# 50.054 Syntax Analysis

ISTD, SUTD

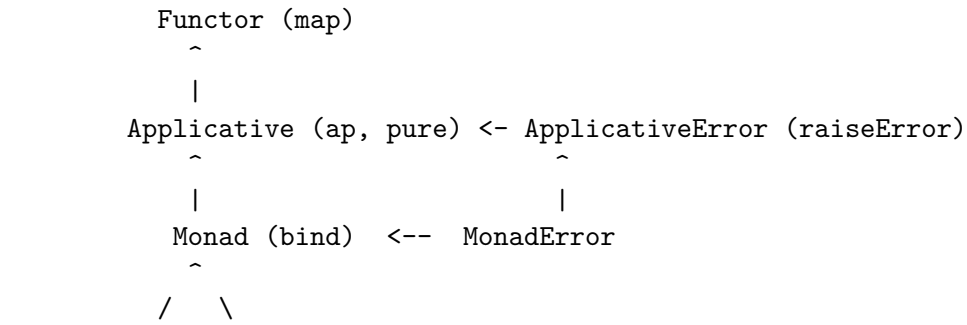
# Learning Outcomes

1. Describe the roles and functionalities of lexers and parsers in a compiler pipeline
2. Describe the top-down parsing

# Recap Monad

- List Monad
- `Option` Monad and `EitherErr` Monad
- `Reader` Monad (with `ReaderReader`)
- `State` Monad (with `StateMonad`)

# Recap Functor, Applicative and Monad

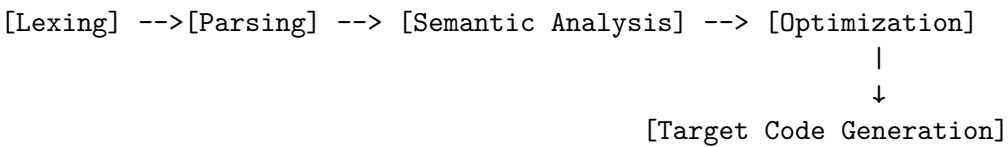


# Back to the big picture

- We want to build a compiler
- We just use Scala/FP as the implementation tool

# Compiler Pipeline

- Recall compilation is a process of mapping source language to target language in the target platform



# Lexing

- Input - source program in string, i.e. a list of characters.
- Output - a list of lexical tokens
- Fails when something it can't be recognized as a lexical token

# What is a lexical token?

A lexical token is a terminal from the syntax grammar.

`<<Grammar 1>>`

$$
\begin{array}{rrcl}
(\texttt{JSON}) & J & ::= & i \mid \ 's' \mid \ [] \mid \ [IS] \mid \{NS\} \\
(\texttt{Items}) & IS & ::= & J, IS \mid J \\
(\texttt{Named Objects}) & NS & ::= & N, NS \mid N \\
(\texttt{Named Object}) & N & ::= & 's' : J
\end{array}
$$

What are the non terminals? What are the terminals?

Terminals = lexical tokens

# What is a lexical token?

A lexical token is a terminal from the syntax grammar.

<<Grammar 1>>

$$
\begin{array}{rrcl}
\texttt{(JSON)} & J & ::= & i \mid {}'s' \mid [] \mid [IS] \mid \{NS\} \\
\texttt{(Items)} & IS & ::= & J, IS \mid J \\
\texttt{(Named Objects)} & NS & ::= & N, NS \mid N \\
\texttt{(Named Object)} & N & ::= & 's' : J
\end{array}
$$

- The set of tokens are $i$, $'$, $s$ [, ], {, }, ,, : and .
- However $i$ denotes an integer, $s$ denotes a string.
- The lexer should be able to handle that, e.g.
  - if the leading character is a digit, consume the following digits, then create an int token.
  - if the leading character is a $'$, consume everything until we see another $'$. then create a string token.

# Lexical Token data type

```
enum LToken { // lexical tokens
    case IntTok(v:Int)
    case StrTok(v:String)
    case SQuote
    case LBracket
    case RBracket
    case LBrace
    case RBrace
    case Colon
    case Comma
    case WhiteSpace
}
```

# Lexer

```
{'k1':1,'k2':[]}
```

The lexer function `lex(s:String):List[LToken]` should return

```
List(LBRace,SQuote,StrTok("k1"),SQuote,Colon,IntTok(1),Comma,SQuote,
StrTok("k2"),SQuote,Colon,LBracket, RBracket,RBrace)
```

# Regex


```scala
import scala.util.matching.Regex.*

val date = raw"(\d{4})-(\d{2})-(\d{2})".r

"2004-01-20" match {
  case date(year, month, day) =>
    s"${year} was a good year for PLs."
}
```

# Implementing a Lexer using Regex

```
import scala.util.matching.Regex.*
import LToken.*
type Error = String
val integer = raw"(\d+)(.*)".r
val string = raw"([^']*)(.*)".r
val squote = raw"(')(.*)".r
val lbracket = raw"(\[)(.*)".r
val rbracket = raw"(\])(.*)".r
val lbrace = raw"(\{)(.*)".r
val rbrace = raw"(\})(.*)".r
val colon = raw"(:)(.*)".r
val comma = raw"(,)(.*)".r
```

```
def lex_one(src:String):
    Either[String, (LToken, String)] = src match {
    case integer(s, rest) =>
        Right((IntTok(s.toInt), rest))
    case squote(_, rest) =>
        Right((SQuote, rest))
    case lbracket(_, rest) =>
        Right((LBracket, rest))
    case rbracket(_, rest) =>
        Right((RBracket, rest))
    case lbrace(_, rest) =>
        Right((LBracket, rest))
    case rbrace(_, rest) =>
        Right((RBracket, rest))
    case colon(_, rest) =>
        Right((Colon, rest))
    case comma(_, rest) =>
        Right((Comma, rest))
    case string(s, rest) =>
        Right((StrTok(s), rest))
    case _ =>
        Left(s"lexer error")
}
```

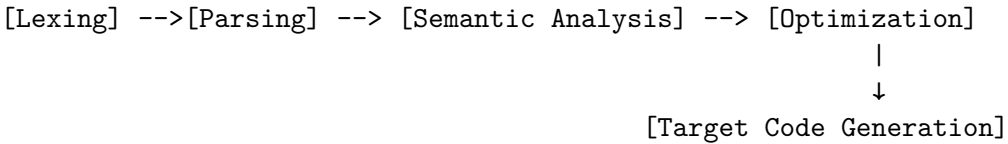
# Implementing a Lexer using Regex

```
def lex(src:String):Either[Error, List[LToken]] = {
    def go(src:String, acc:List[LToken]):Either[Error, List[LToken]] = {
        if (src.length == 0) { Right(acc) }
        else { lex_one(src) match {
                case Left(error) => Left(error)
                case Right((ltoken, rest)) =>
                    go(rest, acc++List(ltoken))
            }
        }
    }
    go(src, List())
}
```

## Other implementation approaches

- Using Parser Combinator
  - We will look into this again
- Using existing Lexer Generators

# Compiler Pipeline



# Parsing

- Input - A list of lexical tokens.
- Output - A parse tree
- Fails when the input can't be parsed by grammar rule.

# Parsing

- Input - A list of lexical tokens.
- Output - A parse tree
- Fails when the input can't be parsed by grammar rule.

```
def parse(tokens:List[LToken]):??? = ???
```

what should the returned type be?

# Abstract Syntax Tree

<<Grammar 1>>

$$
\begin{array}{rrcl}
\text{(JSON)} & J & ::= & i \mid \text{'}s\text{'} \mid [] \mid [IS] \mid \{NS\} \\
\text{(Items)} & IS & ::= & J, IS \mid J \\
\text{(Named Objects)} & NS & ::= & N, NS \mid N \\
\text{(Named Object)} & N & ::= & \text{'}s\text{'} : J
\end{array}
$$

```
enum Json {
    case IntLit(v:Int)
    case StrLit(v:String)
    case JsonList(vs:List[Json])
    case JsonObject(flds:Map[String,Json])
}
```

# Scala Map data type

- `Map[K,V]` is a predefined data type
- Like java `Map` and Python dictionary

```
val m:Map[String, Int] = Map(
    "key1" -> 1,
    "key2" -> 10,
)
val m2 = m + ("key3" -> 11)
val ov:Option[Int] = m2.get("key2") // Some(10)
```

# Parsing

```
enum Json {
    case IntLit(v:Int)
    case StrLit(v:String)
    case JsonList(vs:List[Json])
    case JsonObject(flds:Map[String,Json])
}

val input = List(LBRace,SQuote,StrTok("k1"),SQuote,Colon,IntTok(1),Comma,SQuote,
StrTok("k2"),SQuote,Colon,LBracket, RBracket,RBrace)

val expected = Some(JsonObject(
    Map(
        "k1" -> IntLit(1),
        "k2" -> JsonList(Nil)
    )
))

def parse(tokens:List[LToken]):Option[Json] =  ???
```

# Parsing

$$
\begin{array}{rrcl}
\text{(JSON)} & J & ::= & i \mid \text{'}s\text{'} \mid [] \mid [IS] \mid \{NS\} \\
\text{(Items)} & IS & ::= & J, IS \mid J \\
\text{(Named Objects)} & NS & ::= & N, NS \mid N \\
\text{(Named Object)} & N & ::= & \text{'}s\text{'} : J
\end{array}
$$

```
def parse(toks:List[LToken]):Option[Json] = toks match {
    case Nil => // Done? what to return?
    case (t::ts) if t is digit => {
        val i = parse_an_int(toks); Some(IntLit(i)) }
    case (t::ts) if t is '\'' => {
        val s = parse_a_str(toks); Some(StrLit(s)) }
    case (t::ts) if t is '[' => {
        val l = parse_a_list(toks); Some(JsonList(l)) }
    case (t::ts) => {
        val m = parse_a_map(toks); Some(JsonObject(m)) }
} // Can we always decide which path to go by checking t?
```

# Quick summary

- Compiler pipeline
- Lexing
- Parsing
- Top-down Parsing