

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

SYM $\stackrel{\text{def}}{=}$ (a..zA..Z0..9..)

WORD $\stackrel{\text{def}}{=}$ *SYM*⁺

BTAG $\stackrel{\text{def}}{=}$ < *WORD* >

ETAG $\stackrel{\text{def}}{=}$ < /*WORD* >

WHITE $\stackrel{\text{def}}{=}$ " " + "/n"

Interpreting a List of Token

- text should be formatted consistently up to a specified width, say 60 characters
- potential linebreaks are inserted by the formatter
- repeated whitespaces are "condensed" to a single whitepace
- *< p >* *< /p >* start/end paragraph
- *< b >* *< /b >* start/end bold
- *< red >* *< /red >* start/end red (cyan, etc)

Interpreting a List of Token

The lexer cannot prevent errors like

$\langle b \rangle \dots \langle p \rangle \dots \langle /b \rangle \dots \langle /p \rangle$

or

$\langle /b \rangle \dots \langle b \rangle$

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 $p \parallel q$)

- apply p and also q ; then combine the outputs

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

Sequence parser (code $p \sim q$)

- apply p first producing a set of pairs
- then apply q to the unparsed parts
- the combine the results:
((input₁, input₂), unparsed part)

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

Function parser (code $p \implies f$)

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

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

Token parser:

- if the input is

$tok_1 :: tok_2 :: \dots :: tok_n$

then return

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

or

$\{\}$

if tok_1 is not the right token we are looking for

Number-Token parser:

- if the input is

$num_tok(42) :: tok_2 :: \dots :: tok_n$

then return

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

or

$\{\}$

if tok_1 is not the right token we are looking for

Number-Token parser:

- if the input is

$num_tok(42) :: tok_2 :: \dots :: tok_n$

then return

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

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

$$\begin{aligned} & num_tok(42) :: \\ & \quad num_tok(3) :: \\ & \quad \quad tok_3 :: \dots :: tok_n \end{aligned}$$

and the parser is

$$ntp \sim ntp$$

the successful output will be

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

- if the input is

$$\begin{aligned} & \mathit{num_tok}(42) :: \\ & \quad \mathit{num_tok}(3) :: \\ & \quad \quad \mathit{tok}_3 :: \dots :: \mathit{tok}_n \end{aligned}$$

and the parser is

$$\mathit{ntp} \sim \mathit{ntp}$$

the successful output will be

$$\{((42, 3), \mathit{tok}_2 :: \dots :: \mathit{tok}_n)\}$$

Now we can form

$$(\mathit{ntp} \sim \mathit{ntp}) \implies f$$

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

Semantic Actions

Addition

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

Semantic Actions

Addition

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

Multiplication

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

Semantic Actions

Addition

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

Multiplication

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

Parenthesis

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

Types of Parsers

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

$$T$$

Types of Parsers

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

$$T$$

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

```
1 abstract class Parser[I, T] {
2     def parse(ts: I): Set[(T, I)]
3
4     def parse_all(ts: I) : Set[T] =
5         for ((head, tail) <- parse(ts); if (tail.isEmpty))
6             yield head
7
8     def || (right : => Parser[I, T]) : Parser[I, T] =
9         new AltParser(this, right)
10    def ==>[S] (f: => T => S) : Parser [I, S] =
11        new FunParser(this, f)
12 }
```

```
1 abstract class Parser[I, T] {
2     def parse(ts: I): Set[(T, I)]
3
4     def parse_all(ts: I) : Set[T] =
5         for ((head, tail) <- parse(ts); if (tail.isEmpty))
6             yield head
7
8     def || (right : => Parser[I, T]) : Parser[I, T] =
9         new AltParser(this, right)
10    def ==>[S] (f: => T => S) : Parser [I, S] =
11        new FunParser(this, f)
12 }
```

```
1 class SeqParser[I, T, S] (p: => Parser[I, T],
2                          q: => Parser[I, S])
3                          extends Parser[I, (T, S)] {
4   def parse(sb: I) =
5     for ((head1, tail1) <- p.parse(sb);
6         (head2, tail2) <- q.parse(tail1))
7       yield ((head1, head2), tail2)
8 }
9
10 class AltParser[I, T] (p: => Parser[I, T],
11                       q: => Parser[I, T])
12                       extends Parser[I, T] {
13   def parse(sb: I) = p.parse(sb) ++ q.parse(sb)
14 }
15
16 class FunParser[I, T, S] (p: => Parser[I, T], f: T => S)
17   extends Parser[I, S] {
18   def parse(sb: I) =
19     for ((head, tail) <- p.parse(sb))
20       yield (f(head), tail)
21 }
```

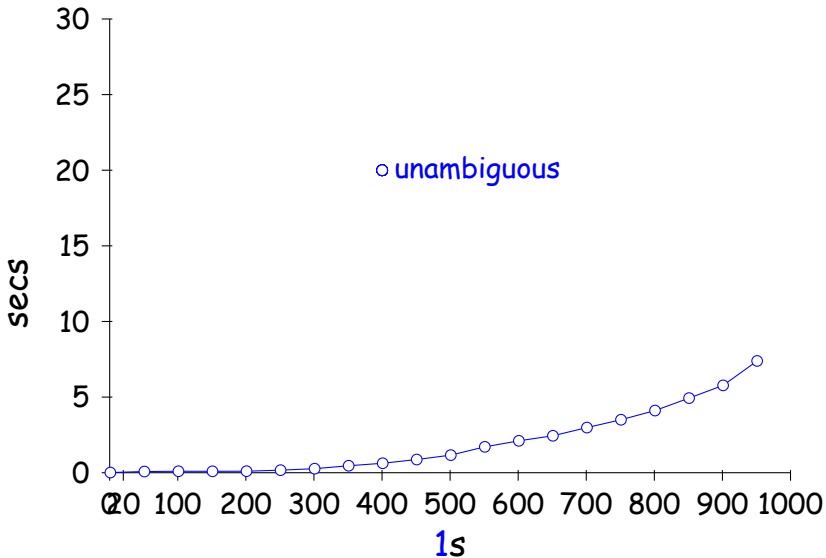

Two Grammars

Which languages are recognised by the following two grammars?

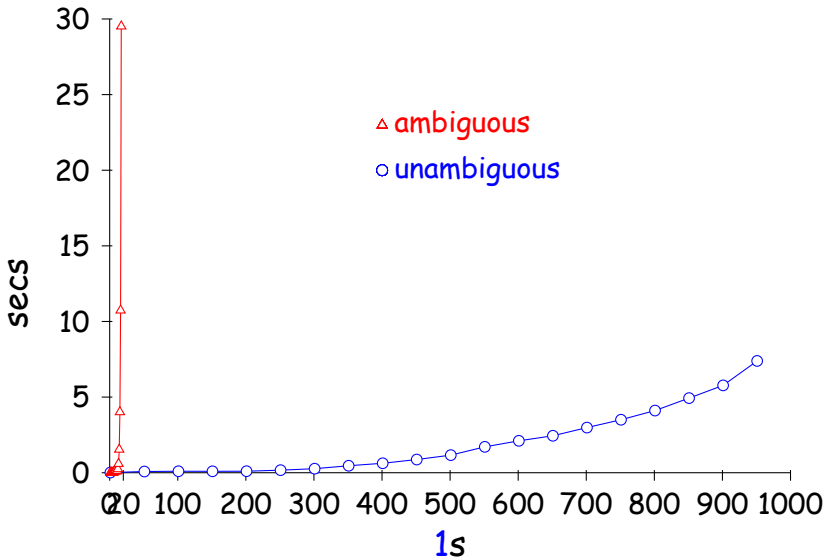
$$\begin{array}{ccc} S & \rightarrow & 1 \cdot S \cdot S \\ & & | \quad \epsilon \end{array}$$

$$\begin{array}{ccc} U & \rightarrow & 1 \cdot U \\ & & | \quad \epsilon \end{array}$$

Ambiguous Grammars



Ambiguous Grammars



What about Left-Recursion?

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

While-Language

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

Stmt → *Stmt* ; *Stmts*
| *Stmt*

Block → {*Stmts*}
| *Stmt*

An Interpreter

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

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

An Interpreter

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

- the interpreter has to record the value of x before assigning a value to y
- $\text{eval}(\text{stmt}, \text{env})$