Automata and Formal Languages (8)

Email:	christian.urban at kcl.ac.uk
Office:	S1.27 (1st floor Strand Building)
Slides:	KEATS (also home work is there)

Building a “Web Browser”

Using a lexer: assume the following regular expressions

$$
\begin{aligned}
SYM &\stackrel{\text{def}}{=} (\text{a..zA..Z0..9..}) \\
WORD &\stackrel{\text{def}}{=} SYM^{+} \\
BTAG &\stackrel{\text{def}}{=} < WORD > \\
ETAG &\stackrel{\text{def}}{=} < / WORD > \\
WHITE &\stackrel{\text{def}}{=} "\ \ " + "/n"
\end{aligned}
$$

Interpreting a List of Tokens

- the text should be formatted consistently up to a specified width, say 60 characters
- potential linebreaks are inserted by the formatter (not the input)
- repeated whitespaces are “condensed” to a single whitepace
- $<p>$ $</p>$ start/end paragraph
- $$ $$ start/end bold
- $<red>$ $</red>$ start/end red (cyan, etc)

Interpreting a List of Tokens

The lexer cannot prevent errors like

$$ \ldots <p> \ldots \ldots </p>$$

or

$$ \ldots <b>$$

Parser Combinators

Parser combinators:

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

- sequencing
- alternative
- semantic action

Alternative parser (code $p \mid\mid q$)

- 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:
  ((output$_1$, output$_2$), unparsed part)

$$\begin{array}{l}\{((o_1, o_2), u_2) \mid \\ \qquad (o_1, u_1) \in p(\text{input}) \wedge \\ \qquad (o_2, u_2) \in q(u_1)\}\end{array}$$

Function parser (code $p \Longrightarrow f$)

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

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

Function parser (code $p \Longrightarrow f$)

- 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")

Token parser:

- if the input is

$$tok_1 :: tok_2 :: \ldots :: tok_n$$

then return

$$\{(tok_1,\ tok_2 :: \ldots :: tok_n)\}$$

or

$$\{\}$$

if tok_1 is not the right token we are looking for

Number-Token parser:

- if the input is

$$num\_tok(42) :: tok_2 :: \ldots :: tok_n$$

then return

$$\{(42,\ tok_2 :: \ldots :: tok_n)\}$$

or

$$\{\}$$

if tok_1 is not the right token we are looking for

Number-Token parser:

- if the input is

$$num\_tok(42) :: tok_2 :: \ldots :: tok_n$$

then return

$$\{(42,\ tok_2 :: \ldots :: tok_n)\}$$

or

$$\{\}$$

if tok_1 is not the right token we are looking for

list of tokens $\Rightarrow$ set of (int, list of tokens)

- if the input is

$$num\_tok(42) :: num\_tok(3) :: tok_3 :: \ldots :: tok_n$$

and the parser is

$$ntp \sim ntp$$

the successful output will be

$$\{((42, 3),\ tok_2 :: \ldots :: tok_n)\}$$

- if the input is

$$num\_tok(42) :: \\ num\_tok(3) :: \\ tok_3 :: \ldots :: tok_n$$

and the parser is

$$ntp \sim ntp$$

the successful output will be

$$\{((42, 3),\ tok_2 :: \ldots :: tok_n)\}$$

Now we can form

$$(ntp \sim ntp) \Longrightarrow f$$

where f is the semantic action (“what to do with the pair”)

Semantic Actions

Addition

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

Semantic Actions

Addition

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

Multiplication

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

Semantic Actions

Addition

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

Multiplication

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

Parenthesis

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

Types of Parsers

- **Sequencing**: if p returns results of type T, and q results of type S, then $p \sim q$ returns results of type

$$T \times S$$

Types of Parsers

- **Sequencing**: if p returns results of type T, and q results of type S, then $p \sim q$ returns results of type

$$T \times S$$

- **Alternative**: if p returns results of type T then q must also have results of type T, and $p \mid\mid q$ returns results of type

$$T$$

Types of Parsers

- **Sequencing**: if p returns results of type T, and q results of type S, then $p \sim q$ returns results of type

$$T \times S$$

- **Alternative**: if p returns results of type T then q must also have results of type T, and $p \;||\; q$ returns results of type

$$T$$

- **Semantic Action**: if p returns results of type T and f is a function from T to S, then $p \Longrightarrow f$ returns results of type

$$S$$

Input Types of Parsers

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

Input Types of Parsers

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

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 lexers are better when whitespaces or comments need to be filtered out

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)

```
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

  def || (right : => Parser[I, T]) : Parser[I, T] =
      new AltParser(this, right)
  def ==>[S] (f: => T => S) : Parser [I, S] =
      new FunParser(this, f)
   def ~[S] (right : => Parser[I, S]) : Parser[I, (T, S)]
      new SeqParser(this, right)
}
```

```
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

  def || (right : => Parser[I, T]) : Parser[I, T] =
      new AltParser(this, right)
  def ==>[S] (f: => T => S) : Parser [I, S] =
      new FunParser(this, f)
   def ~[S] (right : => Parser[I, S]) : Parser[I, (T, S)] =
      new SeqParser(this, right)
}
```

```
class SeqParser[I, T, S](p: => Parser[I, T],
                         q: => Parser[I, S])
                             extends Parser[I, (T, S)] {
  def parse(sb: I) =
    for ((head1, tail1) <- p.parse(sb);
         (head2, tail2) <- q.parse(tail1))
            yield ((head1, head2), tail2)
}

class AltParser[I, T](p: => Parser[I, T],
                      q: => Parser[I, T])
                          extends Parser[I, T] {
  def parse(sb: I) = p.parse(sb) ++ q.parse(sb)
}

class FunParser[I, T, S](p: => Parser[I, T], f: T => S)
  extends Parser[I, S] {
  def parse(sb: I) =
    for ((head, tail) <- p.parse(sb))
      yield (f(head), tail)
}
```

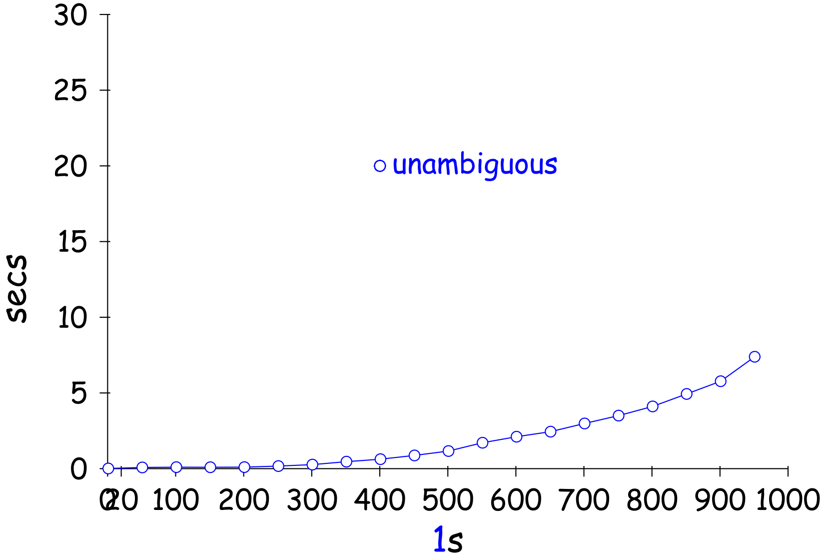
Two Grammars

Which languages are recognised by the following two grammars?

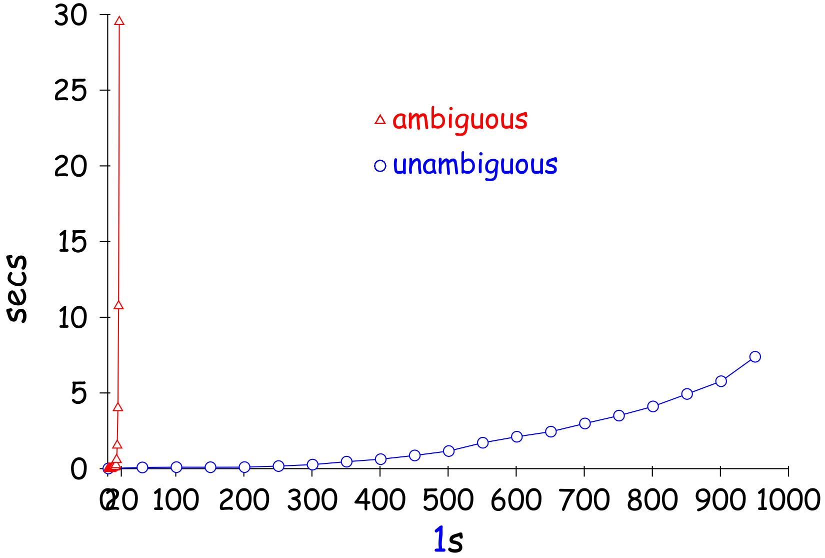
$$\begin{array}{rcl} S & \rightarrow & 1 \cdot S \cdot S \\ & \mid & \epsilon \end{array}$$

$$\begin{array}{rcl} U & \rightarrow & 1 \cdot U \\ & \mid & \epsilon \end{array}$$

Ambiguous Grammars



Ambiguous Grammars



What about Left-Recursion?

- we record when we recursively called a parser
- whenever the is a recursion, the parser must have consumed something — so we can decrease the input string/list of token by one (at the end)

While-Language

$$
\begin{array}{lcl}
Stmt & \rightarrow & \text{skip} \\
 & | & Id := AExp \\
 & | & \text{if } BExp \text{ then } Block \text{ else } Block \\
 & | & \text{while } BExp \text{ do } Block \\
Stmt & \rightarrow & Stmt\ ;\ Stmts \\
 & | & Stmt \\
Block & \rightarrow & \{Stmts\} \\
 & | & Stmt \\
AExp & \rightarrow & \ldots \\
BExp & \rightarrow & \ldots
\end{array}
$$

An Interpreter

$$\begin{array}{l} \{ \\ \quad x := 5; \\ \quad y := x * 3; \\ \quad y := x * 4; \\ \quad x := u * 3 \\ \} \end{array}$$

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

An Interpreter

$$
\begin{array}{l}
\{ \\
\quad x := 5; \\
\quad y := x * 3; \\
\quad y := x * 4; \\
\quad x := u * 3 \\
\}
\end{array}
$$

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