

# Compilers and Formal Languages (5)

Email: christian.urban at kcl.ac.uk

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

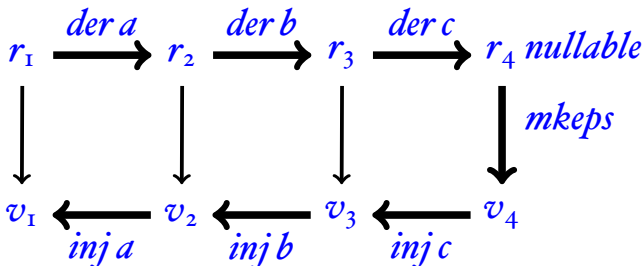
Slides: KEATS (also home work is there)

# Last Week

## Regexes and Values

Regular expressions and their corresponding values:

$r ::=$	<b>0</b>	$v ::=$	<i>Empty</i>
	<b>1</b>		<i>Char(c)</i>
	$c$		<i>Seq(v<sub>1</sub>, v<sub>2</sub>)</i>
	$r_1 \cdot r_2$		<i>Left(v)</i>
	$r_1 + r_2$		<i>Right(v)</i>
	$r^*$		$[v_1, \dots, v_n]$

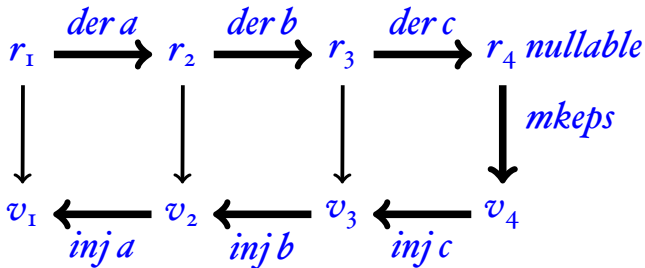
$$\begin{aligned}
 r_1: & a \cdot (b \cdot c) \\
 r_2: & \mathbf{I} \cdot (b \cdot c) \\
 r_3: & (\mathbf{O} \cdot (b \cdot c)) + (\mathbf{I} \cdot c) \\
 r_4: & (\mathbf{O} \cdot (b \cdot c)) + ((\mathbf{O} \cdot c) + \mathbf{I})
 \end{aligned}$$


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

$$\begin{aligned}
 |v_1|: & abc \\
 |v_2|: & bc \\
 |v_3|: & c \\
 |v_4|: & []
 \end{aligned}$$

# 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.



$$(b \cdot c) + (\mathbf{0} + \mathbf{1}) \mapsto (b \cdot c) + \mathbf{1}$$

$$(b \cdot c) + (\mathbf{0} + \mathbf{1}) \mapsto (b \cdot c) + \mathbf{1}$$

$$(\underline{b \cdot c}) + (\underline{\mathbf{0} + \mathbf{I}}) \mapsto (b \cdot c) + \mathbf{I}$$

$$(\underline{b \cdot c}) + (\underline{\mathbf{0} + \mathbf{I}}) \mapsto (b \cdot c) + \mathbf{I}$$

$$f_{s1} = \lambda v.v$$

$$f_{s2} = \lambda v.Right(v)$$

$$\underline{(b \cdot c) + (\mathbf{0} + \mathbf{1})} \mapsto (b \cdot c) + \mathbf{1}$$

$$\begin{aligned} f_{s_1} &= \lambda v. v \\ f_{s_2} &= \lambda v. \mathit{Right}(v) \end{aligned}$$

$$\begin{aligned} f_{alt}(f_{s_1}, f_{s_2}) &\stackrel{\text{def}}{=} \\ &\lambda v. \text{ case } v = \mathit{Left}(v') : \text{ return } \mathit{Left}(f_{s_1}(v')) \\ &\quad \text{ case } v = \mathit{Right}(v') : \text{ return } \mathit{Right}(f_{s_2}(v')) \end{aligned}$$



$$\underline{(b \cdot c) + (\mathbf{0} + \mathbf{I})} \mapsto (b \cdot c) + \mathbf{I}$$

$$\begin{aligned} f_{s1} &= \lambda v.v \\ f_{s2} &= \lambda v.Right(v) \end{aligned}$$

$\lambda v.$  case  $v = Left(v')$ : return  $Left(v')$   
case  $v = Right(v')$ : return  $Right(Right(v'))$

$$\underline{(b \cdot c) + (\mathbf{0} + \mathbf{I})} \mapsto (b \cdot c) + \mathbf{I}$$

$$\begin{aligned} f_{s1} &= \lambda v.v \\ f_{s2} &= \lambda v.Right(v) \end{aligned}$$

$\lambda v.$  case  $v = Left(v')$ : return  $Left(v')$   
case  $v = Right(v')$ : return  $Right(Right(v'))$

*mkeps* simplified case:  $Right(Empty)$   
rectified case:  $Right(Right(Empty))$

# 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:

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

# 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.



# Coursework

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

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

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

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

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

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

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

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

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

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

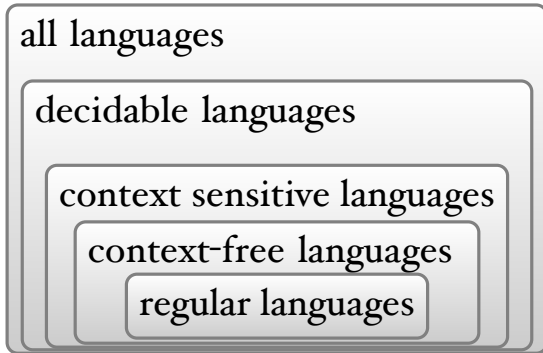
# Regular Languages

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



# CF Grammars

A **context-free grammar**  $G$  consists of

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

$$\langle A \rangle ::= rhs$$

where  $rhs$  are sequences involving terminals and nonterminals, including the empty sequence  $\epsilon$ .

# CF Grammars

A **context-free grammar**  $G$  consists of

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

$$\langle A \rangle ::= rhs$$

where  $rhs$  are sequences involving terminals and nonterminals, including the empty sequence  $\epsilon$ .

We also allow rules

$$\langle A \rangle ::= rhs_1 | rhs_2 | \dots$$

# Palindromes

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

$$\langle S \rangle ::= \epsilon$$

$$\langle S \rangle ::= a \cdot \langle S \rangle \cdot a$$

$$\langle S \rangle ::= b \cdot \langle S \rangle \cdot b$$

# Palindromes

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

$$\langle S \rangle ::= \epsilon$$

$$\langle S \rangle ::= a \cdot \langle S \rangle \cdot a$$

$$\langle S \rangle ::= b \cdot \langle S \rangle \cdot b$$

or

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

# Palindromes

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

$$\langle S \rangle ::= \epsilon$$

$$\langle S \rangle ::= a \cdot \langle S \rangle \cdot a$$

$$\langle S \rangle ::= b \cdot \langle S \rangle \cdot b$$

or

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

Can you find the grammar rules for matched parentheses?



# Arithmetic Expressions

$$\begin{aligned} \langle E \rangle &::= \textit{num\_token} \\ &| \langle E \rangle \cdot + \cdot \langle E \rangle \\ &| \langle E \rangle \cdot - \cdot \langle E \rangle \\ &| \langle E \rangle \cdot * \cdot \langle E \rangle \\ &| (\cdot \langle E \rangle \cdot) \end{aligned}$$

# Arithmetic Expressions

$$\begin{aligned} \langle E \rangle &::= \textit{num\_token} \\ &| \langle E \rangle \cdot + \cdot \langle E \rangle \\ &| \langle E \rangle \cdot - \cdot \langle E \rangle \\ &| \langle E \rangle \cdot * \cdot \langle E \rangle \\ &| (\cdot \langle E \rangle \cdot) \end{aligned}$$

1 + 2 \* 3 + 4

# A CFG Derivation

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

$\langle S \rangle \rightarrow \dots \rightarrow \dots \rightarrow \dots \rightarrow \dots$

# Example Derivation

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

$$\begin{aligned} \langle S \rangle &\rightarrow a \langle S \rangle a \\ &\rightarrow ab \langle S \rangle ba \\ &\rightarrow aba \langle S \rangle aba \\ &\rightarrow abaaba \end{aligned}$$

# Example Derivation

$\langle E \rangle ::= num\_token$

|  $\langle E \rangle \cdot + \cdot \langle E \rangle$

|  $\langle E \rangle \cdot - \cdot \langle E \rangle$

|  $\langle E \rangle \cdot * \cdot \langle E \rangle$

|  $(\cdot \langle E \rangle \cdot)$

$\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$   
 $\rightarrow \langle E \rangle + \langle E \rangle * \langle E \rangle$   
 $\rightarrow \langle E \rangle + \langle E \rangle * \langle E \rangle + \langle E \rangle$   
 $\rightarrow^+ 1 + 2 * 3 + 4$

# Example Derivation

$\langle E \rangle ::= num\_token$

|  $\langle E \rangle \cdot + \cdot \langle E \rangle$

|  $\langle E \rangle \cdot - \cdot \langle E \rangle$

|  $\langle E \rangle \cdot * \cdot \langle E \rangle$

|  $(\cdot \langle E \rangle \cdot)$

$\langle E \rangle \rightarrow \langle E \rangle * \langle E \rangle$   
 $\rightarrow \langle E \rangle + \langle E \rangle * \langle E \rangle$   
 $\rightarrow \langle E \rangle + \langle E \rangle * \langle E \rangle + \langle E \rangle$   
 $\rightarrow^+ 1 + 2 * 3 + 4$

$\langle E \rangle \rightarrow \langle E \rangle + \langle E \rangle$   
 $\rightarrow \langle E \rangle + \langle E \rangle + \langle E \rangle$   
 $\rightarrow \langle E \rangle + \langle E \rangle * \langle E \rangle + \langle E \rangle$   
 $\rightarrow^+ 1 + 2 * 3 + 4$

# Context Sensitive Grammars

It is much harder to find out whether a string is parsed by a context sensitive grammar:

$$\langle S \rangle ::= b\langle S \rangle \langle A \rangle \langle A \rangle \mid \epsilon$$

$$\langle A \rangle ::= a$$

$$b\langle A \rangle ::= \langle A \rangle b$$

# Context Sensitive Grammars

It is much harder to find out whether a string is parsed by a context sensitive grammar:

$$\langle S \rangle ::= b \langle S \rangle \langle A \rangle \langle A \rangle \mid \epsilon$$

$$\langle A \rangle ::= a$$

$$b \langle A \rangle ::= \langle A \rangle b$$

$$\langle S \rangle \rightarrow \dots \rightarrow? ababaa$$



# Language of a CFG

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

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

# Language of a CFG

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

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

- 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).

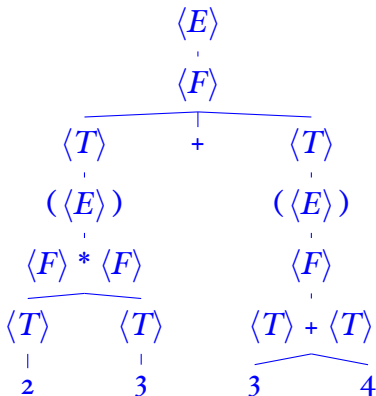
# Parse Trees

$\langle E \rangle ::= \langle F \rangle \mid \langle F \rangle \cdot * \cdot \langle F \rangle$

$\langle F \rangle ::= \langle T \rangle \mid \langle T \rangle \cdot + \cdot \langle T \rangle \mid \langle T \rangle \cdot - \cdot \langle T \rangle$

$\langle T \rangle ::= \text{num\_token} \mid (\cdot \langle E \rangle \cdot)$

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



# Arithmetic Expressions

$\langle E \rangle ::= \textit{num\_token}$

|  $\langle E \rangle \cdot + \cdot \langle E \rangle$

|  $\langle E \rangle \cdot - \cdot \langle E \rangle$

|  $\langle E \rangle \cdot * \cdot \langle E \rangle$

|  $(\cdot \langle E \rangle \cdot)$

# Arithmetic Expressions

$$\begin{aligned}\langle E \rangle &::= \textit{num\_token} \\ &| \langle E \rangle \cdot + \cdot \langle E \rangle \\ &| \langle E \rangle \cdot - \cdot \langle E \rangle \\ &| \langle E \rangle \cdot * \cdot \langle E \rangle \\ &| (\cdot \langle E \rangle \cdot)\end{aligned}$$

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

# Ambiguous Grammars

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

$$\begin{aligned}\langle E \rangle &::= \textit{num\_token} \\ &| \langle E \rangle \cdot + \cdot \langle E \rangle \\ &| \langle E \rangle \cdot - \cdot \langle E \rangle \\ &| \langle E \rangle \cdot * \cdot \langle E \rangle \\ &| (\cdot \langle E \rangle \cdot)\end{aligned}$$

1 + 2 \* 3 + 4

# Dangling Else

Another ambiguous grammar:

$$\begin{array}{l} E \rightarrow \text{if } E \text{ then } E \\ \quad | \text{if } E \text{ then } E \text{ else } E \\ \quad | \dots \end{array}$$

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

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

# Semantic Actions

## Addition

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

## Multiplication

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

# Semantic Actions

## Addition

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

## 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 }

```

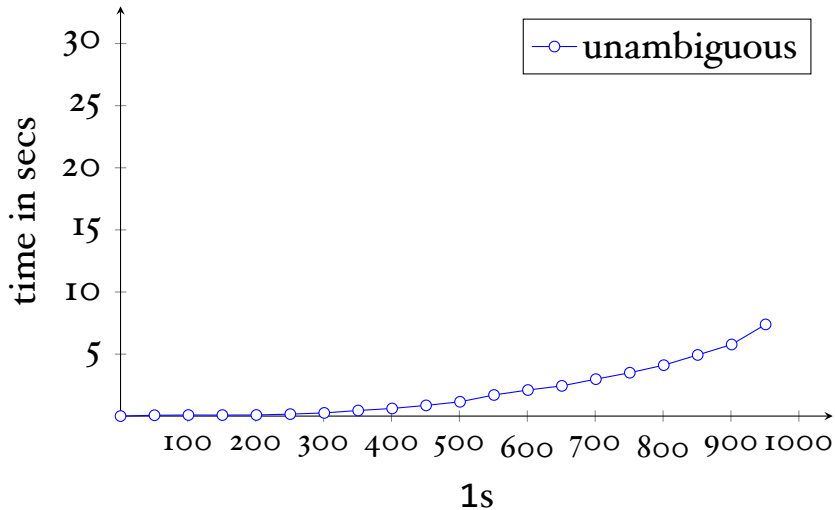
# Two Grammars

Which languages are recognised by the following two grammars?

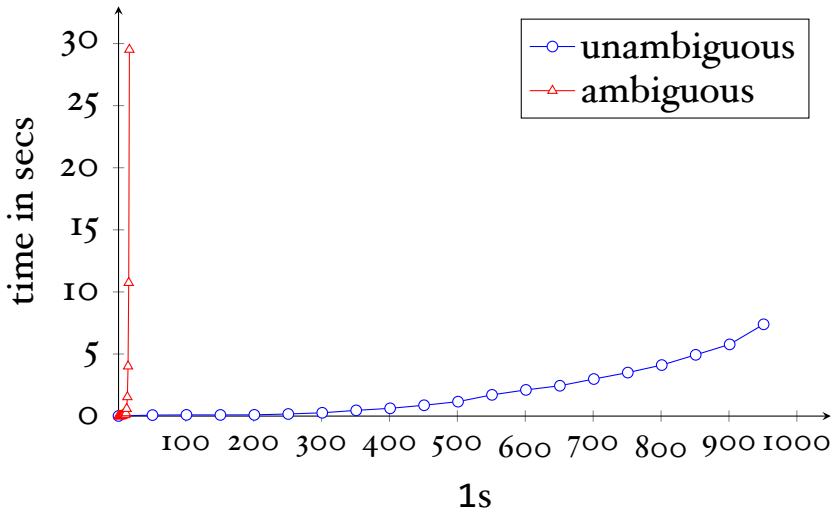
$$S \rightarrow \begin{array}{l} I \cdot S \cdot S \\ | \\ \epsilon \end{array}$$

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

# Ambiguous Grammars



# Ambiguous Grammars



# While-Language

$\langle Stmt \rangle ::= \text{skip}$   
|  $\langle Id \rangle := \langle AExp \rangle$   
| if  $\langle BExp \rangle$  then  $\langle Block \rangle$  else  $\langle Block \rangle$   
| while  $\langle BExp \rangle$  do  $\langle Block \rangle$

$\langle Stmts \rangle ::= \langle Stmt \rangle ; \langle Stmts \rangle$   
|  $\langle Stmt \rangle$

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

$\langle AExp \rangle ::= \dots$

$\langle BExp \rangle ::= \dots$

# 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)`