Compilers and Formal Languages (5)

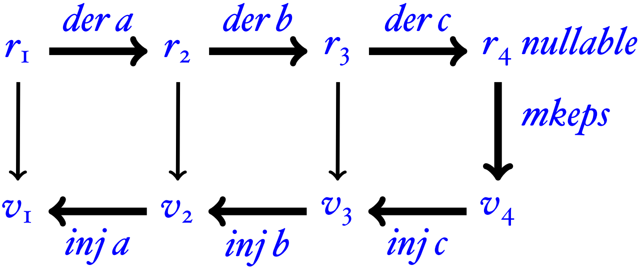
Email:	christian.urban at kcl.ac.uk
Office:	N7.07 (North Wing, Bush House)
Slides:	KEATS (also home work is there)

Last Week
Regexes and Values

Regular expressions and their corresponding values:

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

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

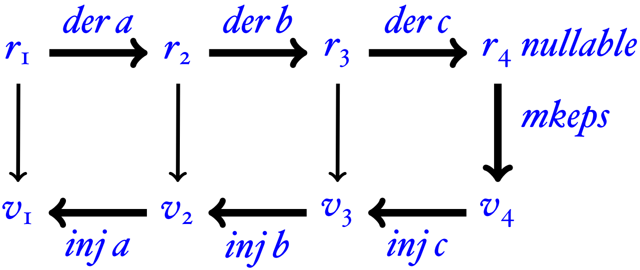


v_1: $Seq(Char(a), Seq(Char(b), Char(c)))$
v_2: $Seq(Empty, Seq(Char(b), Char(c)))$
v_3: $Right(Seq(Empty, Char(c)))$
v_4: $Right(Right(Empty))$

$|v_1|$: 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.



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

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$

Records

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

Records

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

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(+)
```

Coursework: PLs (16)

- Java (16)
- C++, C, C# (14)
- JavaScript (10)
- Scala (9)
- Python (9)
- PHP (6)
- Haskell (3)
- Ruby (4)
- Bash, Perl, Powershell (2)
- TypeScript (1)
- R (1)
- Coconut (1)
- Pascal (1)

Coursework: Nullable

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

$$
\begin{array}{lcl}
der\,c\,([c_1c_2\ldots c_n]) & \stackrel{\text{def}}{=} & ?\\
der\,c\,(r^+) & \stackrel{\text{def}}{=} & ?\\
der\,c\,(r^?) & \stackrel{\text{def}}{=} & ?\\
der\,c\,(r^{\{n\}}) & \stackrel{\text{def}}{=} & if\ n = 0\ then\ \mathbf{0}\ else\ (der\,c\,r)\cdot r^{\{n-1\}}\\
der\,c\,(r^{\{n..\}}) & \stackrel{\text{def}}{=} & if\ n = 0\ then (der\,c\,r)\cdot r^*\\
 & & \quad else\ (der\,c\,r)\cdot r^{\{n-1..\}}\\
der\,c\,(r^{\{..n\}}) & \stackrel{\text{def}}{=} & if\ n = 0\ then\ \mathbf{0}\ else\ (der\,c\,r)\cdot r^{\{..n-1\}}\\
der\,c\,(r^{\{n..m\}}) & \stackrel{\text{def}}{=} & if\ n = 0 \wedge m = 0\ then\ \mathbf{0}\ else\\
\multicolumn{3}{l}{\quad if\ n = 0 \wedge m > 0\ then\ (der\,c\,r)\cdot r^{\{..m-1\}}}\\
\multicolumn{3}{l}{\qquad\qquad else\ (der\,c\,r)\cdot r^{\{n-1..m-1\}}}\\
der\,c\,(\sim r) & \stackrel{\text{def}}{=} & ?
\end{array}
$$

Coursework: CFUN

$$
\begin{array}{lcl}
nullable(CFUN(\_)) & \stackrel{\text{def}}{=} & false \\
der\,c\,(CFUN(f)) & \stackrel{\text{def}}{=} & if\ f(c)\ then\ \mathbf{1}\ else\ \mathbf{0} \\
\\
CHAR(c) & \stackrel{\text{def}}{=} & CFUN(\lambda d.\ c = d) \\
CSET([c_1, \ldots, c_n]) & \stackrel{\text{def}}{=} & CFUN(\lambda d.\ d \in [c_1, \ldots, c_n]) \\
ALL & \stackrel{\text{def}}{=} & CFUN(\lambda d.\ true)
\end{array}
$$

Lexer, Parser

Today a parser.

What Parsing is Not

Usually parsing does not check semantic correctness, e.g.

- whether a function is not used before it is defined
- whether a function has the correct number of arguments or are of correct type
- whether a variable can be declared twice in a scope

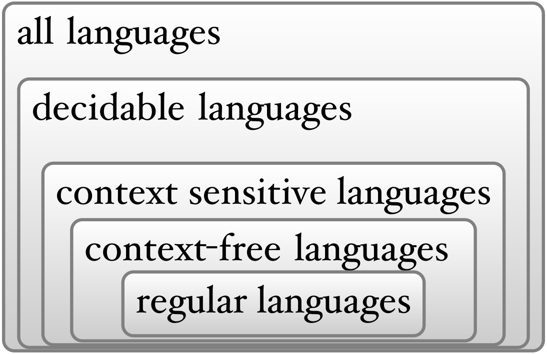
Regular Languages

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

$$A ::= rhs$$

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

$$A ::= rhs$$

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

We also allow rules

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

Palindromes

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

$$\boldsymbol{S} ::= \epsilon$$
$$\boldsymbol{S} ::= a \cdot \boldsymbol{S} \cdot a$$
$$\boldsymbol{S} ::= b \cdot \boldsymbol{S} \cdot b$$

Palindromes

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

$$\boldsymbol{S} ::= \epsilon$$
$$\boldsymbol{S} ::= a \cdot \boldsymbol{S} \cdot a$$
$$\boldsymbol{S} ::= b \cdot \boldsymbol{S} \cdot b$$

or

$$\boldsymbol{S} ::= \epsilon \mid a \cdot \boldsymbol{S} \cdot a \mid b \cdot \boldsymbol{S} \cdot b$$

Palindromes

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

$$\boldsymbol{S} ::= \epsilon$$
$$\boldsymbol{S} ::= a \cdot \boldsymbol{S} \cdot a$$
$$\boldsymbol{S} ::= b \cdot \boldsymbol{S} \cdot b$$

or

$$\boldsymbol{S} ::= \epsilon \mid a \cdot \boldsymbol{S} \cdot a \mid b \cdot \boldsymbol{S} \cdot b$$

Can you find the grammar rules for matched parentheses?

Arithmetic Expressions

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

Arithmetic Expressions

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

```
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 $X ::= rhs$
3. Repeat 2 until there are no nonterminals left

$$S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$$

Example Derivation

$$\mathbf{S} ::= \epsilon \mid a \cdot \mathbf{S} \cdot a \mid b \cdot \mathbf{S} \cdot b$$

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

Example Derivation

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

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

Example Derivation

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

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

Context Sensitive Grammars

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

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

Context Sensitive Grammars

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

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

$$\boldsymbol{S} \rightarrow \ldots \rightarrow^{?} ababaa$$

Language of a CFG

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

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

Language of a CFG

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

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

- 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).

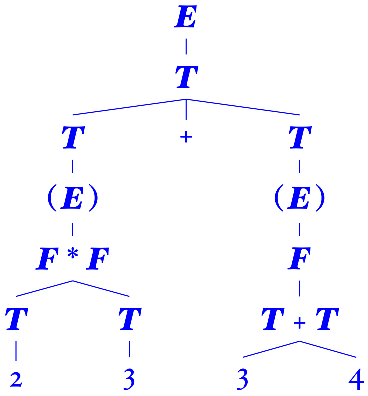
Parse Trees

$$E ::= F \mid T \cdot + \cdot E \mid T \cdot - \cdot E$$

$$T ::= F \mid F \cdot * \cdot T$$

$$F ::= \textit{num\_token} \mid (\cdot E \cdot)$$

(2*3)+(3+4)



Arithmetic Expressions

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

Arithmetic Expressions

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

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

Ambiguous Grammars

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

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

`1 + 2 * 3 + 4`

‘Dangling’ Else

Another ambiguous grammar:

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

```
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 $p \,||\, q$)

- 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 part
- 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

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

Semantic Actions

Addition

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

Multiplication

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

Semantic Actions

Addition

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

Multiplication

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

Parenthesis

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

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 $p \,||\, q$ 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
}
```

```scala
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)
}
```

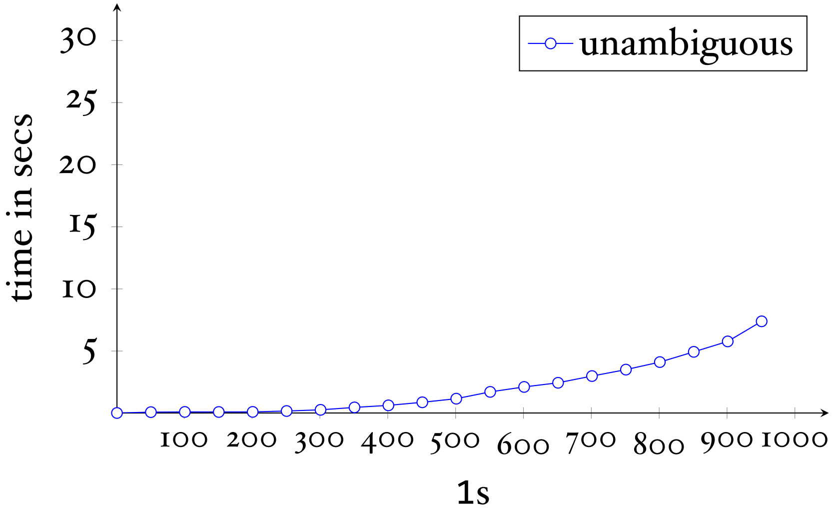
Two Grammars

Which languages are recognised by the following two grammars?

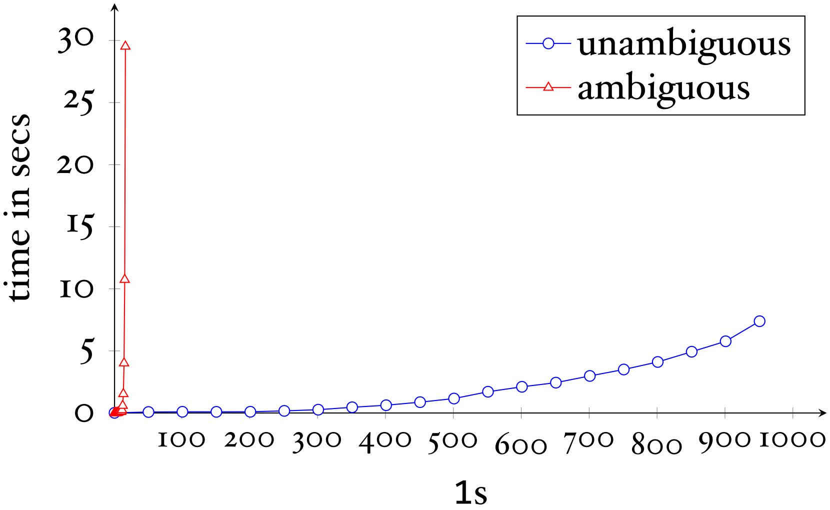
$$\begin{array}{lcl} S & \rightarrow & 1 \cdot S \cdot S \\ & | & \epsilon \end{array}$$

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

Ambiguous Grammars



Ambiguous Grammars



While-Language

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

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

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

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

Interpreter

$$
\begin{array}{lcl}
\text{eval}(n, E) & \stackrel{\text{def}}{=} & n \\
\text{eval}(x, E) & \stackrel{\text{def}}{=} & E(x) \quad \text{lookup } x \text{ in } E \\
\text{eval}(a_1 + a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) + \text{eval}(a_2, E) \\
\text{eval}(a_1 - a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) - \text{eval}(a_2, E) \\
\text{eval}(a_1 * a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) * \text{eval}(a_2, E) \\
\\
\text{eval}(a_1 = a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) = \text{eval}(a_2, E) \\
\text{eval}(a_1 \mathrel{!=} a_2, E) & \stackrel{\text{def}}{=} & \neg(\text{eval}(a_1, E) = \text{eval}(a_2, E)) \\
\text{eval}(a_1 < a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) < \text{eval}(a_2, E)
\end{array}
$$

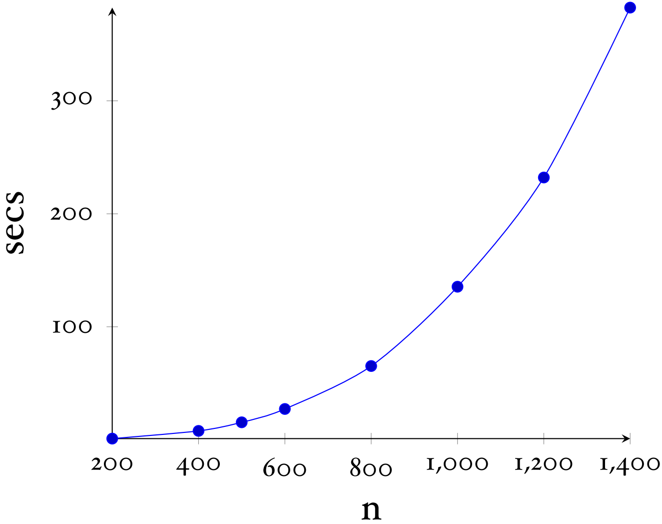
Interpreter (2)

$$
\begin{array}{lcl}
\text{eval}(\text{skip}, E) & \stackrel{\text{def}}{=} & E \\
\text{eval}(x := a, E) & \stackrel{\text{def}}{=} & E(x \mapsto \text{eval}(a, E)) \\
\text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) & \stackrel{\text{def}}{=} & \\
\qquad \text{if eval}(b, E) \text{ then eval}(cs_1, E) & & \\
\qquad \qquad \text{else eval}(cs_2, E) & & \\
\text{eval}(\text{while } b \text{ do } cs, E) & \stackrel{\text{def}}{=} & \\
\qquad \text{if eval}(b, E) & & \\
\qquad \text{then eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) & & \\
\qquad \text{else } E & & \\
\text{eval}(\text{write } x, E) & \stackrel{\text{def}}{=} & \{\ \text{println}(E(x))\ ;\ E\ \}
\end{array}
$$

Test Program

```
start := 1000;
x := start;
y := start;
z := start;
while 0 < x do {
 while 0 < y do {
  while 0 < z do { z := z - 1 };
  z := start;
  y := y - 1
 };
 y := start;
 x := x - 1
}
```

Interpreted Code



Java Virtual Machine

- introduced in 1995
- is a stack-based VM (like Postscript, CLR of .Net)
- contains a JIT compiler
- many languages take advantage of JVM's infrastructure (JRE)
- is garbage collected ⇒ no buffer overflows
- some languages compile to the JVM: Scala, Clojure...