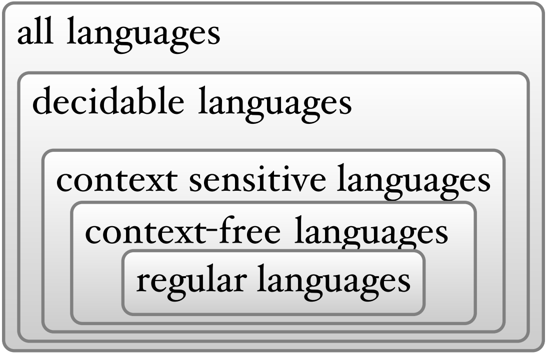
Compilers and Formal Languages (6)

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

Hierarchy of Languages

Recall that languages are sets of strings.



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 $p \,||\, q$)

- 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

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

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 $p \;||\; q$ 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

$$S$$

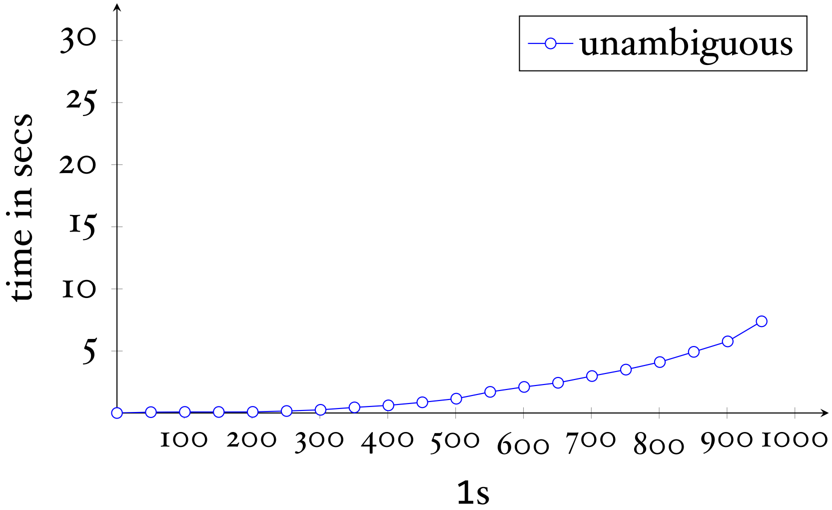
Two Grammars

Which languages are recognised by the following two grammars?

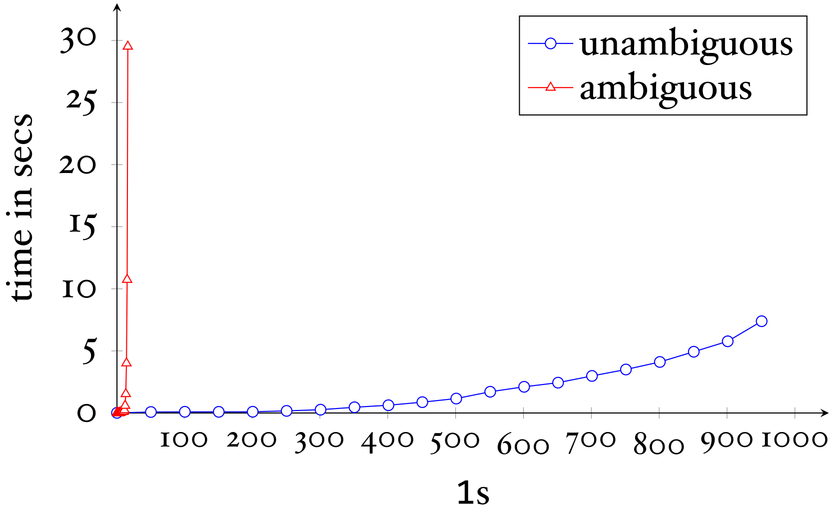
$$\boldsymbol{S} ::= 1 \cdot \boldsymbol{S} \cdot \boldsymbol{S} \mid \epsilon$$

$$\boldsymbol{U} ::= 1 \cdot \boldsymbol{U} \mid \epsilon$$

Ambiguous Grammars



Ambiguous Grammars



Arithmetic Expressions

A grammar for arithmetic expressions and numbers:

$$\boldsymbol{E} ::= \boldsymbol{E} \cdot + \cdot \boldsymbol{E} \mid \boldsymbol{E} \cdot * \cdot \boldsymbol{E} \mid (\cdot \boldsymbol{E} \cdot) \mid \boldsymbol{N}$$

$$\boldsymbol{N} ::= \boldsymbol{N} \cdot \boldsymbol{N} \mid 0 \mid 1 \mid \ldots \mid 9$$

Unfortunately it is left-recursive (and ambiguous).

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

Arithmetic Expressions

A grammar for arithmetic expressions and numbers:

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

Unfortunately it is left-recursive (and ambiguous).

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

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:

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

Translate

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

Removing Left-Recursion

The rule for numbers is directly left-recursive:

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

Translate

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

Which means in this case:

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

Operator Precedences

To disambiguate

$$\boldsymbol{E} \quad ::= \quad \boldsymbol{E} \cdot + \cdot \boldsymbol{E} \mid \boldsymbol{E} \cdot * \cdot \boldsymbol{E} \mid (\cdot \boldsymbol{E} \cdot) \mid \boldsymbol{N}$$

Decide on how many precedence levels, say

highest for $()$, medium for $*$, lowest for $+$

$$\begin{array}{lcl}
\boldsymbol{E}_{low} & ::= & \boldsymbol{E}_{med} \cdot + \cdot \boldsymbol{E}_{low} \mid \boldsymbol{E}_{med} \\
\boldsymbol{E}_{med} & ::= & \boldsymbol{E}_{hi} \cdot * \cdot \boldsymbol{E}_{med} \mid \boldsymbol{E}_{hi} \\
\boldsymbol{E}_{hi} & ::= & (\cdot \boldsymbol{E}_{low} \cdot) \mid \boldsymbol{N}
\end{array}$$

Operator Precedences

To disambiguate

$$\boldsymbol{E} \quad ::= \quad \boldsymbol{E} \cdot + \cdot \boldsymbol{E} \mid \boldsymbol{E} \cdot * \cdot \boldsymbol{E} \mid (\cdot \boldsymbol{E} \cdot) \mid \boldsymbol{N}$$

Decide on how many precedence levels, say

highest for $()$, medium for $*$, lowest for $+$

$$\begin{array}{lcl}
\boldsymbol{E}_{low} & ::= & \boldsymbol{E}_{med} \cdot + \cdot \boldsymbol{E}_{low} \mid \boldsymbol{E}_{med} \\
\boldsymbol{E}_{med} & ::= & \boldsymbol{E}_{hi} \cdot * \cdot \boldsymbol{E}_{med} \mid \boldsymbol{E}_{hi} \\
\boldsymbol{E}_{hi} & ::= & (\cdot \boldsymbol{E}_{low} \cdot) \mid \boldsymbol{N}
\end{array}$$

What happens with $1 + 3 + 4$?

Chomsky Normal Form

All rules must be of the form

$$A ::= a$$

or

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

No rule can contain ϵ.

ϵ-Removal

1. If $A ::= \alpha \cdot B \cdot \beta$ and $B ::= \epsilon$ are in the grammar, then add $A ::= \alpha \cdot \beta$ (iterate if necessary).
2. 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

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

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

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

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

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

CYK Algorithm

If grammar is in Chomsky normalform ...

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

`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 lexer and parser.

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

```
write "Fib";
read n;
minus1 := 0;
minus2 := 1;
while n > 0 do {
       temp := minus2;
       minus2 := minus1 + minus2;
       minus1 := temp;
       n := n - 1
};
write "Result";
write minus2
```

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

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

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

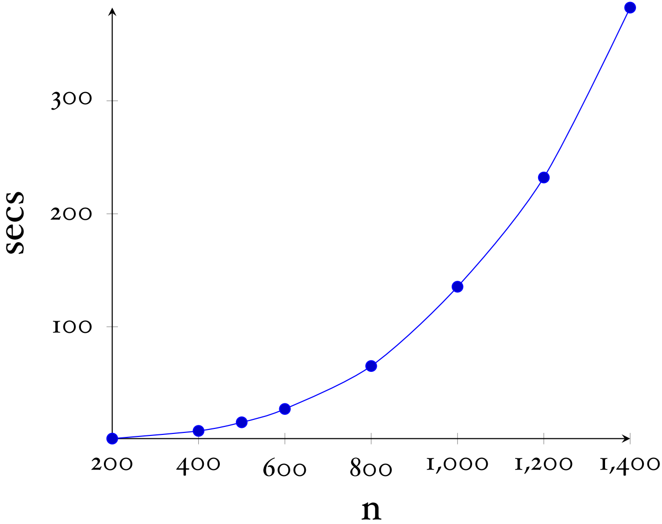
Interpreter (2)

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

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 $\Rightarrow$ no buffer overflows
- some languages compile to the JVM: Scala, Clojure...