1 Parsers

A parser is a piece of software that takes text and breaks it up into parts to build a hierarchical structure out of them. The family of valid inputs and structure of the output of a parser is defined by a language commonly described as a *context-free grammar* (CFG). A parser’s output is called a *parse tree* or an *abstract syntax tree* (AST). This tree defines the hierarchical structure of the output, and the nodes contain the input text.
1.1 Input - Tokens

A parser’s input is a list of tokens. Tokens are (usually small) strings, created by segmenting the original input text using a scanner (sometimes called a reader).

1.2 Language

A parser is created to parse text of a certain language. The language defines the structure of the parse tree and the family of texts that will be handled by the parser. If the input text does not belong to the parser’s language, the parser will reject the text.

Some languages contain ambiguities - parts of the language which can be parsed in more than one way. This means that for some inputs two different parse trees can be created from that input. Usually, ambiguity has to be addressed before a language can effectively and efficiently be parsed.

1.2.1 Reminder, CFG example

Below an example of a language defined using CFG. This is a language of simple assignments of the form < var > = < num >:

1. \[ S \rightarrow A \,'='\,B \]
2. \[ A \rightarrow (a\texttt{-}z)^+ \]
3. \[ A \rightarrow (A\texttt{-}Z)^+ \]
4. \[ B \rightarrow '0' \mid '1'\texttt{-}'9' \texttt{(}'0'\texttt{-}'9'\texttt{)}^* \]

A CFG is composed of Derivation Rules, written as \(< nt >\rightarrow< prod >\), with a Non-terminal on the left-hand side (LHS) and a Production on the right hand side (RHS). A non-terminal can be Derived into more than one production, this is denoted with a Disjunction (’|’), or by adding another derivation rule for that same non-terminal.

1.3 Parser families

Parsers vary widely between each other, both method of operation and by their limitations. The two largest families of parsers are the Top-Down parsers and the Bottom-Up parsers.
• Bottom-Up parsers try to construct the parse tree by building the leaves of the parse tree, then recursively building parent nodes until reaching the root. They do this by applying the production rules in the CFG backwards (reducing the RHS to produce the LHS).

• Top-Down parsers start at the root of the parse tree, and use the CFG’s production rules to produce children nodes until reaching leaves (i.e. terminal tokens).

In both Top-Down and Bottom-Up, the parser maintains some string of terminals and non-terminals that describe the frind of the tree. The nodes on the fringe are the ones that still need to be expanded upon.

A common sub-family of Top-Down parsers are the **LL-parsers** which read their input from left to right and apply derivation rules to the leftmost non-terminal in the intermediate state. The parser in your compiler will be implemented using the **Parser Combinators** technique which produces LL parsers.

## 2 Parser Combinators

### 2.1 Reminder: High-order functions & Combinators

A higher-order function is a function that takes another function as argument, or returns a function (or both).

A combinator is a high order function that doesn’t do anything other than applying the parameters (functions) it receives. This means that combinators combine functions together to create control flow between them.

### 2.2 Parsers combinators

Parser Combinators are higher-order functions which take parsers as their input and construct new parsers as their output. This is a compositional way of constructing parsers and so it allows a programmer to create complex parsers from simpler ones - the result is simpler code that looks a lot like the Context Free Grammar (**CFG**) which defines the language.

### 2.3 Parsers

Parsers created by the parsers combinators package we’ll use take a list of tokens as their input and return a pair of (**accepted tokens, remaining tokens**)
2.4 How do they work?

Let’s take a look at the char function in our pc.ml package.

# char;;
- : char -> char list -> char * char list = <fun>

The char parser-constructor takes a single character, \(C\), and returns a parser (i.e. function) which takes a list of character and returns a pair. That pair is composed of the first character of the list, if it’s \(C\), and the rest of the list. So for example:

```
# let _a_ = (char 'a');;
# _a_ (string_to_list "abc");;
- : char * char list = ('a', ['b'; 'c'])
```

If the first char on the list is not \(C\), the parser will raise an X_no_match exception.

```
# let _b_ = (char 'b');;
# _b_ (string_to_list "abc");
Exception: X_no_match.
```

Now we’ll look at the caten combinator. When we catenate the two parsers _a_ and _b_ we’ll get a parser that accepts the list ['a'; 'b'] (i.e. the string "ab")

```
# let _ab_ = caten _a_ _b_;;
# _ab_ (string_to_list "abc");;
- : (char * char) * char list = (('[a', 'b'), ['c'])
```

The caten combinator essentially takes two functions (since parsers created using PC are functions), and constructs a new function (i.e. parser). The caten function adds the control flow that sends the input into the first parser, and then sends the remaining list of that first parser to the second parser. In a similar fashion, the disj combinator (which implements disjunction) takes to functions, and constructs a sort of "or" statement between them, where if the first parser rejects the input (i.e. raises X_no_match) the input is then sent to the second parser instead.

So in general, parser combinators are functions that take some functions (i.e. parsers) and construct a function with the appropriate control flow to combine them into one function.

3 Example: Simple arithmetic expressions

In this example, we will construct a parser for a simple grammar using parser combinators. The grammar will be of addition and multiplication of numbers, e.g.
3.1 Proposed CFG

A possible CFG to describe a language of inline arithmetic expressions:

1. \(<expr> \rightarrow <num> | <expr> <op> <expr> | <nested-expr>
2. \(<nested-expr> \rightarrow <lparen> <expr> <rparen>
3. \(<op> \rightarrow <expop> | <mulop> | <addop>

However, this grammar is in fact not possible to parse using parser combinators (or any LL parser). The above language contains *Left Recursion*. Left recursion is a production rule that produces its left hand side as the leftmost token (like the second production of \(<expr> \) above, \(<expr> <op> <expr> \)). This will lead the parser into an infinite recursion. Continuously deriving \(<expr> \) like so:

\[
<\ expr > \rightarrow <\ expr > < op > <\ expr > \rightarrow <\ expr > < op > <\ expr > < op > <\ expr > \ldots
\]

The parser can’t rule out the \(<expr> \rightarrow <expr> <op> <expr> \) production without looking ahead in the input. For any lookahead length, there might be an input that requires looking further ahead in the input. This means that the parser will continuously apply the same production rule on the same non-terminal indefinitely.

We could try and fix this issue by using instead the production rule \(<expr> \rightarrow <num> <op> <expr> \) (note, we’ll also need to add \(<expr> \rightarrow <nested-expr> <op> <expr> \)). The will result in a new problem - right associative addition and multiplication.

In addition to being left recursive, the above CFG is ambiguous since \(<expr> <op> <expr> \) can be derived more than one way. For example: 1 + 2 * 3 can be derived in two (equally valid) ways:

\[
1 + 2 * 3 \quad \text{and} \quad 1 + 2 * 3
\]
The structures of these two parse trees suggest very different semantics. On the left we have $1 + (2 \times 3)$, but on the right we have $(1 + 2) \times 3$.

It’s easy for us to see the issue with the parse tree describing $(1 + 2) \times 3$. Since, when evaluated, this expression would produce a different result than expected. However, this is not the issue in the context of parsers. Remember, parsers don’t deal with semantics or evaluations - only structure. The problem above is the structure of the expression is undefined by the CFG.

3.2 Correct CFG

**Remark.** Since we usually don’t use a tokenizer (scanner) when using our PC package (i.e., every char is a token), we define parsers that will be responsible for tokenization. Doing so makes our parsers much simpler to write (especially considering delimiters).

**Token parsers**

1. `<lparen>` → `<space>`* `'('` `<space>`*
2. `<digit>` → `'0'-'9`
3. `<rparen>` → `<space>`* `')'` `<space>`*
4. `<expop>` → `'^'` `<space>`*
5. `<mulop>` → `<space>`* `'*'` `<space>`*
6. `<addop>` → `<space>`* `'+'` `<space>`*

**Production rules**

1. `<num>` → `<digit>`+
2. `<paren>` → `<num>` | `<lparen>` `<expr>` `<rparen>`
3. `<exp>` → `( <paren> `<expop>`)* `<paren>`
4. `<mul>` → `<exp>` ( `<mulop>` `<exp>`)*
5. `<add>` → `<mul>` ( `<addop>` `<mul>`)*
6. `<expr>` → `<add>`

The way this grammar works is by pushing the syntax with higher precedence semantics further down the parse tree. Nodes deeper in the parse tree correspond to higher precedence when processing the tree (e.g., when evaluating the parsed expression).
This cascade of parsers where the lower-precedence productions produce the higher-precedence ones (e.g. <add> produces <mul>, <mul> produces <exp> etc.) is required to make sure our CFG contains no left-recursions and to make it unambiguous. This pattern occurs often when parsing infix-style syntax.