

# Compilers and Formal Languages (6)

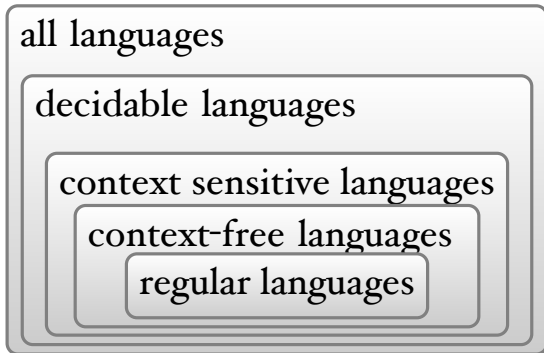
Email: christian.urban at kcl.ac.uk

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

$$I :: \textit{rest} \Rightarrow \{(I, \textit{rest})\}$$

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

Alternative parser (code  $p \parallel q$ )

- 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

$$\{(f(o_I), u_I) \mid (o_I, u_I) \in p(\text{input})\}$$

# Types of Parsers

- **Sequencing:** if  $p$  returns results of type  $T$ , and  $q$  returns 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  $q$  returns 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  $p \parallel q$  returns results of type

$$T$$



# Types of Parsers

- **Sequencing:** if  $p$  returns results of type  $T$ , and  $q$  returns 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  $p \parallel q$  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

$$S$$

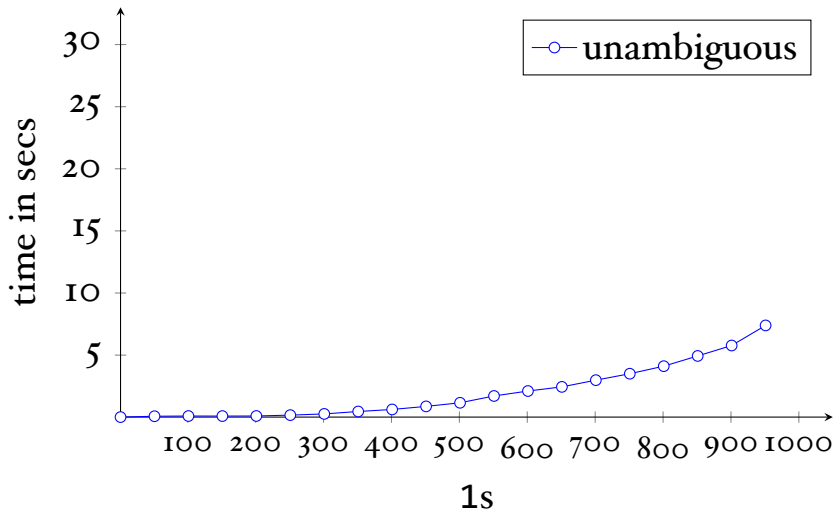
# Two Grammars

Which languages are recognised by the following two grammars?

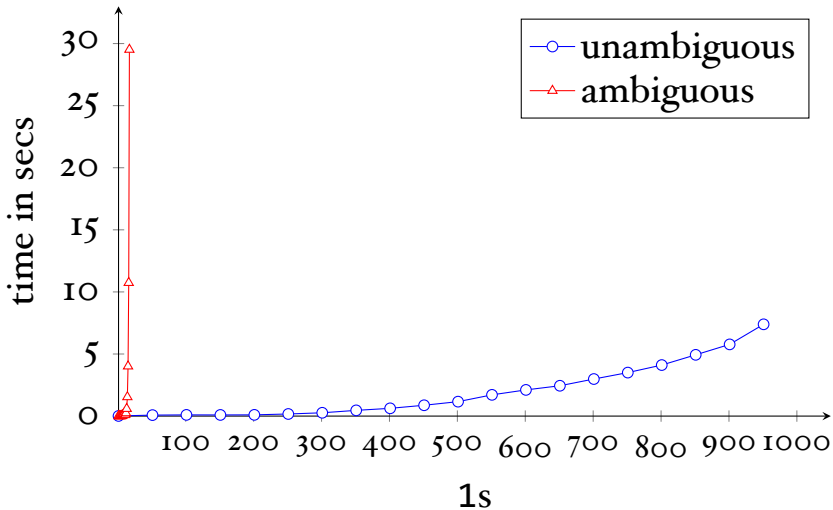
$$S \rightarrow \begin{array}{l} I \cdot S \cdot S \\ | \\ \epsilon \end{array}$$

$$U \rightarrow \begin{array}{l} I \cdot U \\ | \\ \epsilon \end{array}$$

# Ambiguous Grammars



# Ambiguous Grammars



# Arithmetic Expressions

A grammar for arithmetic expressions and numbers:

$$\begin{aligned} E &\rightarrow E \cdot + \cdot E \mid E \cdot * \cdot E \mid (\cdot E \cdot) \mid N \\ N &\rightarrow N \cdot N \mid 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

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:

$$\begin{aligned} E &\rightarrow E \cdot + \cdot E \mid E \cdot * \cdot E \mid (\cdot E \cdot) \mid N \\ N &\rightarrow N \cdot N \mid 0 \mid 1 \mid \dots \mid 9 \end{aligned}$$

Unfortunately it is left-recursive (and ambiguous).

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

# Numbers

$$N \rightarrow N \cdot N \mid 0 \mid 1 \mid \dots \mid 9$$

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

$$N \rightarrow 0 \cdot N \mid 1 \cdot N \mid \dots \mid 0 \mid 1 \mid \dots \mid 9$$

# Removing Left-Recursion

The rule for numbers is directly left-recursive:

$$N \rightarrow N \cdot N \mid 0 \mid 1 \quad (\dots)$$

Translate

$$\begin{array}{l} N \rightarrow N \cdot \alpha \\ \quad \mid \beta \end{array} \quad \Rightarrow \quad \begin{array}{l} N \rightarrow \beta \cdot N' \\ N' \rightarrow \alpha \cdot N' \\ \quad \mid \epsilon \end{array}$$



# Removing Left-Recursion

The rule for numbers is directly left-recursive:

$$N \rightarrow N \cdot N \mid 0 \mid 1 \quad (\dots)$$

Translate

$$\begin{array}{l} N \rightarrow N \cdot \alpha \\ \quad \mid \beta \end{array} \quad \Rightarrow \quad \begin{array}{l} N \rightarrow \beta \cdot N' \\ N' \rightarrow \alpha \cdot N' \\ \quad \mid \epsilon \end{array}$$

Which means

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

# Operator Precedences

To disambiguate

$$E \rightarrow E \cdot + \cdot E \mid E \cdot * \cdot E \mid (\cdot E \cdot) \mid N$$

Decide on how many precedence levels, say  
highest for  $()$ , medium for  $*$ , lowest for  $+$

$$\begin{aligned} E_{low} &\rightarrow E_{med} \cdot + \cdot E_{low} \mid E_{med} \\ E_{med} &\rightarrow E_{hi} \cdot * \cdot E_{med} \mid E_{hi} \\ E_{hi} &\rightarrow (\cdot E_{low} \cdot) \mid N \end{aligned}$$

# Operator Precedences

To disambiguate

$$E \rightarrow E \cdot + \cdot E \mid E \cdot * \cdot E \mid (\cdot E \cdot) \mid N$$

Decide on how many precedence levels, say  
highest for  $()$ , medium for  $*$ , lowest for  $+$

$$\begin{aligned} E_{low} &\rightarrow E_{med} \cdot + \cdot E_{low} \mid E_{med} \\ E_{med} &\rightarrow E_{hi} \cdot * \cdot E_{med} \mid E_{hi} \\ E_{hi} &\rightarrow (\cdot E_{low} \cdot) \mid N \end{aligned}$$

What happens with  $1 + 3 + 4$ ?

# Chomsky Normal Form

All rules must be of the form

$$A \rightarrow a$$

or

$$A \rightarrow B \cdot C$$

No rule can contain  $\epsilon$ .

# $\epsilon$ -Removal

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

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

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

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

# $\epsilon$ -Removal

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

$$\begin{aligned} N &\rightarrow o \cdot N' \mid I \cdot N' \\ N' &\rightarrow N \cdot N' \mid \epsilon \end{aligned}$$

$$\begin{aligned} N &\rightarrow o \cdot N' \mid I \cdot N' \mid o \mid I \\ N' &\rightarrow N \cdot N' \mid N \mid \epsilon \end{aligned}$$

$$\begin{aligned} N &\rightarrow o \cdot N' \mid I \cdot N' \mid o \mid I \\ N' &\rightarrow N \cdot N' \mid N \end{aligned}$$

$$N \rightarrow o \cdot N \mid I \cdot N \mid o \mid I$$

# CYK Algorithm

If grammar is in Chomsky normalform ...

$S \rightarrow N \cdot P$

$P \rightarrow V \cdot N$

$N \rightarrow N \cdot N$

$N \rightarrow \text{students} \mid \text{Jeff} \mid \text{geometry} \mid \text{trains}$

$V \rightarrow \text{trains}$

Jeff trains geometry students

# CYK Algorithm

- fastest possible algorithm for recognition problem
- runtime is  $O(n^3)$
- grammars need to be transferred into CNF



# The Goal of this Course

## Write a Compiler



We have lexer and 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

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

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

# 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  $y$
- $\text{eval}(\text{stmt}, \text{env})$

# Interpreter

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

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

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

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

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

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

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

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

# Interpreter (2)

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

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

$$\begin{aligned} \text{eval}(\text{if } b \text{ then } cs_1 \text{ else } cs_2, E) &\stackrel{\text{def}}{=} \\ &\text{if } \text{eval}(b, E) \text{ then } \text{eval}(cs_1, E) \\ &\text{else } \text{eval}(cs_2, E) \end{aligned}$$

$$\begin{aligned} \text{eval}(\text{while } b \text{ do } cs, E) &\stackrel{\text{def}}{=} \\ &\text{if } \text{eval}(b, E) \\ &\text{then } \text{eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) \\ &\text{else } E \end{aligned}$$

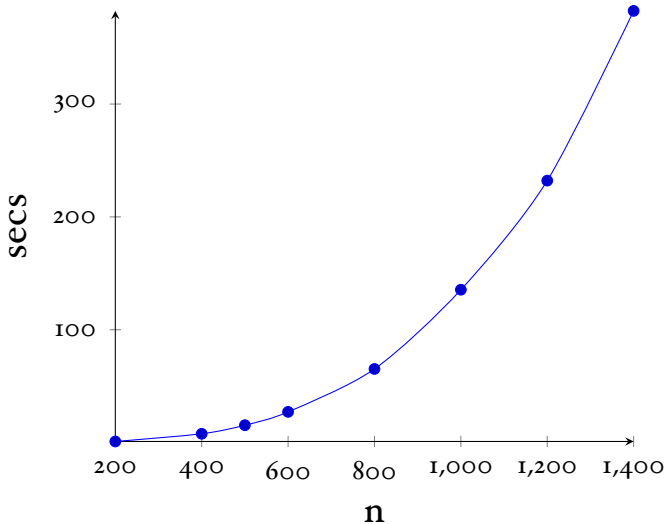
$$\text{eval}(\text{write } x, E) \stackrel{\text{def}}{=} \{ \text{println}(E(x)) ; E \}$$

# 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
- many languages take advantage of JVM's infrastructure (JRE)
- is garbage collected  $\Rightarrow$  no buffer overflows
- some languages compile to the JVM: Scala, Clojure...