is a valid input of the language. So, NAME("x") PLUS INT(5) EOF (i.e. "x+5") is a valid input.

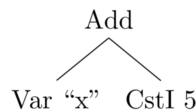
Below is just another example showing the parse tree for LET NAME EQ INT IN NAME PLUS INT EOF (i.e. "let x=5 in x+7").



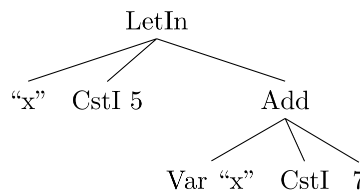
Parsing is the activity of answering the following question: Is there a parse tree for a given input? However, we are not only interested in this yes/no question, we also would like to know what that tree is, so that we can achieve our ultimate goal of building an *abstract syntax tree (AST)*. “An abstract syntax tree? What the hell is that? What's the difference between an abstract and concrete syntax tree?” I hear you asking.

A concrete syntax tree is the derivation tree for an input that contains all the tokens in the input. It contains all the details. We do not need to keep unnecessary details, however. They just create clutter. We can throw away those unneeded details and represent the input using a leaner tree. That's the abstract syntax tree; it is abstraction of the concrete tree to contain just enough information to be able to represent the input program.

Below you see the AST for the input “x+5”. The AST is built using our definition of the `exp` data type.



Here is the AST for the input “let x=5 in x+7”. Compare this to the CST given previously for the same input.



Implementing the Parser

Let us now start implementing a parser for the Deve language. We will assume a simple grammar at the beginning. We will gradually add more rules. Here is the initial version:

```
main ::= exp EOF
exp  ::= INT
       | NAME
```

So, a valid input in this grammar is just an integer literal or a name, followed by the EOF marker. Here is the parser for this grammar:

```
(* parseExp: token list -> (exp, token list)
   Parses an exp out of the given token list,
   returns that exp together with the unconsumed tokens.
 *)
let rec parseExp tokens =
  match tokens with
  | INT i :: rest -> (CstI i, rest)
  | NAME x :: rest -> (Var x, rest)
  | _ -> failwith "Unexpected token."

(* parseMain: token list -> exp *)
let parseMain tokens =
  match parseExp tokens with
  | (e, [EOF]) -> e
  | _ -> failwith "I was expecting to see an EOF."

(* parse: string -> exp *)
let rec parse s =
  parseMain (scan s)
```

In this implementation, the `parseExp` function takes a token list, builds an `exp` out of the tokens it reads from that list, and returns that `exp` together with the tokens that it did not consume. For example:

```
# parseExp (scan "321 + x");;
- : exp * token list = (CstI 321, [PLUS; NAME "x"; EOF])
# parseExp (scan "xyz abc k");;
- : exp * token list = (Var "xyz", [NAME "abc"; NAME "k"; EOF])
```

The `parseMain` function uses the `parseExp` function build an `exp`, and expects the remaining token list to contain just the `EOF` token. If not, the function fails. For convenience, we can use the `parse` function to scan the input string and build the AST:

```
# parse "321";;
- : exp = CstI 321
# parse "abc";;
- : exp = Var "abc"
# parse "abc 321";;
Exception: Failure "I was expecting to see an EOF.".
# parse "abc +";;
Exception: Failure "I was expecting to see an EOF.".
```

Let us now add a simple rule to the grammar to also contain boolean literals:

```
  main ::= exp EOF
  exp  ::= INT
         | NAME
         | BOOL
```

Of course, we assume here that the lexer is capable of recognizing “true” and “false” as the BOOL token. We update the definition of the `parseExp` function to be able to parse booleans as below. Other parts stay the same.

```
let rec parseExp tokens =
  match tokens with
  ...
  | BOOL b :: rest -> (CstB b, rest)
```

Let's test this.

```
# parse "true";;
- : exp = CstB true
# parse "false";;
- : exp = CstB false
```

Good. Let us add two more rules to the grammar:

```
  main ::= exp EOF
  exp  ::= INT
         | NAME
         | BOOL
         | LET NAME EQ exp IN exp
         | IF exp THEN exp ELSE exp
```

We update the definition of the `parseExp` function as follows:

```
let rec parseExp tokens =
  match tokens with
  ...
  | LET::(NAME x)::EQUALS::rest ->
     (match parseExp rest with
      | (e1, IN::rest1) -> let (e2, rest2) = parseExp rest1
                           in (LetIn(x, e1, e2), rest2)
     )
  | IF::rest ->
     (match parseExp rest with
      | (e1, THEN::rest1) ->
         (match parseExp rest1 with
          | (e2, ELSE::rest2) -> let (e3, rest3) = parseExp rest2
                                 in (If(e1, e2, e3), rest3)
         )
     )
```

Here, we call the `parseExp` function recursively to build the subexpressions of the if-then-else and let-in expressions. Let's test this:

```
# parse "let x = 5 in y";;
- : exp = LetIn ("x", CstI 5, Var "y")
# parse "let x = 5 in let z = 42 in sum";;
- : exp = LetIn ("x", CstI 5, LetIn ("z", CstI 42, Var "sum"))
# parse "let x = let w = g in 42 in y";;
- : exp = LetIn ("x", LetIn ("w", Var "g", CstI 42), Var "y")
# parse "if true then 4 else 5";;
```

```
- : exp = If (CstB true, CstI 4, CstI 5)
# parse "if true then 4 else if c then a else b";;
- : exp = If (CstB true, CstI 4, If (Var "c", Var "a", Var "b"))
# parse "if true then if k then a else b else 5";;
- : exp = If (CstB true, If (Var "k", Var "a", Var "b"), CstI 5)
```

Excellent! I'm proud of the progress we're making. Note that the match expressions in the code above are not exhaustive. If the input is bad, a match failure may occur. To fail gracefully with slightly more useful error messages, we can update the code as follows:

```
let rec parseExp tokens =
  match tokens with
  ...
  | LET::(NAME x)::EQUALS::rest ->
     (match parseExp rest with
      | (e1, IN::rest1) -> let (e2, rest2) = parseExp rest1
                           in (LetIn(x, e1, e2), rest2)
      | (e1, _) -> failwith "I was expecting to see an IN."
     )
  | IF::rest ->
     (match parseExp rest with
      | (e1, THEN::rest1) ->
         (match parseExp rest1 with
          | (e2, ELSE::rest2) -> let (e3, rest3) = parseExp rest2
                                 in (If(e1, e2, e3), rest3)
          | (e2, _) -> failwith "I was expecting to see an ELSE."
         )
      | (e1, _) -> failwith "I was expecting to see a THEN."
     )
```

Here you go:

```
# parse "let x = foo then 4";;
Exception: Failure "I was expecting to see an IN.".
# parse "if 3";;
Exception: Failure "I was expecting to see a THEN.".
# parse "if 3 then 8";;
Exception: Failure "I was expecting to see an ELSE.".
```

This is good, but the code looks messy to me. We can clean it up a bit by writing a function that will enforce a certain token in the given list.

```
(* consume: token -> token list -> token list
   Enforces that the given token list's head is the given token;
   returns the tail.
*)
let consume tok tokens =
  match tokens with
  | t::rest when t = tok -> rest
  | t::rest -> failwith ("I was expecting a " ^ (toString tok) ^
                         ", but I found a " ^ toString(t))
```

This way, error checking can be done in a separate place, and we can go with the happy path inside `parseExp`. See below:

```
let rec parseExp tokens =
  match tokens with
  | INT i :: rest -> (CstI i, rest)
  | BOOL b :: rest -> (CstB b, rest)
  | NAME x :: rest -> (Var x, rest)
  | LET::(NAME x)::EQUALS::rest ->
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetIn(x, e1, e2), tokens3)
  | IF::rest ->
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume THEN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     let tokens4 = consume ELSE tokens3 in
     let (e3, tokens5) = parseExp tokens4 in
     (If(e1, e2, e3), tokens5)
  | t::rest -> failwith ("Unsupported token: " ^ toString(t))

(* parseMain: token list -> exp *)
let parseMain tokens =
  let (e, tokens1) = parseExp tokens in
  let tokens2 = consume EOF tokens1 in
  if tokens2 = [] then e
  else failwith "Oops."
```

Parsing Binary Operators

Let us now add binary operators to the grammar.

```
(A)  main ::= exp EOF
(B)  exp  ::= INT
(C)         | NAME
(D)         | BOOL
(E)         | exp PLUS exp
(F)         | exp MINUS exp
(G)         | exp STAR exp
(H)         | exp SLASH exp
(I)         | LET NAME EQ exp IN exp
(J)         | IF exp THEN exp ELSE exp
```

Note that the new rules are quite different than the previous ones in that they don't start with a terminal while the others do. That is, when we see a `LET` as the first token of the given list, we can understand that we must be parsing a `LetIn`. Similarly, when we see an `IF`, we can understand that we must be parsing an `If`. We cannot do this for the `Add`, `Subt`, `Mult`, `Div` cases. You may be tempted to add a case such as

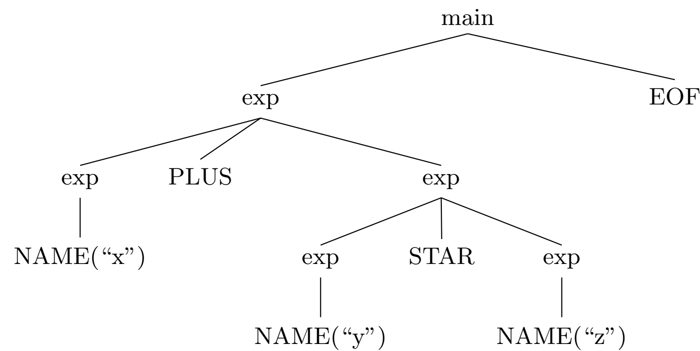
```
let rec parseExp tokens =
  match tokens with
```

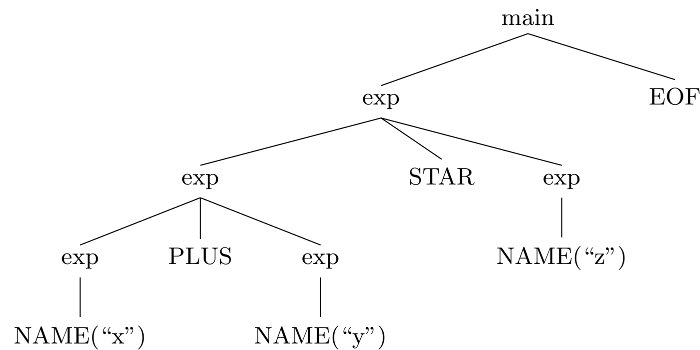
```
  ...
  | _ -> let (e1, tokens1) = parseExp tokens in
         match tokens1 with
         | PLUS::rest -> ...
```

But this does not work, because it has infinite recursion. Moreover, we have another serious problem brought by the new grammar rules: **Ambiguity**. We should now take a side step, and discuss ambiguity. Then, we'll come back to this point.

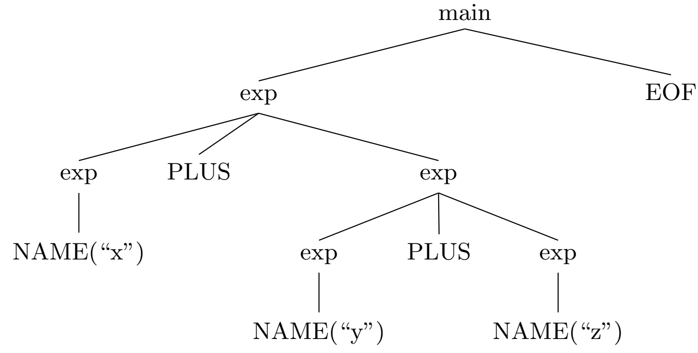
Ambiguity

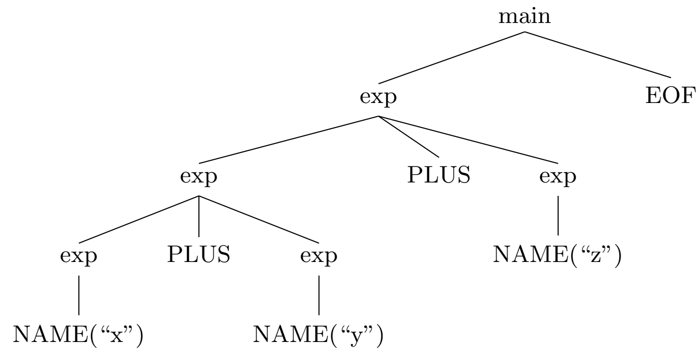
A grammar is **ambiguous** if more than one parse trees exist for the same input. Consider the last grammar given above. Here are two possible trees for the input "x + y * z".



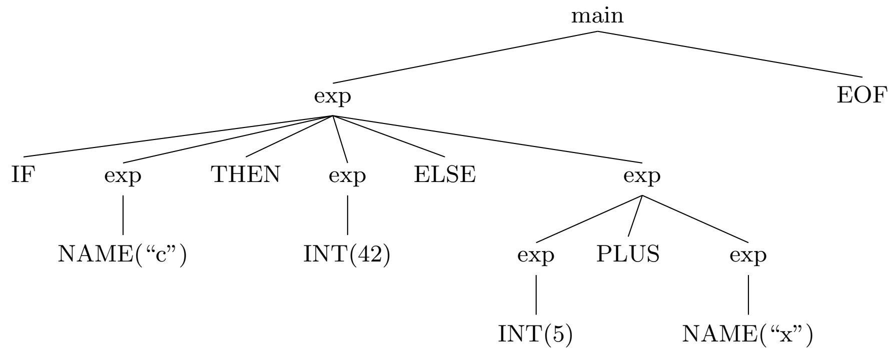


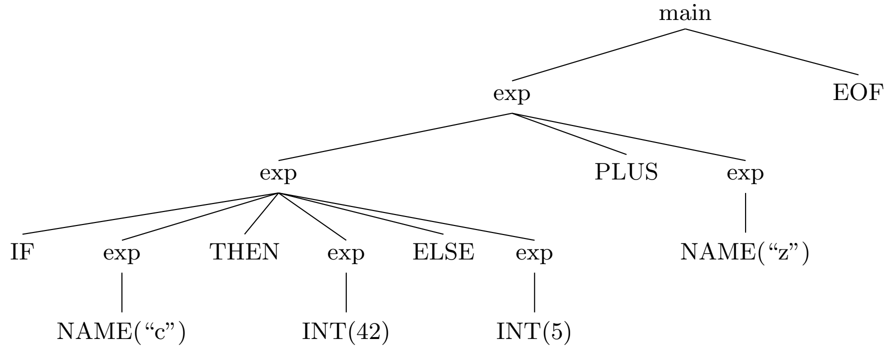
A similar ambiguity exists for "x + y + z":





This problem is not unique to binary operators. Consider "if c then 42 else 5 + x":

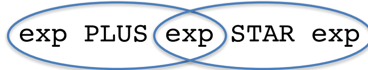




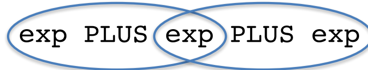
Intuitively, the ambiguity problem is caused by different rules having "overlaps" over terms and fighting for ownership. Consider rule (E) and (G):

```
(E)          | exp PLUS exp
(G)          | exp STAR exp
```

Both the left-end and the right-end of these rules are "open" in the sense that they are not delimited by a concrete token. Therefore, the left-end of rule (E) can overlap with the right-end of rule (G) on an `exp` term, like so:



We can make the same observation about the left- and right-ends of the rule (E) itself!



As another example, now also consider rules (I) and (J).

```
(E)          | exp PLUS exp
(G)          | exp STAR exp
(I)          | LET NAME EQ exp IN exp
(J)          | IF exp THEN exp ELSE exp
```

They have concrete tokens on their left-ends, and are "closed" at that side, in the sense that they cannot overlap with another production. However, their right-ends are "open" and can overlap with the left-ends of the rules (E), (F), (G), and (H). See the diagram below. Note that an overlap is not possible between rule (I) and (J), so there is no ambiguity caused by their combination.

So, essentially, two productions fight here over the ownership of the `exp` in the middle. Which one should win?

Computers are not smart machines. Humans need to drive and direct them. So, we need to decide which of these trees is the desired one. For this, we need to define **precedence** among rules. By precedence, we will decide which rule has priority over others. This way, the "winner" rule can be decided and the ambiguity can

be resolved. Precedence is defined *between* the rules; so, when a rule causes ambiguity with itself precedence won't help. In that case, we need to define **associativity**. We use associativity specification also to resolve the ambiguity between rules of same precedence.

We give the precedence and associativity definitions in the table below. With these, no more ambiguity exists; there is never more than one parse tree for an input. In this table, rules listed in the upper rows have lower precedence than the rows below them. We list rules (B), (C), and (D) here just for the sake of completeness. Because they contain tokens only, they cause no ambiguity whatsoever. In fact, you can consider them as "atoms" of the language, they cannot be broken down into pieces.

Precedence	Rule	Operator	Associativity
1 (lowest)	I,J		-
2	E,F	`+,-`	left
3	G,H	`*,/`	left
4 (highest)	B,C,D		-

Implementing the Parser

Phew, we're back! Let us implement the parser, this time taking the precedence/associativity table into consideration. Essentially, we will implement a parse function for each level in the table. Because `LetIn` (rule I) and `If` (rule J) have the lowest priority, if the input starts with a `LET`/`IF`, we will have a `LetIn`/`If` node at the top. So we should check for these cases first. If we do not find a `LET` or `IF`, we try for level 2.

```
let rec parseExp tokens =
  parseLevel1Exp tokens

and parseLevel1Exp tokens =
  match tokens with
  | LET::rest -> parseLetIn tokens
  | IF::rest -> parseIfThenElse tokens
  | _ -> parseLevel2Exp tokens

and parseLetIn tokens =
  match tokens with
  | LET::NAME(x)::EQUALS::rest ->
     let (e1, tokens1) = parseExp rest in
     let tokens2 = consume IN tokens1 in
     let (e2, tokens3) = parseExp tokens2 in
     (LetIn(x, e1, e2), tokens3)
  | _ -> failwith "Should not be possible."

and parseIfThenElse tokens =
  let rest = consume IF tokens in
  let (e1, tokens1) = parseExp rest in
  let tokens2 = consume THEN tokens1 in
  let (e2, tokens3) = parseExp tokens2 in
  let tokens4 = consume ELSE tokens3 in
  let (e3, tokens5) = parseExp tokens4 in
  (If(e1, e2, e3), tokens5)
```

What shall we do for level 2? Let us simplify the problem. Suppose there is no associativity for addition and subtraction, so that the only possible operands of '+' and '−' are level 3 and level 4 expressions. Based

on this simplification (which we will fix later), here is an implementation. First, parse a level 3 expression. If there is a following `PLUS` or `MINUS`, we need to parse a level 3 expression again, this time as the right operand. If no `PLUS` or `MINUS` exist, we simply return the level 3 exp that we just parsed. See the code below.

```
and parseLevel2Exp tokens =
  let (e1, tokens1) = parseLevel3Exp tokens in
  match tokens1 with
  | PLUS::rest ->
     let (e2, tokens2) = parseLevel3Exp rest
     in (Add(e1, e2), tokens2)
  | MINUS::rest ->
     let (e2, tokens2) = parseLevel3Exp rest
     in (Subt(e1, e2), tokens2)
  | _ -> (e1, tokens1)
```

We do exactly the same thing at level 3:

```
and parseLevel3Exp tokens =
  let (e1, tokens1) = parseLevel4Exp tokens in
  match tokens1 with
  | STAR::rest ->
     let (e2, tokens2) = parseLevel4Exp rest
     in (Mult(e1, e2), tokens2)
  | SLASH::rest ->
     let (e2, tokens2) = parseLevel4Exp rest
     in (Div(e1, e2), tokens2)
  | _ -> (e1, tokens1)
```

The 4th and the final level is easy:

```
and parseLevel4Exp tokens =
  match tokens with
  | INT i :: rest -> (CstI i, rest)
  | NAME x :: rest -> (Var x, rest)
  | BOOL b :: rest -> (CstB b, rest)
  | t::rest -> failwith ("Unsupported token: " ^ toString(t))
```

Let us test this parser:

```
# parse "f + g";;
- : exp = Add (Var "f", Var "g")
# parse "5 * k";;
- : exp = Mult (CstI 5, Var "k")
# parse "3 * 4 + 6";;
- : exp = Add (Mult (CstI 3, CstI 4), CstI 6)
# parse "3 + 4 * 6";;
- : exp = Add (CstI 3, Mult (CstI 4, CstI 6))
# parse "if c then 3 else x + 5";;
- : exp = If (Var "c", CstI 3, Add (Var "x", CstI 5))
# parse "let x = 7 in x + 5";;
- : exp = LetIn ("x", CstI 7, Add (Var "x", CstI 5))
```

Seems good! Let's go further:

```
# parse "3 * let x = 5 in 6";;
Exception: Failure "Unsupported token: LET".
```

Since a let-in has lower precedence than a binary expression, a let-in cannot occur as the left operand of a binary expression because otherwise we would be violating precedence rules. However, it should be perfectly fine to have a let-in as the right operand. This is not an ambiguous case because a let-in expression is "closed" at its left-end with a LET token. And similarly for if-then-else. We should allow let-in and if-then-else on the right-hand-side of binary expressions. To do this, you may be tempted to do something like this:

```
and parseLevel3Exp tokens =
  let (e1, tokens1) = parseLevel4Exp tokens in
  match tokens1 with
  | STAR::rest ->                (* ⇓ *)
     let (e2, tokens2) = parseLevel1Exp rest
     in (Mult(e1, e2), tokens2)
  ...
```

But this is incorrect! It would allow to have an addition as the right operand of multiplication, and that would violate precedence. Instead, we should just check for a LET and IF, and parse for only let-in and if-then-else, instead of the whole level 1.

In our implementation, we had made the assumption that binary expressions are not associative. So, we are parsing expressions of a lower level for both operands. Let's first check that this really does not allow association:

```
# parse "3 * 4 * 5";;
Exception: Failure "I was expecting a EOF, but I found a STAR".
```

As an attempt to improve this, we can parse a level **3** expression on the right-hand-side, instead of a level **4** expression. This will allow having multiplications/divisions side by side. The same discussion applies to addition and subtraction as well. Here is the next version:

```
and parseLevel2Exp tokens =
  let (e1, tokens1) = parseLevel3Exp tokens in
  match tokens1 with
  | PLUS::tok::rest ->
     (match tok with
      | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
               in (Add(e1, e2), tokens2)
      | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
               in (Add(e1, e2), tokens2)
      | t   -> let (e2, tokens2) = parseLevel2Exp (tok::rest)
               in (Add(e1, e2), tokens2)
     )
  | MINUS::tok::rest ->
     (match tok with
      | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
```

```
                in (Subt(e1, e2), tokens2)
      | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                in (Subt(e1, e2), tokens2)
      | t   -> let (e2, tokens2) = parseLevel2Exp (tok::rest)
                in (Subt(e1, e2), tokens2)
     )
  | _ -> (e1, tokens1)

and parseLevel3Exp tokens =
  let (e1, tokens1) = parseLevel4Exp tokens in
  match tokens1 with
  | STAR::tok::rest ->
     (match tok with
      | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                in (Mult(e1, e2), tokens2)
      | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                in (Mult(e1, e2), tokens2)
      | t   -> let (e2, tokens2) = parseLevel3Exp (tok::rest)
                in (Mult(e1, e2), tokens2)
     )
  | SLASH::tok::rest ->
     (match tok with
      | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                in (Div(e1, e2), tokens2)
      | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                in (Div(e1, e2), tokens2)
      | t   -> let (e2, tokens2) = parseLevel3Exp (tok::rest)
                in (Div(e1, e2), tokens2)
     )
  | _ -> (e1, tokens1)
```

Let's test.

```
# parse "3 * 4 * 5";;
- : exp = Mult (CstI 3, Mult (CstI 4, CstI 5))
# parse "3 * 4 * 5 / 6 * 2";;
- : exp = Mult (CstI 3, Mult (CstI 4, Div (CstI 5, Mult (CstI 6, CstI 2))))
# parse "3 * let x = 5 in 6";;
- : exp = Mult (CstI 3, LetIn ("x", CstI 5, CstI 6))
# parse "3 * let x = 5 in 6 + 2";;
- : exp = Mult (CstI 3, LetIn ("x", CstI 5, Add (CstI 6, CstI 2)))
# parse "1 + 2 * 3 - 5";;
- : exp = Add (CstI 1, Subt (Mult (CstI 2, CstI 3), CstI 5))
```

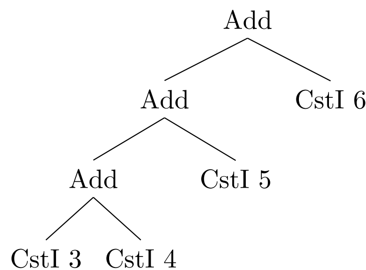
Seems like there is improvement! There is one small little tiny problem remaining: Our implementation produces right associative trees, but we want left associativity. I wish we could fix this problem simply by implementing the functions as

```
and parseLevel2Exp tokens =
  let (e1, tokens1) = parseLevel2Exp tokens in
  ...                         (* ⇑ *)
```

```
and parseLevel3Exp tokens =
  let (e1, tokens1) = parseLevel3Exp tokens in
  ...                        (* ⇑ *)
```

but this does NOT work because it leads to infinite recursion. We gotta find a solution.

Let us make an observation. Consider "3 + 4 + 5 + 6" and its desired AST:



The left-most node has to be a level 3 expression (in this case, it is `CstI 3`). Any right operand is a level 3 expression; it cannot be a level 2 expression, because otherwise associativity would be violated. Once we get a hold on the left-most expression, we can continue in a "rolling" fashion, where we keep accumulating the left operand as follows: as long as we see a `PLUS`, we should parse a level 3 (not 2!) expression following that `PLUS` token (this will be a right operand). Next, we combine that level 3 expression with the accumulating expression coming from the left to build the next accumulated expression. The general idea is similar to how `fold_left` works. To implement this, we will write a helper function that not only takes a list of tokens, but also the left-operand expression, i.e. the level 2 expression that has accumulated so far. The initial value for that accumulator is the first level 3 expression we parse out of the given token list. The idea that we discussed here for addition applies to the `MINUS`, `STAR`, and `SLASH` cases, too, of course. Here is the next version:

```
and parseLevel2Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | PLUS::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Add(e1, e2), tokens2)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Add(e1, e2), tokens2)
        | t   -> let (e2, tokens2) = parseLevel3Exp (tok::rest)
                 in helper tokens2 (Add(e1, e2))
       )
    | MINUS::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Subt(e1, e2), tokens2)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Subt(e1, e2), tokens2)
```

```
        | t   -> let (e2, tokens2) = parseLevel3Exp (tok::rest)
                 in helper tokens2 (Subt(e1, e2))
       )
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel3Exp tokens in
     helper tokens1 e1

and parseLevel3Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | STAR::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Mult(e1, e2), tokens2)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Mult(e1, e2), tokens2)
        | t   -> let (e2, tokens2) = parseLevel4Exp (tok::rest)
                 in helper tokens2 (Mult(e1, e2))
       )
    | SLASH::tok::rest ->
       (match tok with
        | LET -> let (e2, tokens2) = parseLetIn (tok::rest)
                 in (Div(e1, e2), tokens2)
        | IF  -> let (e2, tokens2) = parseIfThenElse (tok::rest)
                 in (Div(e1, e2), tokens2)
        | t   -> let (e2, tokens2) = parseLevel4Exp (tok::rest)
                 in helper tokens2 (Div(e1, e2))
       )
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel4Exp tokens in
     helper tokens1 e1
```

This is probably OK, but there seems to be a lot of code duplication. Let's clean it up a bit. For this, I'll write a function that tries to parse a let-in, or an if-then-else, or another expression based on a given parse function. I'll then use this function to improve code cleanness:

```
and parseLETorIForOther otherParseFun tokens =
  match tokens with
  | LET::rest -> let (e, tokens2) = parseLetIn tokens
                 in (e, tokens2)
  | IF::rest  -> let (e, tokens2) = parseIfThenElse tokens
                 in (e, tokens2)
  | _         -> let (e, tokens2) = otherParseFun tokens
                 in (e, tokens2)

and parseLevel2Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | PLUS::rest ->
       let (e2, tokens2) = parseLETorIForOther parseLevel3Exp rest
       in helper tokens2 (Add(e1, e2))
    | MINUS::rest ->
```

```
      let (e2, tokens2) = parseLETorIForOther parseLevel3Exp rest
      in helper tokens2 (Subt(e1, e2))
   | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel3Exp tokens in
     helper tokens1 e1

and parseLevel3Exp tokens =
  let rec helper tokens e1 =
    match tokens with
    | STAR::rest ->
      let (e2, tokens2) = parseLETorIForOther parseLevel4Exp rest
      in helper tokens2 (Mult(e1, e2))
    | SLASH::rest ->
      let (e2, tokens2) = parseLETorIForOther parseLevel4Exp rest
      in helper tokens2 (Div(e1, e2))
    | _ -> (e1, tokens)
  in let (e1, tokens1) = parseLevel4Exp tokens in
     helper tokens1 e1
```

Looks better. Let's test.

```
# parse "3 + 4 + 5 + 6";;
- : exp = Add (Add (Add (CstI 3, CstI 4), CstI 5), CstI 6)
# parse "3 - 4 - 5 - 6";;
- : exp = Subt (Subt (Subt (CstI 3, CstI 4), CstI 5), CstI 6)
# parse "3 - 4 + 5 - 6 + 7";;
- : exp = Add (Subt (Add (Subt (CstI 3, CstI 4), CstI 5), CstI 6), CstI 7)
# parse "3 - 4 * 5 - 6 * 7";;
- : exp = Subt (Subt (CstI 3, Mult (CstI 4, CstI 5)), Mult (CstI 6, CstI 7))
# parse "3 * 4 + 5 * 6 + 7";;
- : exp = Add (Add (Mult (CstI 3, CstI 4), Mult (CstI 5, CstI 6)), CstI 7)
# parse "let x = 2 in 3 + 4 + let x = 5 in 6 + 7";;
- : exp =
LetIn ("x", CstI 2,
 Add (Add (CstI 3, CstI 4), LetIn ("x", CstI 5, Add (CstI 6, CstI 7))))
```