

Compilers and Formal Languages

Email: christian.urban at kcl.ac.uk

Office Hour: Fridays 11:30 – 12:30

Location: N7.07 (North Wing, Bush House)

Slides & Progs: KEATS

1 Introduction, Languages	6 While-Language
2 Regular Expressions, Derivatives	7 Compilation, JVM
3 Automata, Regular Languages	8 Compiling Functional Languages
4 Lexing, Tokenising	9 Optimisations
5 Grammars, Parsing	10 LLVM

Why the hell did we bother for 4 long weeks with a worse-than-cubic lexing algorithm for regular expressions, given the CYK-algorithm can decide in cubic time whether a string is matched by a context free grammar? Is the lecturer insane?

- Yes, if we are only after string matching. But we are after lexing where we need POSIX matching and calculate POSIX values. You cannot do this with any of the CFG-algorithms.
- Also it is not so easy to include extended regular expressions like $\sim r$ and $r^{\{n\}}$.

Why the hell did we bother for 4 long weeks with a lexing algorithm generating tokens, given the lecturer explains the parsing algorithm using strings as input? Has the lecturer gone insane?

Parser Combinators

One of the simplest ways to implement a parser, see <https://vimeo.com/142341803> (by Haoyi Li)

- built-in library in Scala
- fastparse (2) library by Haoyi Li; is part of Ammonite
- possible exponential runtime behaviour

Palindromes

A grammar for palindromes over the alphabet $\{a, b\}$:

$$S ::= a \cdot S \cdot a \mid b \cdot S \cdot b \mid a \mid b \mid \epsilon$$

Parser Combinators

Parser combinators:

$\underbrace{\text{list of tokens}}_{\text{input}} \Rightarrow \underbrace{\text{set of (parsed input, unparsed input)}}_{\text{output}}$

- atomic parsers
- sequencing
- alternative
- semantic action (map-parser)

Abstract Parser Class

```
abstract class Parser[I, T] {  
  def parse(ts: I): Set[(T, I)]  
  
  def parse_all(ts: I) : Set[T] =  
    for ((head, tail) <- parse(ts);  
        if (tail.isEmpty)) yield head  
}
```

Atomic parsers, for example, number tokens

$$\text{Num}(123) :: \text{rest} \Rightarrow \{(\text{Num}(123), \text{rest})\}$$

- you consume one or more token from the input (stream)
- also works for characters and strings

Alternative parser (code $p \parallel q$)

- apply p and also q ; then combine the outputs

$$p(\text{input}) \cup q(\text{input})$$

Sequence parser (code $p \sim q$)

- apply first p producing a set of pairs
- then apply q to the unparsed parts
- then combine the results:

$((\text{output}_1, \text{output}_2), \text{unparsed part})$

$$\{ ((o_1, o_2), u_2) \mid \\ (o_1, u_1) \in p(\text{input}) \wedge \\ (o_2, u_2) \in q(u_1) \}$$

Map-parser (code $p.map(f)$)

- apply p producing a set of pairs
- then apply the function f to each first component

$$\{(f(o_1), u_1) \mid (o_1, u_1) \in p(\text{input})\}$$

Map-parser (code $p.map(f)$)

- apply p producing a set of pairs
- then apply the function f to each first component

$$\{(f(o_1), u_1) \mid (o_1, u_1) \in p(\text{input})\}$$

f is the semantic action (“what to do with the parsed input”)

Semantic Actions

Addition

$$T \sim + \sim E \Rightarrow \underbrace{f((x, y), z) \Rightarrow x + z}_{\text{semantic action}}$$

Semantic Actions

Addition

$$T \sim + \sim E \Rightarrow \underbrace{f((x, y), z) \Rightarrow x + z}_{\text{semantic action}}$$

Multiplication

$$F \sim * \sim T \Rightarrow f((x, y), z) \Rightarrow x * z$$

Semantic Actions

Addition

$$T \sim + \sim E \Rightarrow \underbrace{f((x, y), z) \Rightarrow x + z}_{\text{semantic action}}$$

Multiplication

$$F \sim * \sim T \Rightarrow f((x, y), z) \Rightarrow x * z$$

Parenthesis

$$(\sim E \sim) \Rightarrow f((x, y), z) \Rightarrow y$$

Input Types of Parsers

- input: token list
- output: set of (output_type, token list)

Input Types of Parsers

- input: **token list**
- output: set of (output_type, **token list**)

actually it can be any input type as long as it is a kind of sequence (for example a string)

Scannerless Parsers

- input: **string**
- output: set of (output_type, **string**)

but using lexers is better because whitespaces or comments can be filtered out; then input is a sequence of tokens

Abstract Parser Class

```
abstract class Parser[I, T] {  
  def parse(ts: I): Set[(T, I)]  
  
  def parse_all(ts: I) : Set[T] =  
    for ((head, tail) <- parse(ts);  
        if (tail.isEmpty)) yield head  
}
```

Successful Parses

- input: string
- output: **set of** (output_type, string)

a parse is successful whenever the input has been fully “consumed” (that is the second component is empty)

```
start := 1000;  
x := start;  
y := start;  
z := start;  
while 0 < x do {  
  while 0 < y do {  
    while 0 < z do { z := z - 1 };  
    z := start;  
    y := y - 1  
  };  
  y := start;  
  x := x - 1  
}
```

While-Language

$Stmt ::= skip$
| $Id := AExp$
| if $BExp$ then $Block$ else $Block$
| while $BExp$ do $Block$

$Stmts ::= Stmt ; Stmts$
| $Stmt$

$Block ::= \{ Stmts \}$
| $Stmt$

$AExp ::= \dots$

$BExp ::= \dots$

Aexprs

$$\begin{aligned}\text{eval}(n) &\stackrel{\text{def}}{=} n \\ \text{eval}(a_1 + a_2) &\stackrel{\text{def}}{=} \text{eval}(a_1) + \text{eval}(a_2) \\ \text{eval}(a_1 - a_2) &\stackrel{\text{def}}{=} \text{eval}(a_1) - \text{eval}(a_2) \\ \text{eval}(a_1 * a_2) &\stackrel{\text{def}}{=} \text{eval}(a_1) * \text{eval}(a_2) \\ \text{eval}(x) &\stackrel{\text{def}}{=} ???\end{aligned}$$

Interpreter

$$\text{eval}(n, E) \stackrel{\text{def}}{=} n$$

$$\text{eval}(x, E) \stackrel{\text{def}}{=} E(x) \quad \text{lookup } x \text{ in } E$$

$$\text{eval}(a_1 + a_2, E) \stackrel{\text{def}}{=} \text{eval}(a_1, E) + \text{eval}(a_2, E)$$

$$\text{eval}(a_1 - a_2, E) \stackrel{\text{def}}{=} \text{eval}(a_1, E) - \text{eval}(a_2, E)$$

$$\text{eval}(a_1 * a_2, E) \stackrel{\text{def}}{=} \text{eval}(a_1, E) * \text{eval}(a_2, E)$$

$$\text{eval}(a_1 = a_2, E) \stackrel{\text{def}}{=} \text{eval}(a_1, E) = \text{eval}(a_2, E)$$

$$\text{eval}(a_1 \neq a_2, E) \stackrel{\text{def}}{=} \neg(\text{eval}(a_1, E) = \text{eval}(a_2, E))$$

$$\text{eval}(a_1 < a_2, E) \stackrel{\text{def}}{=} \text{eval}(a_1, E) < \text{eval}(a_2, E)$$

An Interpreter (1)

```
{  
  x := 5;  
  y := x * 3;  
  y := x * 4;  
  x := u * 3  
}
```

- the interpreter has to record the value of x before assigning a value to y

An Interpreter (1)

```
{  
  x := 5;  
  y := x * 3;  
  y := x * 4;  
  x := u * 3  
}
```

- the interpreter has to record the value of x before assigning a value to y
- `eval(stmt, env)`

Interpreter (2)

$$\text{eval}(\text{skip}, E) \stackrel{\text{def}}{=} E$$

$$\text{eval}(x := a, E) \stackrel{\text{def}}{=} E(x \mapsto \text{eval}(a, E))$$

$$\begin{aligned} \text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) &\stackrel{\text{def}}{=} \\ &\quad \text{if } \text{eval}(b, E) \text{ then } \text{eval}(cs_1, E) \\ &\quad \text{else } \text{eval}(cs_2, E) \end{aligned}$$

$$\begin{aligned} \text{eval}(\text{while } b \text{ do } cs, E) &\stackrel{\text{def}}{=} \\ &\quad \text{if } \text{eval}(b, E) \\ &\quad \text{then } \text{eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) \\ &\quad \text{else } E \end{aligned}$$

$$\text{eval}(\text{write } x, E) \stackrel{\text{def}}{=} \{ \text{println}(E(x)) ; E \}$$

Interpreted Code



