Compilers and Formal Languages (6)

Email:	christian.urban at kcl.ac.uk
Office Hours:	Thursdays 12 – 14
Location:	N7.07 (North Wing, Bush House)
Slides & Progs:	KEATS (also homework is there)

Starting Symbol

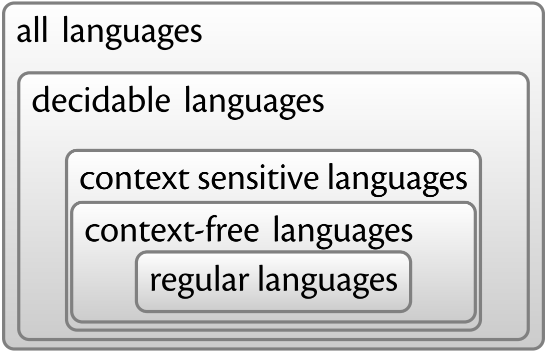
$$S ::= A \cdot S \cdot B \mid B \cdot S \cdot A \mid \epsilon$$

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

$$B ::= b$$

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 $p \mid\mid q$)

- 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

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

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
$$S$$

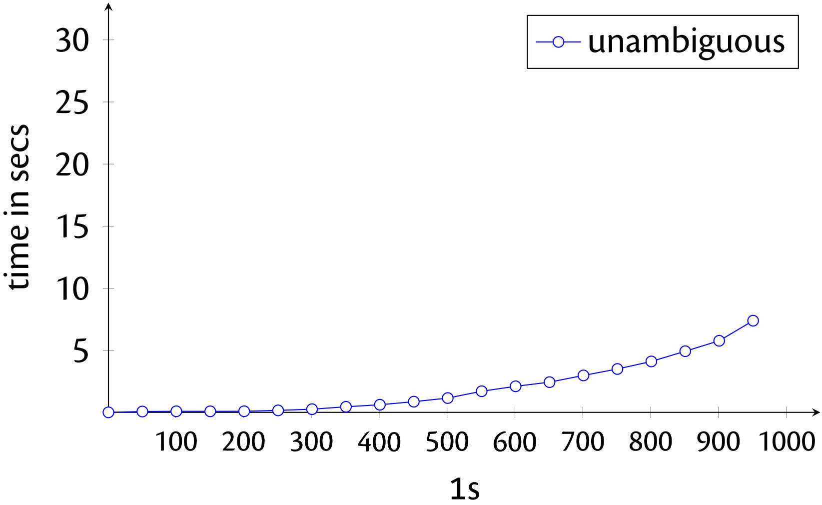
Two Grammars

Which languages are recognised by the following two grammars?

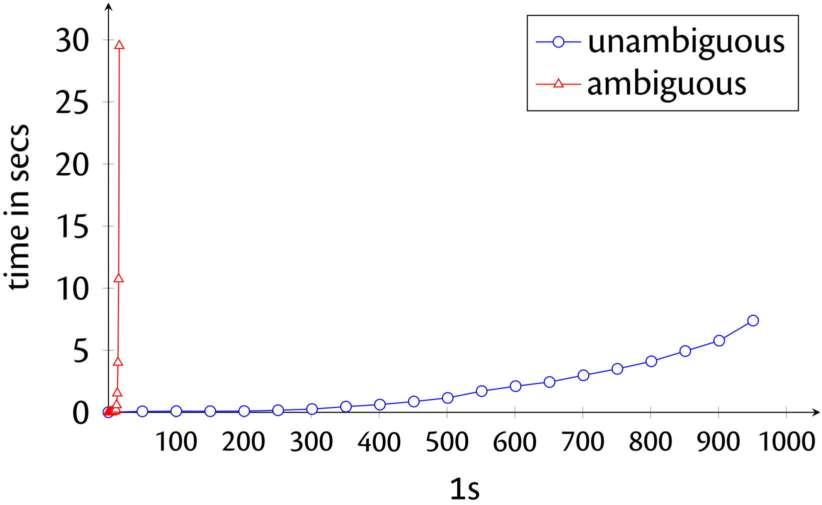
$$S ::= 1 \cdot S \cdot S \mid \epsilon$$

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

Ambiguous Grammars



Ambiguous Grammars



Arithmetic Expressions

A grammar for arithmetic expressions and numbers:

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

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

Unfortunately it is left-recursive (and ambiguous).

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

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:

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

Translate

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

Removing Left-Recursion

The rule for numbers is directly left-recursive:

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

Translate

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

Which means in this case:

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

Chomsky Normal Form

All rules must be of the form

$$A ::= a$$

or

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

No rule can contain ϵ.

ϵ-Removal

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 $B ::= \epsilon$.

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

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

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

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 $B ::= \epsilon$.

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

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

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

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

CYK Algorithm

If grammar is in Chomsky normalform …

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

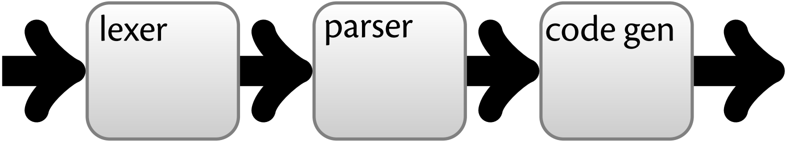
`Jeff trains geometry students`

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...

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

```
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)

An Interpreter

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

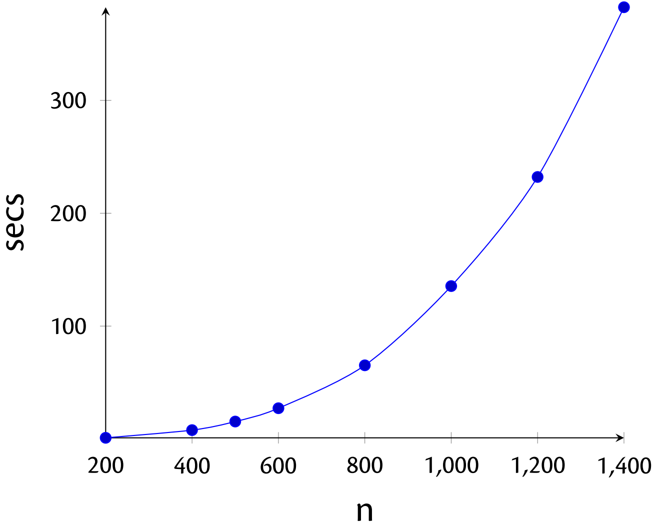
An Interpreter (2)

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

Test Program

```
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
  - From the Cradle to the Holy Graal - the JDK Story
  - `https://www.youtube.com/watch?v=h419kfbLhUI`
- is garbage collected $\Rightarrow$ 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