Compilers and Formal Languages (5)

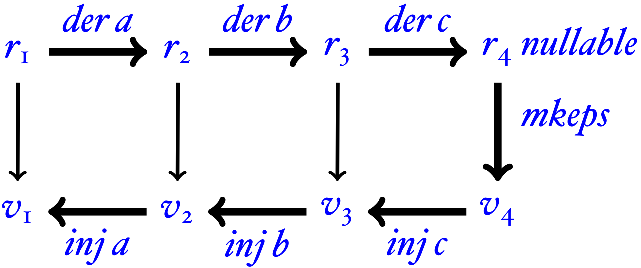
Email: christian.urban at kcl.ac.uk
Office: S1.27 (1st floor Strand Building)
Slides: KEATS (also home work is there)

Last Week
Regexes and Values

Regular expressions and their corresponding values:

$$
\begin{array}{rcl}
r & ::= & \mathbf{0} \\
  & | & \mathbf{1} \\
  & | & c \\
  & | & r_1 \cdot r_2 \\
  & | & r_1 + r_2 \\
  & & \\
  & | & r^*
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \\
  & & \mathit{Empty} \\
  & | & \mathit{Char}(c) \\
  & | & \mathit{Seq}(v_1, v_2) \\
  & | & \mathit{Left}(v) \\
  & | & \mathit{Right}(v) \\
  & | & [v_1, \ldots v_n]
\end{array}
$$

r_1: $a \cdot (b \cdot c)$
r_2: $\mathbf{1} \cdot (b \cdot c)$
r_3: $(\mathbf{0} \cdot (b \cdot c)) + (\mathbf{1} \cdot c)$
r_4: $(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})$

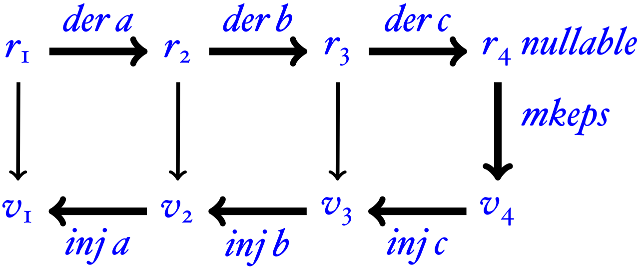


v_1: $Seq(Char(a), Seq(Char(b), Char(c)))$
v_2: $Seq(Empty, Seq(Char(b), Char(c)))$
v_3: $Right(Seq(Empty, Char(c)))$
v_4: $Right(Right(Empty))$

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

Simplification

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



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

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

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

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

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

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

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

$$f_{alt}(f_{s1}, f_{s2}) \stackrel{\text{def}}{=}$$

$$\lambda v.\ \text{case } v = Left(v'):\ \ \text{return } Left(f_{s1}(v'))$$
$$\text{case } v = Right(v'):\ \text{return } Right(f_{s2}(v'))$$

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

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

$$\lambda v.\ \begin{array}{l} \text{case } v = Left(v'): \quad \text{return } Left(v') \\ \text{case } v = Right(v'): \text{return } Right(Right(v')) \end{array}$$

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

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

$$\begin{array}{ll} \lambda v.\ \text{case } v = Left(v'): & \text{return } Left(v') \\ \quad\ \ \text{case } v = Right(v'): & \text{return } Right(Right(v')) \end{array}$$

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:

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

While Tokens

$$
\begin{aligned}
\texttt{WHILE\_REGS} \stackrel{\text{def}}{=} ((&\text{"k"} : \texttt{KEYWORD}) + \\
(&\text{"i"} : \texttt{ID}) + \\
(&\text{"o"} : \texttt{OP}) + \\
(&\text{"n"} : \texttt{NUM}) + \\
(&\text{"s"} : \texttt{SEMI}) + \\
(&\text{"p"} : (\texttt{LPAREN} + \texttt{RPAREN})) + \\
(&\text{"b"} : (\texttt{BEGIN} + \texttt{END})) + \\
(&\text{"w"} : \texttt{WHITESPACE}))^*
\end{aligned}
$$

”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

$$nullable([c_1 c_2 \ldots c_n]) \stackrel{\text{def}}{=} ?$$
$$nullable(r^+) \stackrel{\text{def}}{=} ?$$
$$nullable(r^?) \stackrel{\text{def}}{=} ?$$
$$nullable(r^{\{n,m\}}) \stackrel{\text{def}}{=} ?$$
$$nullable(\sim r) \stackrel{\text{def}}{=} ?$$

$$der\,c\,([c_1 c_2 \ldots c_n]) \stackrel{\text{def}}{=} ?$$
$$der\,c\,(r^+) \stackrel{\text{def}}{=} ?$$
$$der\,c\,(r^?) \stackrel{\text{def}}{=} ?$$
$$der\,c\,(r^{\{n,m\}}) \stackrel{\text{def}}{=} ?$$
$$der\,c\,(\sim r) \stackrel{\text{def}}{=} ?$$

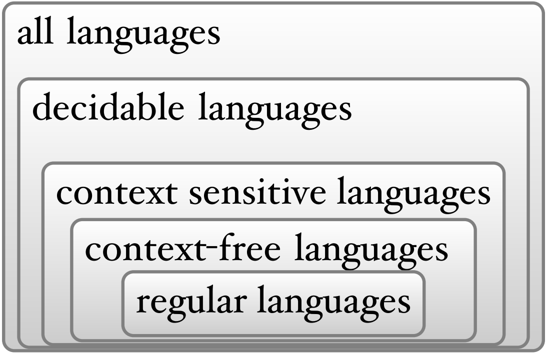
Regular Languages

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

$$(((()()))()) \quad \text{vs.} \quad (((()()))()))$$

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

$$\langle A \rangle ::= \mathit{rhs}$$

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

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 ϵ.

We also allow rules

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

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

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

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 $\langle X\rangle ::= \textit{rhs}$
3. Repeat 2 until there are no nonterminals

$$\langle S\rangle \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$$

Example Derivation

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

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

Example Derivation

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

$$
\begin{aligned}
\langle E \rangle \rightarrow \quad & \langle E \rangle * \langle E \rangle \\
\rightarrow \quad & \langle E \rangle + \langle E \rangle * \langle E \rangle \\
\rightarrow \quad & \langle E \rangle + \langle E \rangle * \langle E \rangle + \langle E \rangle \\
\rightarrow^{+} \; & 1 + 2 * 3 + 4
\end{aligned}
$$

Example Derivation

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

$$\begin{aligned}\langle E\rangle &\rightarrow \langle E\rangle * \langle E\rangle\\ &\rightarrow \langle E\rangle + \langle E\rangle * \langle E\rangle\\ &\rightarrow \langle E\rangle + \langle E\rangle * \langle E\rangle + \langle E\rangle\\ &\rightarrow^{+} 1 + 2 * 3 + 4\end{aligned}$$

$$\begin{aligned}\langle E\rangle &\rightarrow \langle E\rangle + \langle E\rangle\\ &\rightarrow \langle E\rangle + \langle E\rangle + \langle E\rangle\\ &\rightarrow \langle E\rangle + \langle E\rangle * \langle E\rangle + \langle E\rangle\\ &\rightarrow^{+} 1 + 2 * 3 + 4\end{aligned}$$

Context Sensitive Grammars

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

$$
\begin{aligned}
\langle S\rangle &::= b\langle S\rangle\langle A\rangle\langle A\rangle \mid \epsilon \\
\langle A\rangle &::= a \\
b\langle A\rangle &::= \langle A\rangle b
\end{aligned}
$$

Context Sensitive Grammars

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

$$\begin{array}{rcl} \langle S\rangle & ::= & b\langle S\rangle\langle A\rangle\langle A\rangle \mid \epsilon \\ \langle A\rangle & ::= & a \\ b\langle A\rangle & ::= & \langle A\rangle b \end{array}$$

$$\langle S\rangle \rightarrow \ldots \rightarrow^{?} ababaa$$

Language of a CFG

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

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

Language of a CFG

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

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

- 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).

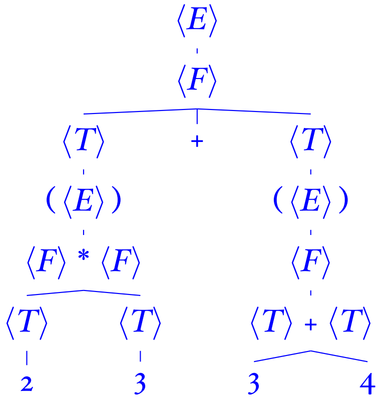
Parse Trees

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

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

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

(2*3)+(3+4)



Arithmetic Expressions

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

Arithmetic Expressions

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

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

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:

$$\begin{array}{rcl} E & \rightarrow & \text{if } E \text{ then } E \\ & | & \text{if } E \text{ then } E \text{ else } E \\ & | & \ldots \end{array}$$

```
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

$$\texttt{Num(123)} :: \textit{rest} \;\Rightarrow\; \{(\texttt{Num(123)}, \textit{rest})\}$$

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

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)

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

Function parser (code $p \Rightarrow f$)

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

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

Function parser (code $p \Rightarrow f$)

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

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

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

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

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 $p \Rightarrow f$ 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

```
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
}
```

```
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 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 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)
}
```

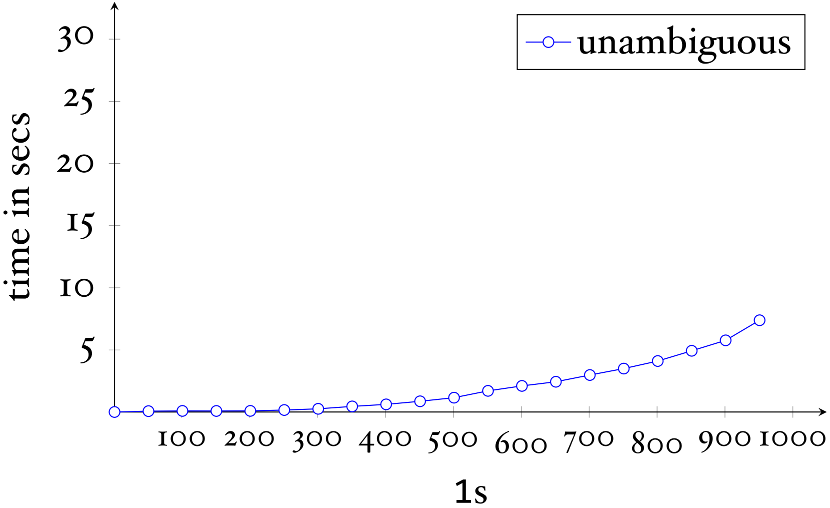
Two Grammars

Which languages are recognised by the following two grammars?

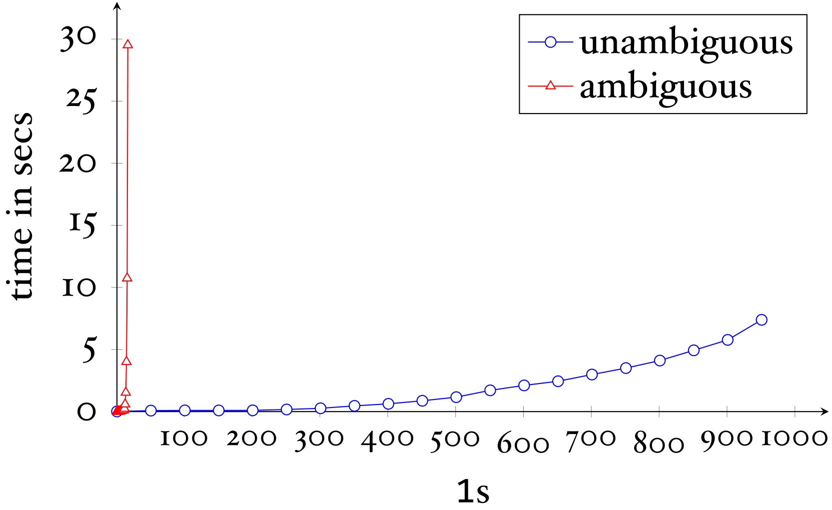
$$\begin{array}{rcl} S & \rightarrow & 1 \cdot S \cdot S \\ & | & \epsilon \end{array}$$

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

Ambiguous Grammars



Ambiguous Grammars



While-Language

$$
\begin{array}{rcl}
\langle \mathit{Stmt} \rangle & ::= & \mathrm{skip} \\
 & | & \langle \mathit{Id} \rangle := \langle \mathit{AExp} \rangle \\
 & | & \mathrm{if}\ \langle \mathit{BExp} \rangle\ \mathrm{then}\ \langle \mathit{Block} \rangle\ \mathrm{else}\ \langle \mathit{Block} \rangle \\
 & | & \mathrm{while}\ \langle \mathit{BExp} \rangle\ \mathrm{do}\ \langle \mathit{Block} \rangle \\
\langle \mathit{Stmts} \rangle & ::= & \langle \mathit{Stmt} \rangle\ ;\ \langle \mathit{Stmts} \rangle \\
 & | & \langle \mathit{Stmt} \rangle \\
\langle \mathit{Block} \rangle & ::= & \{\ \langle \mathit{Stmts} \rangle\ \} \\
 & | & \langle \mathit{Stmt} \rangle \\
\langle \mathit{AExp} \rangle & ::= & \ldots \\
\langle \mathit{BExp} \rangle & ::= & \ldots
\end{array}
$$

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)`