Compilers and
Formal Languages (6)
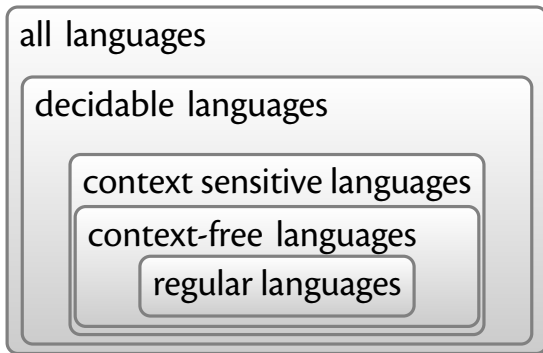
Email: christian.urban at kcl.ac.uk
Office Hours: Thursdays 12 – 14
Location: N7.07 (North Wing, Bush House)
Slides & Progs: KEATS (also homework is there)

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

$$1 :: rest \implies \{(1, rest)\}$$

- 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:

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

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

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 $p \mid q$ 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
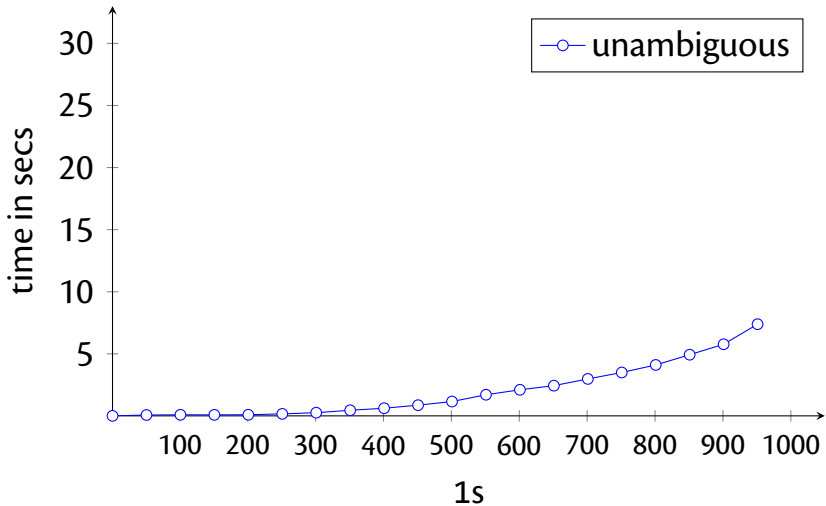
$$S$$

Two Grammars

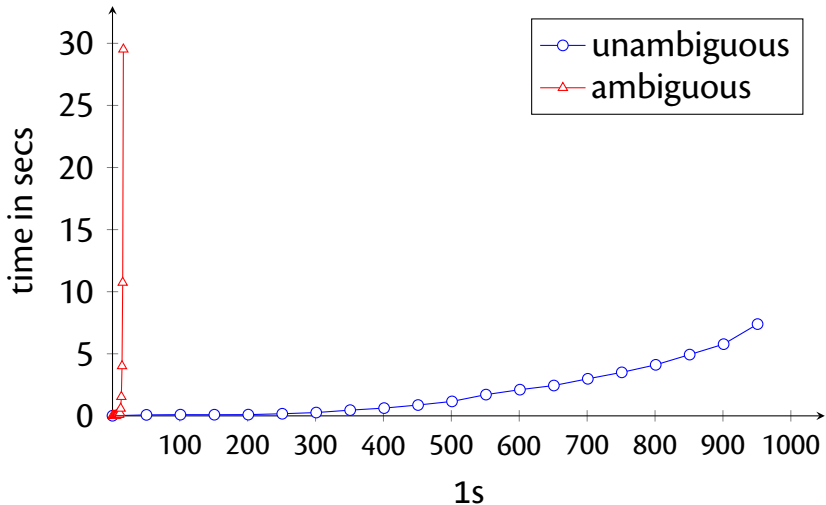Which languages are recognised by the following two grammars?

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

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

Ambiguous Grammars

Ambiguous Grammars

Arithmetic Expressions

A grammar for arithmetic expressions and numbers:

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

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:

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

Translate

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

Removing Left-Recursion

The rule for numbers is directly left-recursive:

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

Translate

$$
\begin{aligned}
N \quad &::= \quad N \cdot \alpha \\
&\mid \quad \beta
\end{aligned}
\quad \Longrightarrow \quad
\begin{aligned}
N \quad &::= \quad \beta \cdot N' \\
N' \quad &::= \quad \alpha \cdot N' \\
&\mid \quad \epsilon
\end{aligned}
$$

Which means in this case:

$$
\begin{aligned}
N \quad &\to \quad 0 \cdot N' \mid 1 \cdot N' \\
N' \quad &\to \quad N \cdot N' \mid \epsilon
\end{aligned}
$$

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 $B ::= \epsilon$.

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

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

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

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 $B ::= \epsilon$.

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

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

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

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

CYK Algorithm

If grammar is in Chomsky normalform ...

$$S \quad ::= \quad N \cdot P$$
$$P \quad ::= \quad V \cdot N$$
$$N \quad ::= \quad N \cdot N$$
$$N \quad ::= \quad \text{students} \mid \text{Jeff} \mid \text{geometry} \mid \text{trains}$$
$$V \quad ::= \quad \text{trains}$$

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...

Stmt	::=	skip
	|	*Id* := *AExp*
	|	if *BExp* then *Block* else *Block*
	|	while *BExp* do *Block*
	|	read *Id*
	|	write *Id*
	|	write *String*

| *Stmts* | ::= | *Stmt* ; *Stmts* |
| | \| | *Stmt* |

| *Block* | ::= | { *Stmts* } |
| | \| | *Stmt* |

| *AExp* | ::= | ... |
| *BExp* | ::= | ... |

```
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

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

- 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)

An Interpreter

$$\text{eval}(n, E) \quad \stackrel{\text{def}}{=} \quad n$$

$$\text{eval}(x, E) \quad \stackrel{\text{def}}{=} \quad E(x) \quad \text{lookup } x \text{ in } E$$

$$\text{eval}(a_1 + a_2, E) \quad \stackrel{\text{def}}{=} \quad \text{eval}(a_1, E) + \text{eval}(a_2, E)$$

$$\text{eval}(a_1 - a_2, E) \quad \stackrel{\text{def}}{=} \quad \text{eval}(a_1, E) - \text{eval}(a_2, E)$$

$$\text{eval}(a_1 * a_2, E) \quad \stackrel{\text{def}}{=} \quad \text{eval}(a_1, E) * \text{eval}(a_2, E)$$

$$\text{eval}(a_1 = a_2, E) \quad \stackrel{\text{def}}{=} \quad \text{eval}(a_1, E) = \text{eval}(a_2, E)$$

$$\text{eval}(a_1 \mathbin{!} = a_2, E) \quad \stackrel{\text{def}}{=} \quad \neg(\text{eval}(a_1, E) = \text{eval}(a_2, E))$$

$$\text{eval}(a_1 < a_2, E) \quad \stackrel{\text{def}}{=} \quad \text{eval}(a_1, E) < \text{eval}(a_2, E)$$

An Interpreter (2)

$\text{eval}(\text{skip}, E) \quad \overset{\text{def}}{=} \quad E$

$\text{eval}(x := a, E) \quad \overset{\text{def}}{=} \quad E(x \mapsto \text{eval}(a, E))$

$\text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) \overset{\text{def}}{=}$
$\qquad\qquad \text{if } \text{eval}(b, E) \text{ then } \text{eval}(cs_1, E)$
$\qquad\qquad\qquad\qquad \text{else } \text{eval}(cs_2, E)$

$\text{eval}(\text{while } b \text{ do } cs, E) \overset{\text{def}}{=}$
$\qquad\qquad \text{if } \text{eval}(b, E)$
$\qquad\qquad \text{then } \text{eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E))$
$\qquad\qquad \text{else } E$

$\text{eval}(\text{write } x, E) \quad \overset{\text{def}}{=} \quad \{ \text{ println}(E(x)) \; ; \; E \}$

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
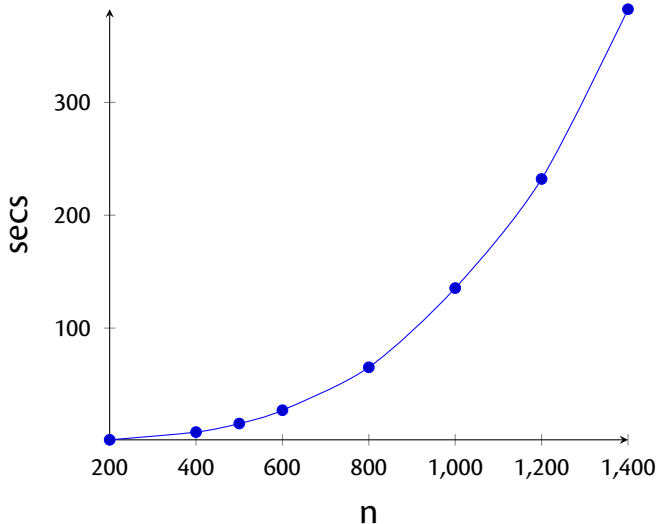
Java Virtual Machine

- 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
- source languages: C, C++, Rust, Go, Swift
- target CPUs: x86, ARM, PowerPC, …