1 Lexing and Parsing

1.1 Using the Parser Generators

There are two special types of files.

- `Lexer.x` (or any `.x` file): here, you will be writing the specifications for the lexer here (the token format). We will use the Haskell tool `alex` to translate this file to `Lexer.hs` (i.e., generate the lexer).
- `Parser.y` (or any `.y` file): here, you will describe the form of the parser (the grammar). This will be fed to a tool called `happy`, which will be translated to the file `Parser.hs`.

1.1.1 Regular Expressions

Before we talk about lexing, we first need to talk about regular expressions. Simply put, a regular expression is a sequence of characters that specifies a search pattern in text. For our **token format** file (used by `alex`), regular expressions have one of the following form[1]:

- a **character class**: written as a bunch of characters in a square bracket, this particular regular expression will match any of the characters in the brackets. In other words, for the character class `[c1 c2 ... cn]`, this will match *any one* of the characters `c1 c2 ... cn`.

  There are several special character classes.

  - `[0-9]` will match any *digit*.
  - `[a-z]` will match any *lowercase letter*.
  - `[A-Z]` will match any *uppercase letter*.
  - `[a-z A-Z]` will match *any letter*.

  If a character class has one character, then you do not need to put any brackets; in other words, `[c1]` is equivalent to `c1`.
- `R1 R2` matches a string `s1 ++ s2`, where `s1` matches `R1` and `s2` matches `R2`. For example, `[0-9] [0-9]` matches any two-digit string.
- `R+` matches *one or more repetitions* of what `R` matches. For example, `[0-9]+` matches a natural number.
- `R*` matches *zero or more repetitions* of what `R` matches.

(Quiz.) Which of the following strings are matched by `[a-z A-Z] [a-z A-Z 0-9]*`?

(a) (Empty string)

(b) `5`

(c) `x5`

(d) `x`

(e) C and D

The answer is **E**. The regular expression matches any *one* letter (`[a-z A-Z]`) followed by *zero oe more* letters and digits (`[a-z A-Z 0-9]*`).

[1] These may differ in different languages.

We can also name regular expressions. For example, instead of writing out `[0-9]` or `[a-z A-Z]` over and over again, we can instead write

```
$digit = [0-9]
$alpha = [a-z A-Z]
```

Then, instead of writing `[a-z A-Z] [a-z A-Z 0-9]*`, we can write

```
$alpha [$alpha $digit]*
```

1.1.2 Lexing

We use the tool `alex` to generate the lexer. We want to give `alex` a `.x` file, describing the lexer. There are several components in a `.x` file.

- First, you need to list the kinds of **tokens** we have in the language.

  ```
  data Token
      = NUM    AlexPosn Int
      | ID     AlexPosn String
      | PLUS   AlexPosn
      | MINUS  AlexPosn
      | MUL    AlexPosn
      | DIV    AlexPosn
      | LPAREN AlexPosn
      | RPAREN AlexPosn
      | EOF    AlexPosn
  ```

  We use a Haskell datatype to represent these possible tokens. This is completely normal Haskell syntax.

  Note that each constructor has the parameter `AlexPosn`. The whole point of this is to store information regarding where the token was found[2]. Then, we might have some other parameters (e.g., `NUM` with `Int`) to *store* the data associated with this type.

- Next, you need to list the **token rules**.

  ```
  -- Regex:                          Function: AlexPosn -> String -> Token
  \+                              { \p _ -> PLUS   p }
  \-                              { \p _ -> MINUS  p }
  \*                              { \p _ -> MUL    p }
  \/                              { \p _ -> DIV    p }
  \(                              { \p _ -> LPAREN p }
  \)                              { \p _ -> RPAREN p }
  $alpha [$alpha $digit \_ \']* { \p s -> ID     p s }
  $digit+                         { \p s -> NUM p (read s) }
  ```

  Each line is a token rule. The content on the left-hand side is a **regular expression** that tells you what the token looks like. The content on the right-hand side tells `alex` how to construct this `Token` type from this information.

  For example, let's look at

  ```
  \+                              { \p _ -> PLUS   p }
  ```

[2] Otherwise, you do not need to worry about this.

Obviously, the string to the left is a regular expression. It should be noted that a lot of the regular expressions have a backslash prepended to it (for example, this `+` regular expression has a backslash before it). The reason for this is because these characters, or at least some of them, have a special meaning in regular expressions, so we need to *escape* them so we interpret the `+` literally as opposed to in a regular expression class.

Now, what is going on at the right side? On the right, we have a **Haskell function** surrounded by curly braces. Each function has **two arguments**:

- The first argument is the position (`AlexPosn`).
- The second one is the string that was parsed.

For this, we consider a more interesting example:

```
$alpha [$alpha $digit \_ \']*       { \p s -> ID     p s }
```

What this regular expression is saying is to match anything that looks like an identifier (starts with a letter followed by any letter, digit, underscore, or tick). Then, the function takes in two arguments, `p` (for position) and `s` (the string that was parsed) and returns an `ID` with the position and string parsed.

Note that the last regular expression above is no different; the only difference is that we have `read s`, which converts the string read into an `Int`.

- Finally, we need to run the lexer. From the token rules, `alex` will generate a function `alexScan` which
  - Given the input string, find the longest prefix `p` that matches one of the rules.
  - If `p` is empty, it fails (at the beginning of the string, there's something that doesn't match).
  - Otherwise, it will convert `p` into a token and returns the *rest of the string*.

  As implied, this function will be executed in a loop so you can read the whole input string. Note that, in the programming assignments, we wrap this function into a function

  ```
  parseTokens :: String -> Either ErrMsg [Token]
  ```

  which repeatedly calls `alexScan` until it consumes the whole input string or fails. Note that the return type here means that we will return *either* `ErrMsg` or `[Token]`.

  Thus, testing the function gives us

  ```
  $ parseTokens "23 + 4 / off -"
  Right [ NUM (AlexPosn 0 1 1) 23
      , PLUS (AlexPosn 3 1 4)
      , NUM (AlexPosn 5 1 6) 4
      , DIV (AlexPosn 7 1 8)
      , ID (AlexPosn 9 1 10) "off"
      , MINUS (AlexPosn 13 1 14)
      ]

  $ parseTokens "%"
  Left "lexical error at 1 line, 1 column"
  ```

(Quiz.) What is the result of `parseTokens "92zoo"` (positions omitted for readability)?

(a) Lexical error

(b) `[ID "92zoo"]`

(c) `[NUM "92"]`

(d) `[NUM "92", ID "zoo"]`

The answer can either be **A** or **D**, depending on whether you have whitespace sensitivity. If we care about whitespace, then **A** is the answer. If we do not care about whitespace, then **D** is the answer.

Remark: In a real programming langauge, you probably care about whitespace sensitivity.

1.1.3 Parser

Just like how `alex` is for lexing, we use `happy` for the parser. We input to `happy` a `.y` file that describes the *grammar*. As always, there are several components.

- Note that something like

```
data Aexpr
    = AConst  Int
    | AVar    Id
    | APlus   Aexpr Aexpr
    | AMinus  Aexpr Aexpr
    | AMul    Aexpr Aexpr
```

  is the *abstract* syntax. This tells us that there is a binary operation called (for example) `APlus` with two `Aexpr`, but this doesn't tell us *what* this might look like in source code. Thus, what we want is the *concrete syntax*; in particular,

  - What programs look like when written as text.
  - How to map that text into the abstract syntax.

  What is a grammar? A grammar is a recursive definition of a set of parse trees; in particular, a grammar is made of

  - **Terminals**: the leaves of the tree (corresponds to tokens).
  - **Nonterminals**: the internal nodes of the tree (corresponds to more complex constructs, all kinds of subexpressions).
  - **Production Rules**: describes how to "produce" a non-terminal from terminals and other non-terminals (i.e., what children each nonterminal can have).

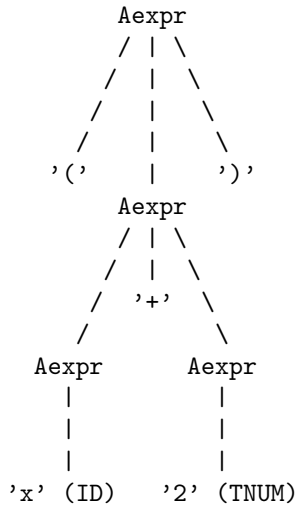
  So, consider the following example:

```
Aexpr :      -- Non-term Aexpr can be either:
      | TNUM             -- Term of format "number", or
      | ID               -- Term of format "identifier", or
      | '(' Aexpr ')'    -- Term '(', non-term Aexpr, term ')'
      | Aexpr '*' Aexpr  -- Non-term Aexpr, term '*', non-term Aexpr
      | Aexpr '+' Aexpr  -- Non-term Aexpr, term '+', non-term Aexpr
      | Aexpr '-' Aexpr  -- Non-term Aexpr, term '-', non-term Aexpr
```

Here, we have a single nonterminal (`Aexpr`). Then, our terminals are things like `TNUM`, `ID`, `'('`, `')'`, `'*'`, `'+'`, and `'-'`. Notice that we have 6 production rules, which describes how our `Aexpr` can be represented as a sequence of other things. For example,

- `Aexpr` can be written as a `TNUM`.
- `Aexpr` can be written as an `ID`.
- `Aexpr` can be written as an `Aexpr` surrounded by parentheses.
- And so on.

So, a parse tree for the string `(x + 2)` will look like



(Quiz.) Using the above grammar of `Aexpr`, which string *cannot* be parsed as `Aexpr`?

(a) `x`
(b) `x 5`
(c) `(x +) 5`
(d) `x + 5 + 1`
(e) B and C

The answer is **E**. For B, there is no rule that lets us generate both an identifier and number. For C, there is no rule that lets us generate `(x +)`.

- So far, our grammar tells us whether a string is syntactically correct or not. However, we want to convert our string to an AST.

```
parse :: String -> AExpr
```

An **attribute grammar** associates a value with each node in the parse tree such that

- Each production is annotated with a rule.
- Each rule computes the value[3] of a non-terminal from the values of its children.

So, the attribute grammar for arithmetic expression is as follows:

[3]The AST (i.e. the Haskell value of type `AExpr`.)

```
--      Format                    Value
Aexpr : TNUM                    { AConst $1    }
      | ID                      { AVar   $1    }
      | '(' Aexpr ')'           { $2           }
      | Aexpr '*' Aexpr         { AMul   $1 $3 }
      | Aexpr '+' Aexpr         { APlus  $1 $3 }
      | Aexpr '-' Aexpr         { AMinus $1 $3 }
```

Like with the token rules, there is a format for each expression.

- The left-hand side is a context-free grammar (just like what we saw in the example above).
- The right-hand side is the attributes part. In particular, you have a **Haskell expression** surrounded by curly braces. This will then construct the value associated with each node. So, essentially, you can think of the right-hand side as a regular Haskell expression, with the dollar signs.

What do the dollar signs mean? Well,

- `$1` refers to the *value* of the first child.
- `$2` refers to the value of the second child.
- And so on.

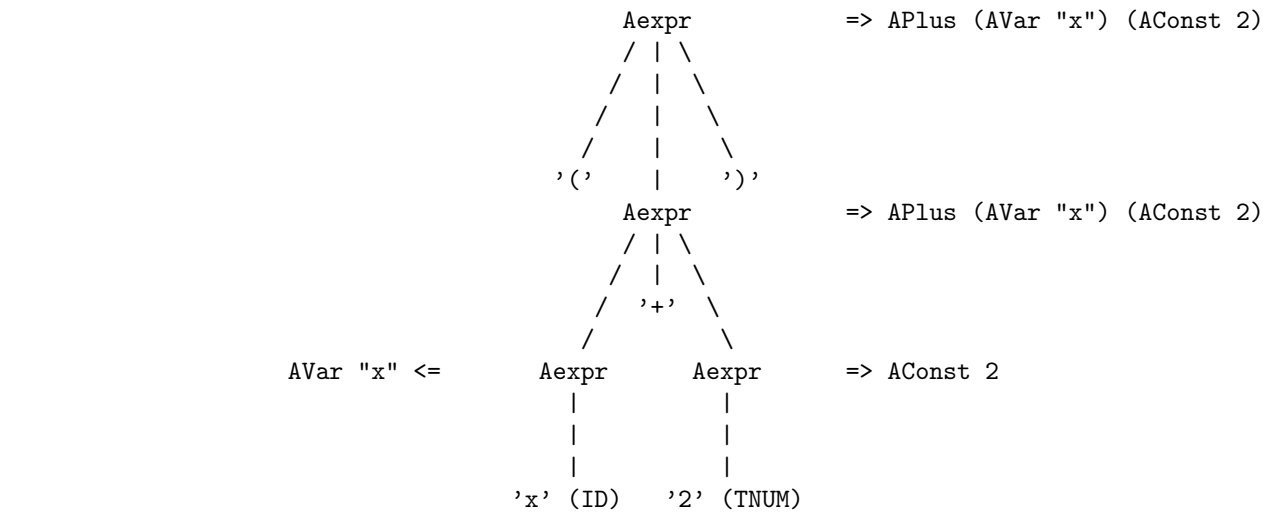
Let's take this line (from the grammar) as an example:

```
Aexpr '+' Aexpr         { APlus  $1 $3 }
```

To construct the value, i.e. the AST, for the plus node, you should apply the constructor `APlus` to the value of the first child (`$1`) and the value of the third child (`$3`). Here, each child (`$n`) is annotated below:

```
    Aexpr '+' Aexpr         { APlus  $1 $3 }
--  $1    $2  $3
```

So, there's no reason for us to use the second child, `$2`, since it's just literally `'+'`. So, going back to the parse tree for `(x + 2)`, the AST would look like So, a parse tree for the string `(x + 2)` will look like



How do we compute the value of a terminal? In particular,

- how do we map terminal `'x'` to string `"x"`?
- how do we map terminal `'2'` to integer `2`?

- We have to describe what the values of the terminals are to `happy`. Recall that the terminals correspond to the tokens returned by the lexer. So, in the `.y` file, we have to declare which termianls in the rules correspond to which tokens from the `Token` datatype.

```
-- Terminals:   Tokens from lexer:
    TNUM        { NUM _ $$ }
    ID          { ID _ $$  }
    '+'         { PLUS _   }
    '-'         { MINUS _  }
    '*'         { MUL _    }
    '/'         { DIV _    }
    '('         { LPAREN _ }
    ')'         { RPAREN _ }
```

Each thing on the left is a terminal (which appears in the production rules). Each thing on the right is a Haskell pattern for the datatype `Token`. So, the `$$` designates one parameter of a token constructor as the **value** of that token.

(Quiz.) What is the value of the root `AExpr` node when parsing `1 + 2 + 3`?

```
Aexpr : TNUM                    { AConst $1    }
      | ID                      { AVar   $1    }
      | '(' Aexpr ')'           { $2           }
      | Aexpr '*' Aexpr         { AMul   $1 $3 }
      | Aexpr '+' Aexpr         { APlus  $1 $3 }
      | Aexpr '-' Aexpr         { AMinus $1 $3 }
```

(a) Cannot be parsed as `AExpr`.

(b) `6`

(c) `APlus (APlus (AConst 1) (AConst 2)) (AConst 3)`

(d) `APlus (AConst 1) (APlus (AConst 2) (AConst 3))`

The answer is both **C** and **D**. If we draw parse trees, we would be able to get both answers.