**Parser Combinators**

For the task of implementing a parser, we’ve seen two options so far:

* Hand-written recursive-descent parsers
* Compiler-compilers

We now briefly discuss a third, intermediate one: parser combinators.

**Parsers are Functions**

From a conceptual point of view, a parser is just a function: it takes as an input a stream of characters or tokens, and either successfully consumes a part of the input and returns “success”, or fails to read a proper sequence and fails. Parser combinators are operators that combine several such parsers to build a new one.

From two parsers $A$and $B$, the two basic operations are:

* sequencing: construct the parser $A \sim B$that recognizes the language $L(A) \cdot L(B)$
* alternation: construct the parser $A | B$that recognizes the language $L(A) \cup L(B)$

In a programming language that supports lazy evaluation, it is possible to implement these two operations in a very natural way. Additionally, when the language supports a form of operator overloading, it is possible to construct parsers in a way that looks very close to context-free grammars; we’ll see how we can for instance write a parser for arithmetic expressions that looks like this:

[def](http://scala-lang.org) ID : Parser = "x" | "y" | "z"

[def](http://scala-lang.org) expr : Parser = factor ~ ((

"+" ~ factor

| "-" ~ factor

) | epsilon)

[def](http://scala-lang.org) factor : Parser = term ~ ((

"\*" ~ term

| "/" ~ term

) | epsilon)

[def](http://scala-lang.org) term : Parser = (

"(" ~ expr ~ ")"

| ID

| NUM

)

**Implementing Parser Combinators in Scala**

See the code [here](http://lara.epfl.ch/w/cc10:parsing_combinators_example). Observe how simple the definitions of Sequence and Alternative are. Some remarks on this code:

* Sequence and Alternative take lazy arguments. Why?
* There are no semantic actions (the parser just recognizes a string).
* White spaces are not handled specially, ie. “space” is a character like any other.
* Where does backtracking occur?

The second point can be improved by adding a return value as part of the Success class (ie. on success, parsers return the remaining stream and a value).

**Note on the ~ combinator**

The implementation of ~ as shown in the source file above does not always produce a parser that recognizes $L(A) \cdot L(B)$. As an example, consider a parser $A$that recognizes any number of repetitions of “a”, and a parser $B$that recognizes the words starting with one “a” and immediately followed by any number of “b”. The sequence “a” “a” “a” “b” will not be recognized as belonging to $L(A \sim B)$by the combined parser, because the parser $A$in the combination will consume all “a” and $B$will then fail.

**Scala Standard Library Implementation**

The [standard library](http://www.scala-lang.org/api/current/index.html) includes an implementation of parser combinators (in the package scala.util.parsing.combinator). There are more combinators, for instance the \* and ? operators, as well as full support for semantic actions (see the ^^ combinator).