

Automata and Formal Languages (5)

Email: christian.urban at kcl.ac.uk

Office: SI.27 (1st floor Strand Building)

Slides: KEATS (also home work is there)

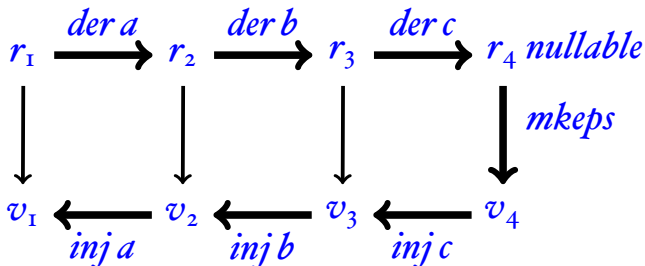
Last Week

Regexes and Values

Regular expressions and their corresponding values:

$r ::=$	$v ::=$
\emptyset	$Empty$
ϵ	$Char(c)$
c	$Seq(v_1, v_2)$
$r_1 \cdot r_2$	$Left(v)$
$r_1 + r_2$	$Right(v)$
r^*	$[v_1, \dots, v_n]$

$r_1: a \cdot (b \cdot c)$
 $r_2: \epsilon \cdot (b \cdot c)$
 $r_3: (\emptyset \cdot (b \cdot c)) + (\epsilon \cdot c)$
 $r_4: (\emptyset \cdot (b \cdot c)) + ((\emptyset \cdot c) + \epsilon)$

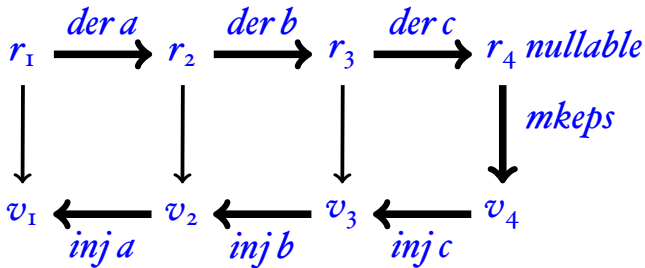


$v_1: \text{Seq}(\text{Char}(a), \text{Seq}(\text{Char}(b), \text{Char}(c)))$
 $v_2: \text{Seq}(\text{Empty}, \text{Seq}(\text{Char}(b), \text{Char}(c)))$
 $v_3: \text{Right}(\text{Seq}(\text{Empty}, \text{Char}(c)))$
 $v_4: \text{Right}(\text{Right}(\text{Empty}))$

$|v_1|: abc$
 $|v_2|: bc$
 $|v_3|: c$
 $|v_4|: []$

Simplification

- If we simplify after the derivative, then we are building the value for the simplified regular expression, but *not* for the original regular expression.



$$(\emptyset \cdot (b \cdot c)) + ((\emptyset \cdot c) + \epsilon) \mapsto \epsilon$$

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$
- $nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$
- $der\ c\ (x : r) \stackrel{\text{def}}{=} (x : der\ c\ r)$
- $mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$
- $inj\ (x : r)\ c\ v \stackrel{\text{def}}{=} Rec(x, inj\ r\ c\ v)$

for extracting subpatterns $(z : ((x : ab) + (y : ba)))$

Environments

Obtaining the “recorded” parts of a value:

$$\begin{aligned} env(Empty) &\stackrel{\text{def}}{=} [] \\ env(Char(c)) &\stackrel{\text{def}}{=} [] \\ env(Left(v)) &\stackrel{\text{def}}{=} env(v) \\ env(Right(v)) &\stackrel{\text{def}}{=} env(v) \\ env(Seq(v_1, v_2)) &\stackrel{\text{def}}{=} env(v_1) @ env(v_2) \\ env([v_1, \dots, v_n]) &\stackrel{\text{def}}{=} env(v_1) @ \dots @ env(v_n) \\ env(Rec(x : v)) &\stackrel{\text{def}}{=} (x : |v|) :: env(v) \end{aligned}$$

While Tokens

WHILE_REGS $\stackrel{\text{def}}{=}$ ((**"k"** : KEYWORD) +
(**"i"** : ID) +
(**"o"** : OP) +
(**"n"** : NUM) +
(**"s"** : SEMI) +
(**"p"** : (LPAREN + RPAREN)) +
(**"b"** : (BEGIN + END)) +
(**"w"** : WHITESPACE))*

”if true then then 42 else +”

KEYWORD(if),
WHITESPACE,
IDENT(true),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
NUM(42),
WHITESPACE,
KEYWORD(else),
WHITESPACE,
OP(+)

”if true then then 42 else +”

KEYWORD(if),
IDENT(true),
KEYWORD(then),
KEYWORD(then),
NUM(42),
KEYWORD(else),
OP(+)

Two Rules

- Longest match rule (“maximal munch rule”): The longest initial substring matched by any regular expression is taken as next token.
- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

- Regular expression for email addresses

(name: $[a-z0-9_.-]^+$).@.
(domain: $[a-z0-9.-]^+$) ..
(top_level: $[a-z.]\{2,6\}$)

christian.urban@kcl.ac.uk

- result environment:

$[(name : christian.urban),$
 $(domain : kcl),$
 $(top_level : ac.uk)]$

Coursework

$$\text{nullable}([c_1 c_2 \dots c_n]) \stackrel{\text{def}}{=} ?$$

$$\text{nullable}(r^+) \stackrel{\text{def}}{=} ?$$

$$\text{nullable}(r^?) \stackrel{\text{def}}{=} ?$$

$$\text{nullable}(r^{\{n,m\}}) \stackrel{\text{def}}{=} ?$$

$$\text{nullable}(\sim r) \stackrel{\text{def}}{=} ?$$

$$\text{der } c ([c_1 c_2 \dots c_n]) \stackrel{\text{def}}{=} ?$$

$$\text{der } c (r^+) \stackrel{\text{def}}{=} ?$$

$$\text{der } c (r^?) \stackrel{\text{def}}{=} ?$$

$$\text{der } c (r^{\{n,m\}}) \stackrel{\text{def}}{=} ?$$

$$\text{der } c (\sim r) \stackrel{\text{def}}{=} ?$$

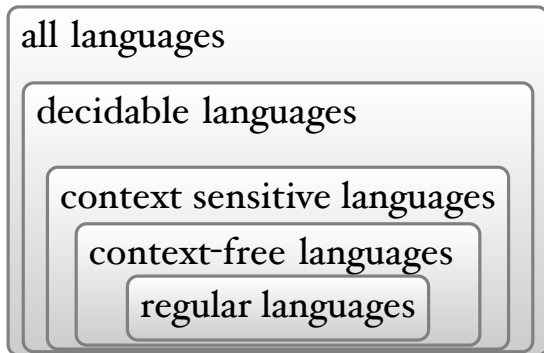
Regular Languages

While regular expressions are very useful for lexing, there is no regular expression that can recognise the language $a^n b^n$.

$((((()))))$ vs. $((((())) ()))$

So we cannot find out with regular expressions whether parentheses are matched or unmatched. Also regular expressions are not recursive, e.g. $(1 + 2) + 3$.

Hierarchy of Languages



Grammars

A (context-free) grammar G consists of

- a finite set of nonterminal symbols (upper case)
- a finite terminal symbols or tokens (lower case)
- a start symbol (which must be a nonterminal)
- a set of rules

$$A \rightarrow rhs$$

where rhs are sequences involving terminals and nonterminals, including the empty sequence ϵ .

Grammars

A (context-free) grammar G consists of

- a finite set of nonterminal symbols (upper case)
- a finite terminal symbols or tokens (lower case)
- a start symbol (which must be a nonterminal)
- a set of rules

$$A \rightarrow rhs$$

where rhs are sequences involving terminals and nonterminals, including the empty sequence ϵ .

We also allow rules

$$A \rightarrow rhs_1 | rhs_2 | \dots$$

Palindromes

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

$$S \rightarrow \epsilon$$

$$S \rightarrow a \cdot S \cdot a$$

$$S \rightarrow b \cdot S \cdot b$$

Palindromes

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

$$S \rightarrow \epsilon$$

$$S \rightarrow a \cdot S \cdot a$$

$$S \rightarrow b \cdot S \cdot b$$

or

$$S \rightarrow \epsilon \mid a \cdot S \cdot a \mid b \cdot S \cdot b$$

Palindromes

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

$$S \rightarrow \epsilon$$

$$S \rightarrow a \cdot S \cdot a$$

$$S \rightarrow b \cdot S \cdot b$$

or

$$S \rightarrow \epsilon \mid a \cdot S \cdot a \mid b \cdot S \cdot b$$

Can you find the grammar rules for matched parentheses?

Arithmetic Expressions

$$E \rightarrow \textit{num_token}$$

$$E \rightarrow E \cdot + \cdot E$$

$$E \rightarrow E \cdot - \cdot E$$

$$E \rightarrow E \cdot * \cdot E$$

$$E \rightarrow (\cdot E \cdot)$$

Arithmetic Expressions

$$E \rightarrow \textit{num_token}$$

$$E \rightarrow E \cdot + \cdot E$$

$$E \rightarrow E \cdot - \cdot E$$

$$E \rightarrow E \cdot * \cdot E$$

$$E \rightarrow (\cdot E \cdot)$$

1 + 2 * 3 + 4

A CFG Derivation

- 1 Begin with a string containing only the start symbol, say S
- 2 Replace any nonterminal X in the string by the right-hand side of some production $X \rightarrow rhs$
- 3 Repeat 2 until there are no nonterminals

$$S \rightarrow \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots$$

Example Derivation

$$S \rightarrow \epsilon \mid a \cdot S \cdot a \mid b \cdot S \cdot b$$

$$\begin{aligned} S &\rightarrow aSa \\ &\rightarrow abSba \\ &\rightarrow abaSaba \\ &\rightarrow abaaba \end{aligned}$$

Example Derivation

$$E \rightarrow \textit{num_token}$$

$$E \rightarrow E \cdot + \cdot E$$

$$E \rightarrow E \cdot - \cdot E$$

$$E \rightarrow E \cdot * \cdot E$$

$$E \rightarrow (\cdot E \cdot)$$

$$E \rightarrow E * E$$

$$\rightarrow E + E * E$$

$$\rightarrow E + E * E + E$$

$$\rightarrow^+ 1 + 2 * 3 + 4$$

Example Derivation

$$E \rightarrow \textit{num_token}$$

$$E \rightarrow E \cdot + \cdot E$$

$$E \rightarrow E \cdot - \cdot E$$

$$E \rightarrow E \cdot * \cdot E$$

$$E \rightarrow (\cdot E \cdot)$$

$$E \rightarrow E * E$$

$$\rightarrow E + E * E$$

$$\rightarrow E + E * E + E$$

$$\rightarrow^+ 1 + 2 * 3 + 4$$

$$E \rightarrow E + E$$

$$\rightarrow E + E + E$$

$$\rightarrow E + E * E + E$$

$$\rightarrow^+ 1 + 2 * 3 + 4$$

Context Sensitive Grms

$$\begin{aligned} S &\Rightarrow bSAA \mid \epsilon \\ A &\Rightarrow a \\ bA &\Rightarrow Ab \end{aligned}$$

Context Sensitive Grms

$$S \Rightarrow bSAA \mid \epsilon$$

$$A \Rightarrow a$$

$$bA \Rightarrow Ab$$

$$S \Rightarrow \dots \Rightarrow? "ababaa"$$

Language of a CFG

Let G be a context-free grammar with start symbol S . Then the language $L(G)$ is:

$$\{c_1 \dots c_n \mid \forall i. c_i \in T \wedge S \rightarrow^* c_1 \dots c_n\}$$

Language of a CFG

Let G be a context-free grammar with start symbol S . Then the language $L(G)$ is:

$$\{c_1 \dots c_n \mid \forall i. c_i \in T \wedge S \rightarrow^* c_1 \dots c_n\}$$

- Terminals, because there are no rules for replacing them.
- Once generated, terminals are “permanent”.
- Terminals ought to be tokens of the language (but can also be strings).

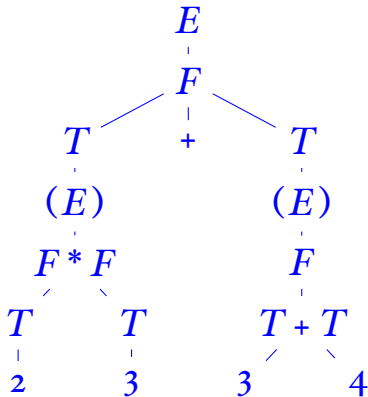
Parse Trees

$$E \rightarrow F \mid F \cdot * \cdot F$$

$$F \rightarrow T \mid T \cdot + \cdot T \mid T \cdot - \cdot T$$

$$T \rightarrow \text{num_token} \mid (\cdot E \cdot)$$

$(2*3)+(3+4)$



Arithmetic Expressions

$$E \rightarrow \textit{num_token}$$

$$E \rightarrow E \cdot + \cdot E$$

$$E \rightarrow E \cdot - \cdot E$$

$$E \rightarrow E \cdot * \cdot E$$

$$E \rightarrow (\cdot E \cdot)$$

Arithmetic Expressions

$$E \rightarrow \textit{num_token}$$

$$E \rightarrow E \cdot + \cdot E$$

$$E \rightarrow E \cdot - \cdot E$$

$$E \rightarrow E \cdot * \cdot E$$

$$E \rightarrow (\cdot E \cdot)$$

A CFG is **left-recursive** if it has a nonterminal E such that $E \rightarrow^+ E \cdot \dots$

Ambiguous Grammars

A grammar is **ambiguous** if there is a string that has at least two different parse trees.

$$E \rightarrow \textit{num_token}$$

$$E \rightarrow E \cdot + \cdot E$$

$$E \rightarrow E \cdot - \cdot E$$

$$E \rightarrow E \cdot * \cdot E$$

$$E \rightarrow (\cdot E \cdot)$$

1 + 2 * 3 + 4

Dangling Else

Another ambiguous grammar:

$$\begin{array}{lcl} E & \rightarrow & \text{if } E \text{ then } E \\ & | & \text{if } E \text{ then } E \text{ else } E \\ & | & \dots \end{array}$$

if a then if x then y else c

Parser Combinators

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

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

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:

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

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

Function parser (code $p \Rightarrow f$)

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

$$\{ (f(o_I), u_I) \mid (o_I, u_I) \in p(\text{input}) \}$$

Function parser (code $p \Rightarrow f$)

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

$$\{ (f(o_I), u_I) \mid (o_I, u_I) \in p(\text{input}) \}$$

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

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

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 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

$$T$$

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 $p \Rightarrow f$ returns results of type

$$S$$

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 lexers are better when whitespaces or comments need to be filtered out; then input is a sequence of tokens

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 Parser Class

```
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);  
6              if (tail.isEmpty)) yield head  
7  }
```

```

1  class AltParser[I, T](p: => Parser[I, T],
2                               q: => Parser[I, T])
3                               extends Parser[I, T] {
4      def parse(sb: I) = p.parse(sb) ++ q.parse(sb)
5  }
6
7  class SeqParser[I, T, S](p: => Parser[I, T],
8                               q: => Parser[I, S])
9                               extends Parser[I, (T, S)] {
10     def parse(sb: I) =
11         for ((head1, tail1) <- p.parse(sb);
12             (head2, tail2) <- q.parse(tail1))
13             yield ((head1, head2), tail2)
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 }

```

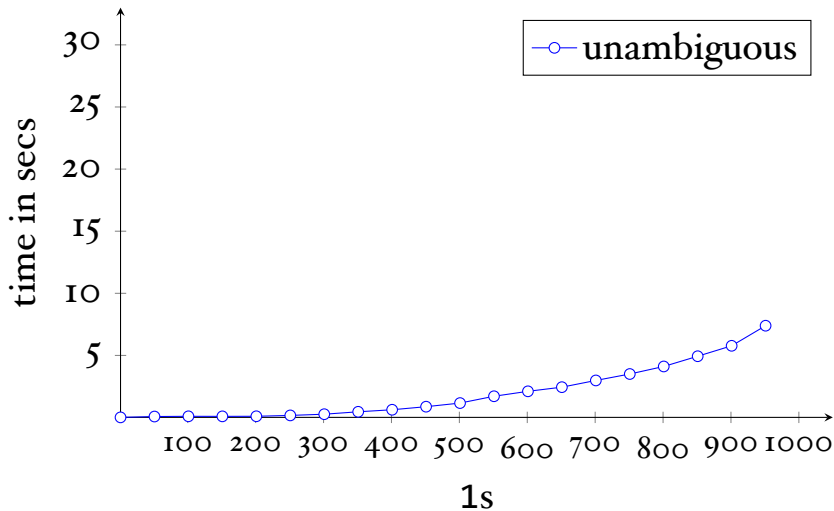
Two Grammars

Which languages are recognised by the following two grammars?

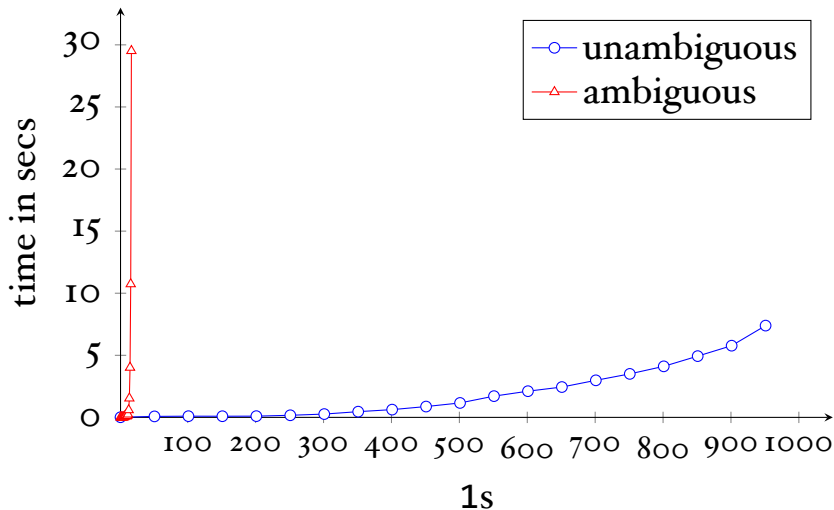
$$\begin{array}{ccc} S & \rightarrow & I \cdot S \cdot S \\ & | & \epsilon \end{array}$$

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

Ambiguous Grammars



Ambiguous Grammars



While-Language

$$\begin{aligned}\langle Stmt \rangle &::= \text{skip} \\ &\quad | \langle Id \rangle := \langle AExp \rangle \\ &\quad | \text{if } \langle BExp \rangle \text{ then } \langle Block \rangle \text{ else } \langle Block \rangle \\ &\quad | \text{while } \langle BExp \rangle \text{ do } \langle Block \rangle\end{aligned}$$
$$\begin{aligned}\langle Stmts \rangle &::= \langle Stmt \rangle ; \langle Stmts \rangle \\ &\quad | \langle Stmt \rangle\end{aligned}$$
$$\begin{aligned}\langle Block \rangle &::= \{ \langle Stmts \rangle \} \\ &\quad | \langle Stmt \rangle\end{aligned}$$
$$\langle AExp \rangle ::= \dots$$
$$\langle BExp \rangle ::= \dots$$

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
- `eval(stmt, env)`