Compilers and Formal Languages

Email:	christian.urban at kcl.ac.uk
Slides & Progs:	KEATS (also homework is there)

1 Introduction, Languages	6 While-Language
2 Regular Expressions, Derivatives	7 Compilation, JVM
3 Automata, Regular Languages	8 Compiling Functional Languages
4 Lexing, Tokenising	9 Optimisations
5 Grammars, Parsing	10 LLVM

Parser Combinators

One of the simplest ways to implement a parser, see `https://vimeo.com/142341803` (by Haoyi Li)

- build-in library in Scala
- fastparse (2) library by Haoyi Li; is part of Ammonite
- possible exponential runtime behaviour

Parser Combinators

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 (map-parser)

Atomic parsers, for example, number tokens

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

- 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 part

then combine the results:

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

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

Map-parser (code $p.map(f)$)

apply p producing a set of pairs

then apply the function f to each first component

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

Map-parser (code $p.map(f)$)

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

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

Multiplication

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

Semantic Actions

Addition

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

Multiplication

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

Parenthesis

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

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 $p \mid\mid q$ 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 using lexers is better because whitespaces or comments can 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)
}
```

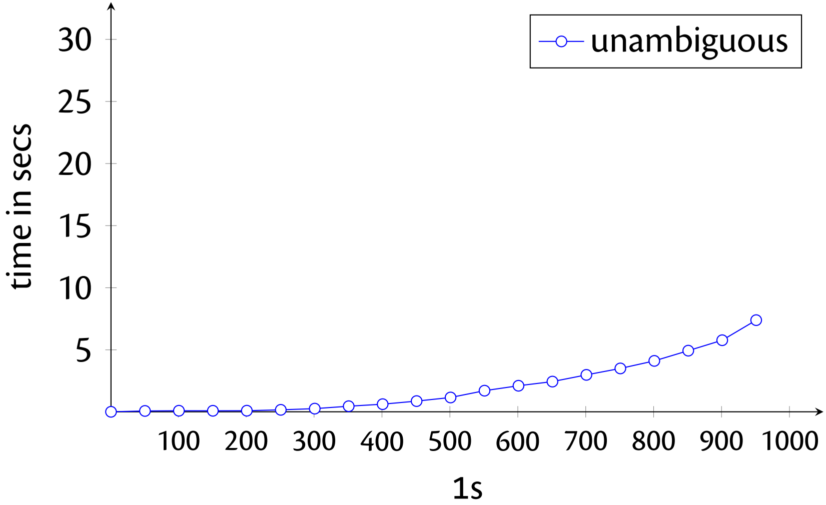
Two Grammars

Which languages are recognised by the following two grammars?

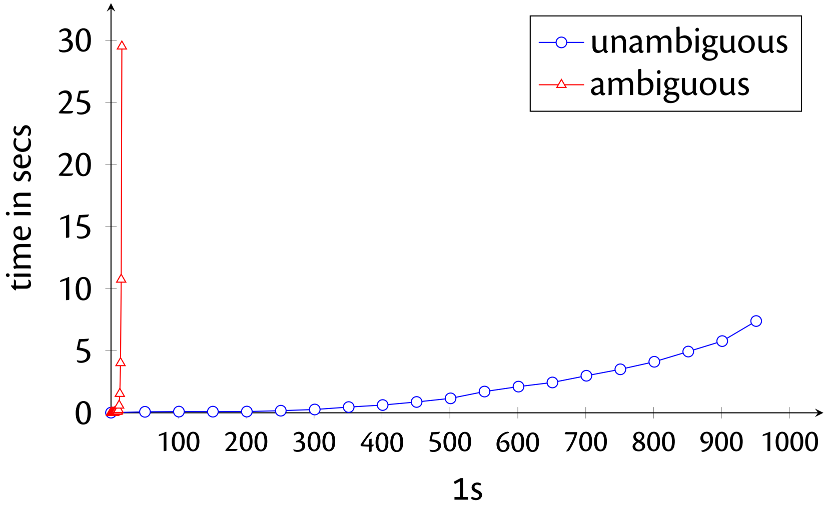
$$\begin{aligned} S &\rightarrow 1 \cdot S \cdot S \\ &\mid \epsilon \end{aligned}$$

$$\begin{aligned} U &\rightarrow 1 \cdot U \\ &\mid \epsilon \end{aligned}$$

Ambiguous Grammars



Ambiguous Grammars



While-Language

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

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

$$
\begin{array}{l}
\{\\
\quad x := 5;\\
\quad y := x * 3;\\
\quad y := x * 4;\\
\quad x := u * 3\\
\}
\end{array}
$$

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

`eval(stmt, env)`

Interpreter

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

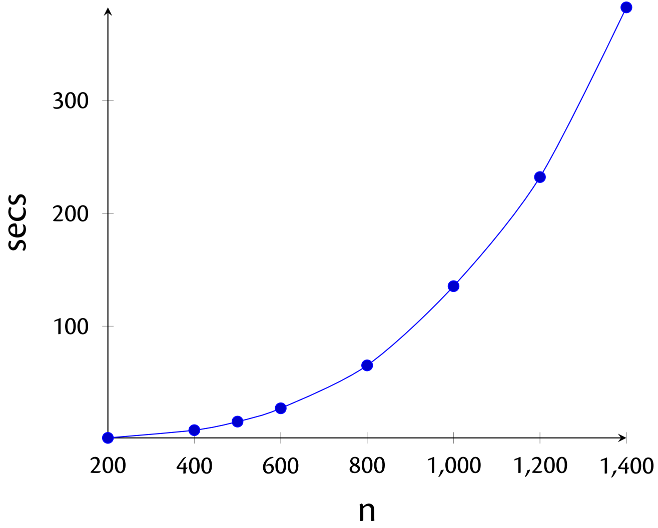
Interpreter (2)

$$
\begin{array}{lcl}
\mathrm{eval}(\mathrm{skip}, E) & \stackrel{\mathrm{def}}{=} & E \\
\mathrm{eval}(x := a, E) & \stackrel{\mathrm{def}}{=} & E(x \mapsto \mathrm{eval}(a, E)) \\
\multicolumn{3}{l}{\mathrm{eval}(\mathrm{if}\ b\ \mathrm{then}\ cs_1\ \mathrm{else}\ cs_2, E) \stackrel{\mathrm{def}}{=}} \\
\multicolumn{3}{l}{\qquad\qquad \mathrm{if}\ \mathrm{eval}(b, E)\ \mathrm{then}\ \mathrm{eval}(cs_1, E)} \\
\multicolumn{3}{l}{\qquad\qquad\qquad\qquad\quad\ \ \mathrm{else}\ \mathrm{eval}(cs_2, E)} \\
\multicolumn{3}{l}{\mathrm{eval}(\mathrm{while}\ b\ \mathrm{do}\ cs, E) \stackrel{\mathrm{def}}{=}} \\
\multicolumn{3}{l}{\qquad\qquad \mathrm{if}\ \mathrm{eval}(b, E)} \\
\multicolumn{3}{l}{\qquad\qquad \mathrm{then}\ \mathrm{eval}(\mathrm{while}\ b\ \mathrm{do}\ cs, \mathrm{eval}(cs, E))} \\
\multicolumn{3}{l}{\qquad\qquad \mathrm{else}\ E} \\
\mathrm{eval}(\mathrm{write}\ x, E) & \stackrel{\mathrm{def}}{=} & \{\ \mathrm{println}(E(x))\ ;\ E\ \}
\end{array}
$$

Test Program

??

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…

Compilers and Formal Languages

Email: christian.urban at kcl.ac.uk
Slides & Progs: KEATS (also homework is there)

1 Introduction, Languages
2 Regular Expressions, Derivatives
3 Automata, Regular Languages
4 Lexing, Tokenising
5 Grammars, Parsing
6 While-Language
7 Compilation, JVM
8 Compiling Functional Languages
9 Optimisations
10 LLVM

Starting Symbol

$$S ::= A \cdot S \cdot B \mid B \cdot S \cdot A \mid \epsilon$$

$$A ::= a \mid \epsilon$$

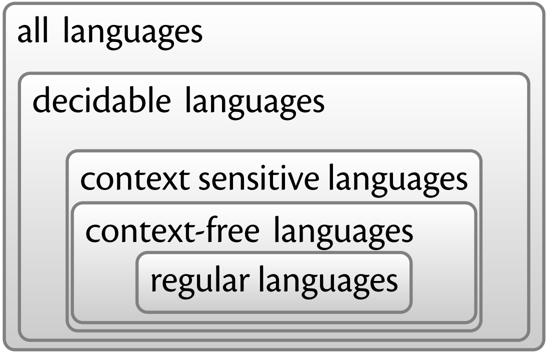
$$B ::= b$$

TODO: Testcases for math expressions

`https://github.com/ArashPartow/math-parser-benchmark-project`

Hierarchy of Languages

Recall that languages are sets of strings.



Parser Combinators

Atomic parsers, for example

$$1 :: rest \;\Rightarrow\; \{(1, rest)\}$$

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

Alternative parser (code $p \mid q$)

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)

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

Function parser (code $p \Rightarrow f$)

apply p producing a set of pairs

then apply the function f to each first component

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

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 $p \mid q$ 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 $p \mid q$ 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

$$S$$

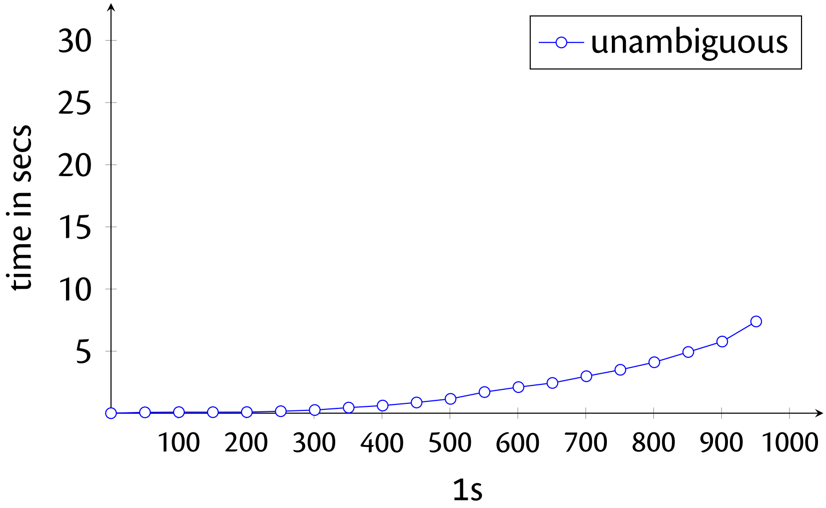
Two Grammars

Which languages are recognised by the following two grammars?

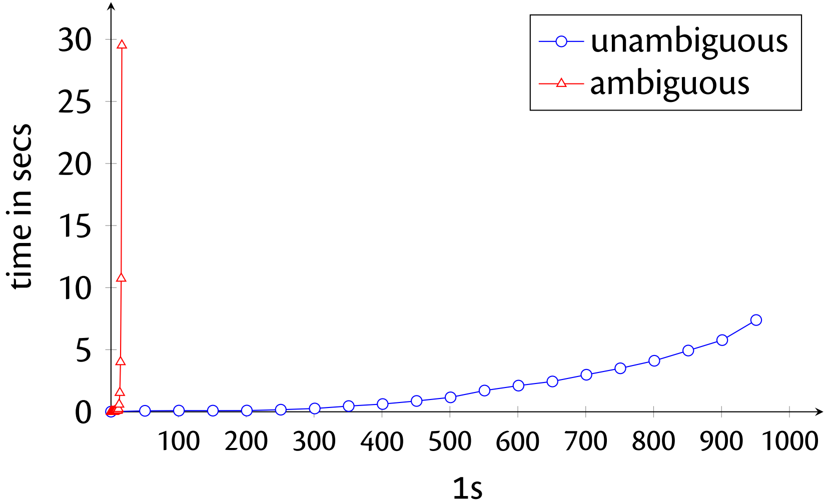
$$S ::= 1 \cdot S \cdot S \mid \epsilon$$

$$U ::= 1 \cdot U \mid \epsilon$$

Ambiguous Grammars



Ambiguous Grammars



Arithmetic Expressions

A grammar for arithmetic expressions and numbers:

$$
\begin{aligned}
E &::= E \cdot + \cdot E \mid E \cdot * \cdot E \mid (\cdot E \cdot) \mid N \\
N &::= N \cdot N \mid 0 \mid 1 \mid \ldots \mid 9
\end{aligned}
$$

Unfortunately it is left-recursive (and ambiguous).

A problem for recursive descent parsers (e.g. parser combinators).

Arithmetic Expressions

A grammar for arithmetic expressions and numbers:

$$E ::= E \cdot + \cdot E \mid E \cdot * \cdot E \mid (\cdot E \cdot) \mid N$$
$$N ::= N \cdot N \mid 0 \mid 1 \mid \ldots \mid 9$$

Unfortunately it is left-recursive (and ambiguous).

A problem for recursive descent parsers (e.g. parser combinators).

Numbers

$$N ::= N \cdot N \mid 0 \mid 1 \mid \ldots \mid 9$$

A non-left-recursive, non-ambiguous grammar for numbers:

$$N ::= 0 \cdot N \mid 1 \cdot N \mid \ldots \mid 0 \mid 1 \mid \ldots \mid 9$$

Removing Left-Recursion

The rule for numbers is directly left-recursive:

$$\boldsymbol{N} \quad ::= \quad \boldsymbol{N} \cdot \boldsymbol{N} \mid 0 \mid 1 \qquad (\ldots)$$

Translate

$$\begin{array}{lcl} \boldsymbol{N} & ::= & \boldsymbol{N} \cdot \alpha \\ & \mid & \beta \end{array} \quad \Rightarrow \quad \begin{array}{lcl} \boldsymbol{N} & ::= & \beta \cdot \boldsymbol{N}' \\ \boldsymbol{N}' & ::= & \alpha \cdot \boldsymbol{N}' \\ & \mid & \epsilon \end{array}$$

Removing Left-Recursion

The rule for numbers is directly left-recursive:

$$N \quad ::= \quad N \cdot N \mid 0 \mid 1 \qquad (\ldots)$$

Translate

$$\begin{array}{lll} N & ::= & N \cdot \alpha \\ & \mid & \beta \end{array} \quad \Rightarrow \quad \begin{array}{lll} N & ::= & \beta \cdot N' \\ N' & ::= & \alpha \cdot N' \\ & \mid & \epsilon \end{array}$$

Which means in this case:

$$\begin{array}{lll} N & \rightarrow & 0 \cdot N' \mid 1 \cdot N' \\ N' & \rightarrow & N \cdot N' \mid \epsilon \end{array}$$

Chomsky Normal Form

All rules must be of the form

$$A ::= a$$

or

$$A ::= B \cdot C$$

No rule can contain ϵ.

ϵ-Removal

If $A ::= \alpha \cdot B \cdot \beta$ and $B ::= \epsilon$ are in the grammar, then add $A ::= \alpha \cdot \beta$ (iterate if necessary).

Throw out all $B ::= \epsilon$.

$$\begin{array}{lcl} N & ::= & 0 \cdot N' \mid 1 \cdot N' \\ N' & ::= & N \cdot N' \mid \epsilon \end{array}$$

$$\begin{array}{lcl} N & ::= & 0 \cdot N' \mid 1 \cdot N' \mid 0 \mid 1 \\ N' & ::= & N \cdot N' \mid N \mid \epsilon \end{array}$$

$$\begin{array}{lcl} N & ::= & 0 \cdot N' \mid 1 \cdot N' \mid 0 \mid 1 \\ N' & ::= & N \cdot N' \mid N \end{array}$$

ϵ-Removal

If $A ::= \alpha \cdot B \cdot \beta$ and $B ::= \epsilon$ are in the grammar, then add $A ::= \alpha \cdot \beta$ (iterate if necessary).

Throw out all $B ::= \epsilon$.

$$\begin{array}{lcl} N & ::= & 0 \cdot N' \mid 1 \cdot N' \\ N' & ::= & N \cdot N' \mid \epsilon \end{array}$$

$$\begin{array}{lcl} N & ::= & 0 \cdot N' \mid 1 \cdot N' \mid 0 \mid 1 \\ N' & ::= & N \cdot N' \mid N \mid \epsilon \end{array}$$

$$\begin{array}{lcl} N & ::= & 0 \cdot N' \mid 1 \cdot N' \mid 0 \mid 1 \\ N' & ::= & N \cdot N' \mid N \end{array}$$

$$N ::= 0 \cdot N \mid 1 \cdot N \mid 0 \mid 1$$

CYK Algorithm

If grammar is in Chomsky normalform …

$$
\begin{array}{lcl}
S & ::= & N \cdot P \\
P & ::= & V \cdot N \\
N & ::= & N \cdot N \\
N & ::= & \texttt{students} \,|\, \texttt{Jeff} \,|\, \texttt{geometry} \,|\, \texttt{trains} \\
V & ::= & \texttt{trains}
\end{array}
$$

`Jeff trains geometry students`

CYK Algorithm

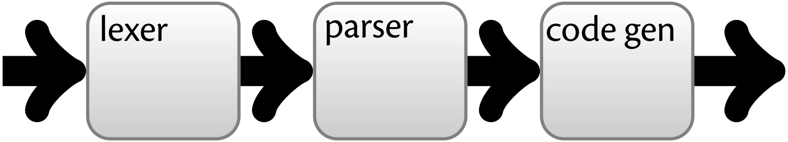
fastest possible algorithm for recognition problem

runtime is $O(n^3)$

grammars need to be transformed into CNF

The Goal of this Course

Write a Compiler



We have a lexer and a parser...

$$
\begin{array}{lcl}
\mathit{Stmt} & ::= & \texttt{skip} \\
 & | & \mathit{Id} \texttt{ := } \mathit{AExp} \\
 & | & \texttt{if } \mathit{BExp} \texttt{ then } \mathit{Block} \texttt{ else } \mathit{Block} \\
 & | & \texttt{while } \mathit{BExp} \texttt{ do } \mathit{Block} \\
 & | & \texttt{read } \mathit{Id} \\
 & | & \texttt{write } \mathit{Id} \\
 & | & \texttt{write } \mathit{String} \\
\mathit{Stmts} & ::= & \mathit{Stmt} \ ; \ \mathit{Stmts} \\
 & | & \mathit{Stmt} \\
\mathit{Block} & ::= & \{ \mathit{Stmts} \} \\
 & | & \mathit{Stmt} \\
\mathit{AExp} & ::= & \ldots \\
\mathit{BExp} & ::= & \ldots
\end{array}
$$

??

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

$$
\begin{array}{l}
\{\\
\quad x := 5;\\
\quad y := x * 3;\\
\quad y := x * 4;\\
\quad x := u * 3\\
\}
\end{array}
$$

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

eval(stmt, env)

An Interpreter

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

$$\begin{array}{lcl}
\text{eval}(a_1 = a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) = \text{eval}(a_2, E) \\
\text{eval}(a_1 \mathrel{!}= a_2, E) & \stackrel{\text{def}}{=} & \neg(\text{eval}(a_1, E) = \text{eval}(a_2, E)) \\
\text{eval}(a_1 < a_2, E) & \stackrel{\text{def}}{=} & \text{eval}(a_1, E) < \text{eval}(a_2, E)
\end{array}$$

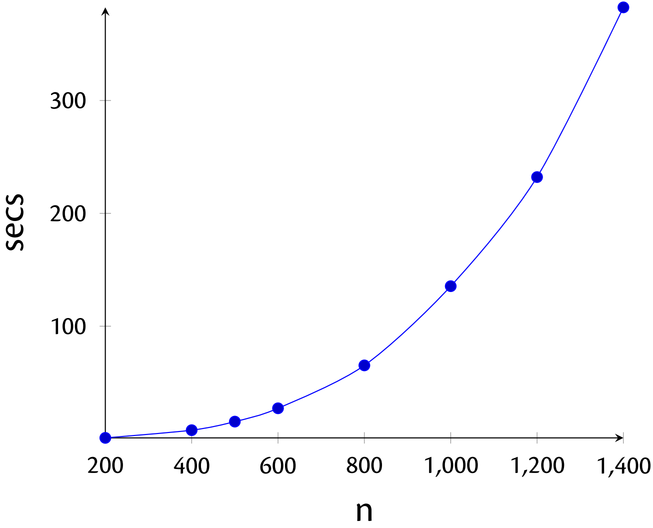
An Interpreter (2)

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

Test Program

??

Interpreted Code



Java Virtual Machine

- introduced in 1995
- is a stack-based VM (like Postscript, CLR of .Net)
- contains a JIT compiler
  - From the Cradle to the Holy Graal - the JDK Story
    https://www.youtube.com/watch?v=h419kfbLhUI
- is garbage collected ⇒ no buffer overflows
- some languages compile to the JVM: Scala, Clojure…

LLVM

LLVM started by academics in 2000 (University of Illinois in Urbana-Champaign)

suite of compiler tools

SSA-based intermediate language

no need to allocate registers

source languages: C, C++, Rust, Go, Swift

target CPUs: x86, ARM, PowerPC, …