Automata and Formal Languages (6)

Email:	christian.urban at kcl.ac.uk
Office:	S1.27 (1st floor Strand Building)
Slides:	KEATS (also home work is there)

Regular Languages

While regular expressions are very useful for lexing, there is no regular expression that can recognise the language $a^n b^n$.

$$(((()()))()) \quad \text{vs.} \quad (((()()))()))$$

Grammars

A (context-free) grammar G consists of

- a finite set of nonterminal symbols (upper case)
- a finite terminal symbols or tokens (lower case)
- a start symbol (which must be a nonterminal)
- a set of rules

$$A \rightarrow \text{rhs}$$

where rhs are sequences involving terminals and nonterminals, including the empty sequence ϵ.

Grammars

A (context-free) grammar G consists of

- a finite set of nonterminal symbols (upper case)
- a finite terminal symbols or tokens (lower case)
- a start symbol (which must be a nonterminal)
- a set of rules

$$A \rightarrow \text{rhs}$$

where rhs are sequences involving terminals and nonterminals, including the empty sequence ϵ.

We also allow rules

$$A \rightarrow \text{rhs}_1|\text{rhs}_2|\ldots$$

Palindromes

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & a \cdot S \cdot a \\ S & \rightarrow & b \cdot S \cdot b \end{array}$$

Palindromes

$$\begin{array}{lcl} S & \rightarrow & \epsilon \\ S & \rightarrow & a \cdot S \cdot a \\ S & \rightarrow & b \cdot S \cdot b \end{array}$$

or

$$S \rightarrow \epsilon \mid a \cdot S \cdot a \mid b \cdot S \cdot b$$

Arithmetic Expressions

$$
\begin{array}{lcl}
E & \rightarrow & num\_token \\
E & \rightarrow & E \cdot + \cdot E \\
E & \rightarrow & E \cdot - \cdot E \\
E & \rightarrow & E \cdot * \cdot E \\
E & \rightarrow & (\cdot E \cdot)
\end{array}
$$

Arithmetic Expressions

$$
\begin{array}{lcl}
E & \rightarrow & num\_token \\
E & \rightarrow & E \cdot + \cdot E \\
E & \rightarrow & E \cdot - \cdot E \\
E & \rightarrow & E \cdot * \cdot E \\
E & \rightarrow & (\cdot E \cdot)
\end{array}
$$

1 + 2 * 3 + 4

A CFG Derivation

1. Begin with a string containing only the start symbol, say S
2. Replace any nonterminal X in the string by the right-hand side of some production $X \rightarrow \text{rhs}$
3. Repeat 2 until there are no nonterminals

$$S \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots \rightarrow \ldots$$

Example Derivation

$$S \;\rightarrow\; \epsilon \mid a \cdot S \cdot a \mid b \cdot S \cdot b$$

$$\begin{aligned} S &\rightarrow aSa \\ &\rightarrow abSba \\ &\rightarrow abaSaba \\ &\rightarrow abaaba \end{aligned}$$

Example Derivation

$$
\begin{array}{lcl}
E & \rightarrow & num\_token \\
E & \rightarrow & E \cdot + \cdot E \\
E & \rightarrow & E \cdot - \cdot E \\
E & \rightarrow & E \cdot * \cdot E \\
E & \rightarrow & (\cdot E \cdot)
\end{array}
$$

$$
\begin{array}{ll}
E \rightarrow & E * E \\
\rightarrow & E + E * E \\
\rightarrow & E + E * E + E \\
\rightarrow^{+} & 1 + 2 * 3 + 4
\end{array}
$$

Example Derivation

$$
\begin{array}{lcl}
E & \rightarrow & num\_token \\
E & \rightarrow & E \cdot + \cdot E \\
E & \rightarrow & E \cdot - \cdot E \\
E & \rightarrow & E \cdot * \cdot E \\
E & \rightarrow & (\cdot E \cdot)
\end{array}
$$

$$
\begin{array}{ll}
E \rightarrow & E * E \\
\rightarrow & E + E * E \\
\rightarrow & E + E * E + E \\
\rightarrow^{+} & 1 + 2 * 3 + 4
\end{array}
\qquad
\begin{array}{ll}
E \rightarrow & E + E \\
\rightarrow & E + E + E \\
\rightarrow & E + E * E + E \\
\rightarrow^{+} & 1 + 2 * 3 + 4
\end{array}
$$

Language of a CFG

Let G be a context-free grammar with start symbol S. Then the language $L(G)$ is:

$$\{c_1 \ldots c_n \mid \forall i.\ c_i \in T \wedge S \rightarrow^* c_1 \ldots c_n\}$$

Language of a CFG

Let G be a context-free grammar with start symbol S. Then the language $L(G)$ is:

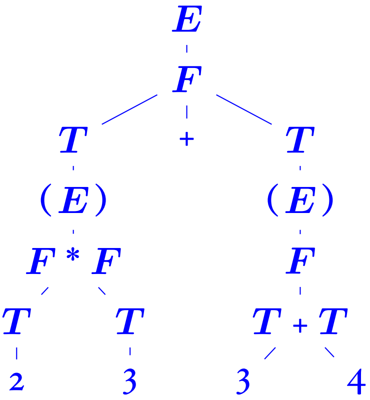
$$\{c_1 \ldots c_n \mid \forall i.\ c_i \in T \wedge S \rightarrow^* c_1 \ldots c_n\}$$

- Terminals, because there are no rules for replacing them.
- Once generated, terminals are “permanent”.
- Terminals ought to be tokens of the language (but can also be strings).

Parse Trees

$$
\begin{array}{lcl}
E & \rightarrow & F \mid F \cdot * \cdot F \\
F & \rightarrow & T \mid T \cdot + \cdot T \mid T \cdot - \cdot T \\
T & \rightarrow & num\_token \mid (\cdot E \cdot)
\end{array}
$$

(2*3)+(3+4)



Arithmetic Expressions

$$
\begin{array}{lcl}
E & \rightarrow & num\_token \\
E & \rightarrow & E \cdot + \cdot E \\
E & \rightarrow & E \cdot - \cdot E \\
E & \rightarrow & E \cdot * \cdot E \\
E & \rightarrow & (\cdot E \cdot)
\end{array}
$$

Arithmetic Expressions

$$
\begin{array}{lcl}
E & \rightarrow & num\_token \\
E & \rightarrow & E \cdot + \cdot E \\
E & \rightarrow & E \cdot - \cdot E \\
E & \rightarrow & E \cdot * \cdot E \\
E & \rightarrow & (\cdot E \cdot)
\end{array}
$$

A CFG is left-recursive if it has a nonterminal E such that $E \rightarrow^{+} E \cdot \ldots$

Ambiguous Grammars

A grammar is ambiguous if there is a string that has at least two different parse trees.

$$\begin{array}{lcl} E & \rightarrow & num\_token \\ E & \rightarrow & E \cdot + \cdot E \\ E & \rightarrow & E \cdot - \cdot E \\ E & \rightarrow & E \cdot * \cdot E \\ E & \rightarrow & (\cdot E \cdot) \end{array}$$

```
1 + 2 * 3 + 4
```

Dangling Else

Another ambiguous grammar:

$$
\begin{array}{rcl}
E & \rightarrow & \text{if } E \text{ then } E \\
  & | & \text{if } E \text{ then } E \text{ else } E \\
  & | & \ldots
\end{array}
$$

```
if a then if x then y else c
```

Parser Combinators

Parser combinators:

$$\underbrace{\text{list of tokens}}_{\text{input}} \Rightarrow \underbrace{\text{set of (parsed input, unparsed input)}}_{\text{output}}$$

- sequencing
- alternative
- semantic action

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:
  ((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

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

Function parser (code $p \Rightarrow f$)

- apply p producing a set of pairs
- then apply the function f to each first component

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

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

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

Multiplication

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

Semantic Actions

Addition

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

Multiplication

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

Parenthesis

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

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 $p \;||\; q$ 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: string
- output: set of (output_type, string)

Input Types of Parsers

- input: string
- output: set of (output_type, string)

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 lexers are better when whitespaces or comments need to 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

```scala
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)
}
```

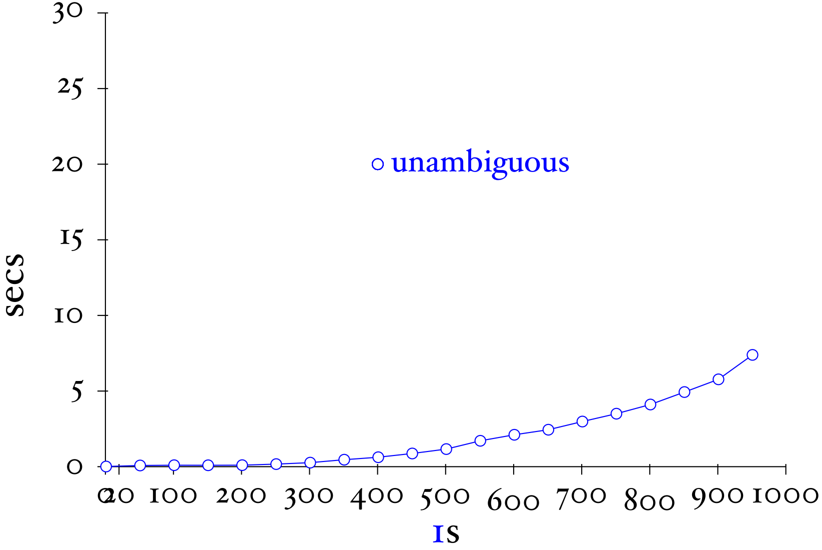
Two Grammars

Which languages are recognised by the following two grammars?

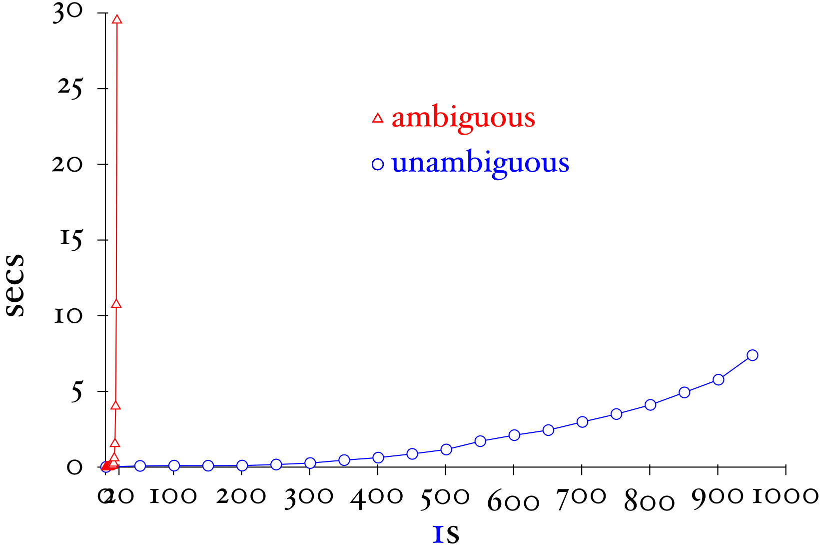
$$
\begin{array}{rcl}
S & \rightarrow & 1 \cdot S \cdot S \\
 & | & \epsilon
\end{array}
$$

$$
\begin{array}{rcl}
U & \rightarrow & 1 \cdot U \\
 & | & \epsilon
\end{array}
$$

Ambiguous Grammars



Ambiguous Grammars



While-Language

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

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

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

- the interpreter has to record the value of x before assigning a value to y
- eval(stmt, env)

Chomsky Normal Form

All rules must be of the form

$$A \rightarrow a$$

or

$$A \rightarrow B \cdot C$$

CYK Algorithm

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

`Jeff trains geometry students`

CYK Algorithm

- runtime is $\boldsymbol{O}(\boldsymbol{n}^3)$
- grammars need to be transferred into CNF