Compilers and Formal Languages

Email: christian.urban at kcl.ac.uk
Slides & Progs: KEATS (also homework is there)

1 Introduction, Languages	6 While-Language
2 Regular Expressions, Derivatives	7 Compilation, JVM
3 Automata, Regular Languages	8 Compiling Functional Languages
4 Lexing, Tokenising	9 Optimisations
5 Grammars, Parsing	10 LLVM

Lexer, Parser

Today a parser.

What Parsing is Not

Usually parsing does not check semantic correctness, e.g.

- whether a function is not used before it is defined
- whether a function has the correct number of arguments or are of correct type
- whether a variable can be declared twice in a scope

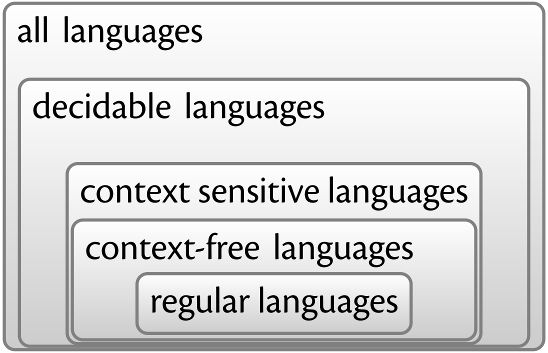
Regular Languages

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

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

So we cannot find out with regular expressions whether parentheses are matched or unmatched. Also regular expressions are not recursive, e.g. $(1 + 2) + 3$.

Hierarchy of Languages



CF Grammars

A **context-free grammar** G consists of

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

$$A ::= rhs$$

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

CF Grammars

A **context-free grammar** G consists of

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

$$A ::= rhs$$

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

We also allow rules

$$A ::= rhs_1 | rhs_2 | \ldots$$

Palindromes

A grammar for palindromes over the alphabet $\{a, b\}$:

$$S ::= a \cdot S \cdot a$$

$$S ::= b \cdot S \cdot b$$

$$S ::= a$$

$$S ::= b$$

$$S ::= \epsilon$$

Palindromes

A grammar for palindromes over the alphabet $\{a, b\}$:

$$S ::= a \cdot S \cdot a$$
$$S ::= b \cdot S \cdot b$$
$$S ::= a$$
$$S ::= b$$
$$S ::= \epsilon$$

or

$$S ::= a \cdot S \cdot a \mid b \cdot S \cdot b \mid a \mid b$$

Palindromes

A grammar for palindromes over the alphabet $\{a, b\}$:

$$S ::= a \cdot S \cdot a$$
$$S ::= b \cdot S \cdot b$$
$$S ::= a$$
$$S ::= b$$
$$S ::= \epsilon$$

or

$$S ::= a \cdot S \cdot a \mid b \cdot S \cdot b \mid a \mid b$$

Arithmetic Expressions

$$
\begin{aligned}
E ::= &\ num\_token \\
| &\ E \cdot + \cdot E \\
| &\ E \cdot - \cdot E \\
| &\ E \cdot * \cdot E \\
| &\ (\cdot E \cdot)
\end{aligned}
$$

Arithmetic Expressions

$$
\begin{aligned}
E ::= &\; num\_token \\
| &\; E \cdot + \cdot E \\
| &\; E \cdot - \cdot E \\
| &\; E \cdot * \cdot E \\
| &\; (\cdot E \cdot)
\end{aligned}
$$

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 $X ::= rhs$
3. Repeat 2 until there are no nonterminals left

$$S \to \ldots \to \ldots \to \ldots \to \ldots$$

Example Derivation

$$S ::= \epsilon \mid a \cdot S \cdot a \mid b \cdot S \cdot b$$

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

Example Derivation

$$\begin{aligned} E ::= &\ num\_token \\ &| \ E \cdot + \cdot E \\ &| \ E \cdot - \cdot E \\ &| \ E \cdot * \cdot E \\ &| \ (\cdot E \cdot) \end{aligned}$$

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

Example Derivation

$$
\begin{aligned}
E ::= &\ \mathit{num\_token} \\
 |&\ E \cdot + \cdot E \\
 |&\ E \cdot - \cdot E \\
 |&\ E \cdot * \cdot E \\
 |&\ (\cdot E \cdot)
\end{aligned}
$$

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

Context Sensitive Grammars

It is much harder to find out whether a string is parsed by a context sensitive grammar:

$$
\begin{aligned}
S &::= bSAA \mid \epsilon \\
A &::= a \\
bA &::= Ab
\end{aligned}
$$

Context Sensitive Grammars

It is much harder to find out whether a string is parsed by a context sensitive grammar:

$$
\begin{aligned}
S &::= bSAA \mid \epsilon \\
A &::= a \\
bA &::= Ab
\end{aligned}
$$

$$S \rightarrow \ldots \rightarrow^{?} ababaa$$

Context Sensitive Grammars

It is much harder to find out whether a string is parsed by a context sensitive grammar:

$$
\begin{aligned}
S &::= bSAA \mid \epsilon \\
A &::= a \\
bA &::= Ab
\end{aligned}
$$

$$S \rightarrow \ldots \rightarrow^{?} ababaa$$

```
Time flies like an arrow;
fruit flies like bananas.
```

Language of a CFG

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

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

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).

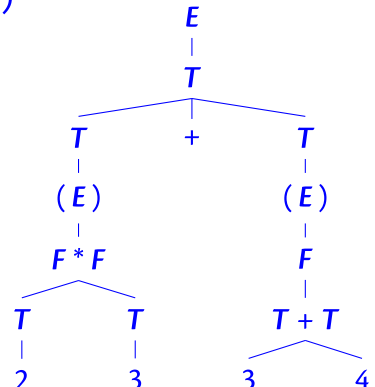
Parse Trees

$E ::= T \mid T \cdot + \cdot E \mid T \cdot - \cdot E$

$T ::= F \mid F \cdot * \cdot T$

$F ::= num\_token \mid (\cdot E \cdot)$

(2*3)+(3+4)



Arithmetic Expressions

$$
\begin{aligned}
E ::= &\ num\_token\\
\mid &\ E \cdot + \cdot E\\
\mid &\ E \cdot - \cdot E\\
\mid &\ E \cdot * \cdot E\\
\mid &\ (\cdot E \cdot)
\end{aligned}
$$

Arithmetic Expressions

$$
\begin{aligned}
E ::= \; & num\_token \\
| \; & E \cdot + \cdot E \\
| \; & E \cdot - \cdot E \\
| \; & E \cdot * \cdot E \\
| \; & (\cdot E \cdot)
\end{aligned}
$$

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.

$$
\begin{aligned}
E ::= &\ \mathit{num\_token} \\
 |\ & E \cdot + \cdot E \\
 |\ & E \cdot - \cdot E \\
 |\ & E \cdot * \cdot E \\
 |\ & (\cdot E \cdot)
\end{aligned}
$$

```
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

One of the simplest ways to implement a parser, see

https://vimeo.com/142341803

Parser combinators:

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

- atomic parsers
- sequencing
- alternative
- semantic action

Atomic parsers, for example, number tokens

$$\texttt{Num(123)} :: \mathit{rest} \;\Rightarrow\; \{(\texttt{Num(123)}, \mathit{rest})\}$$

- you consume one or more token 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 part
- 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

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

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

Input Types of Parsers

- input: token list
- output: set of (output_type, token list)

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 using lexers is better because whitespaces or comments can 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

```
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
}
```

```scala
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)
}
```

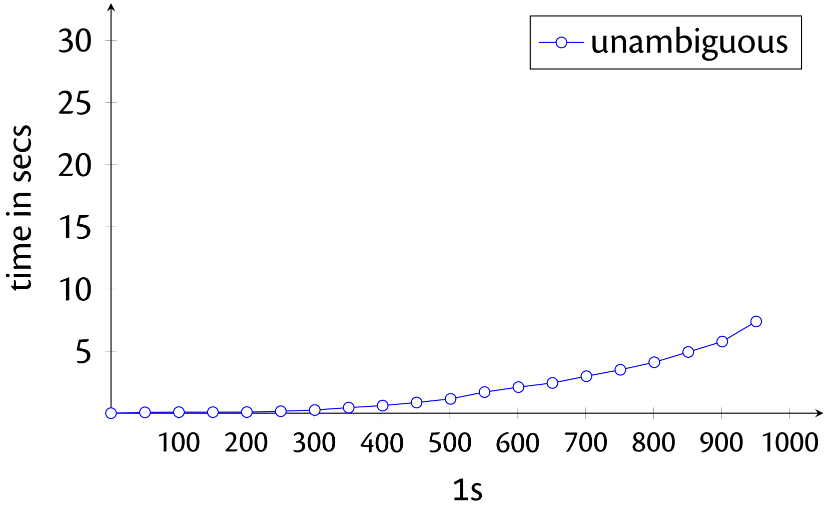
Two Grammars

Which languages are recognised by the following two grammars?

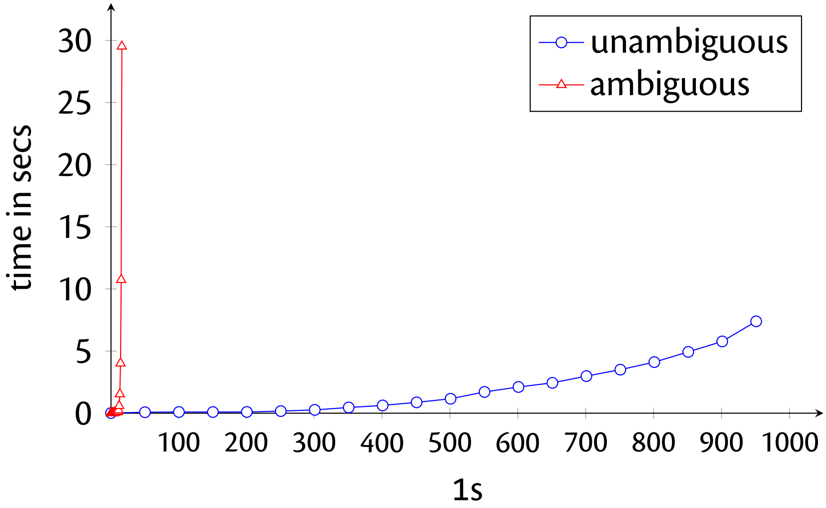
$$
\begin{array}{lcl}
S & \rightarrow & 1 \cdot S \cdot S \\
 & | & \epsilon
\end{array}
$$

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

Ambiguous Grammars



Ambiguous Grammars



While-Language

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

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

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

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

Interpreter

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

Interpreter (2)

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

$$
\begin{array}{l}
\quad\quad \text{if eval}(b, E) \text{ then eval}(cs_1, E) \\
\quad\quad \phantom{\text{if eval}(b, E)} \text{ else eval}(cs_2, E)
\end{array}
$$

$$
\text{eval}(\text{while } b \text{ do } cs, E) \stackrel{\text{def}}{=}
$$

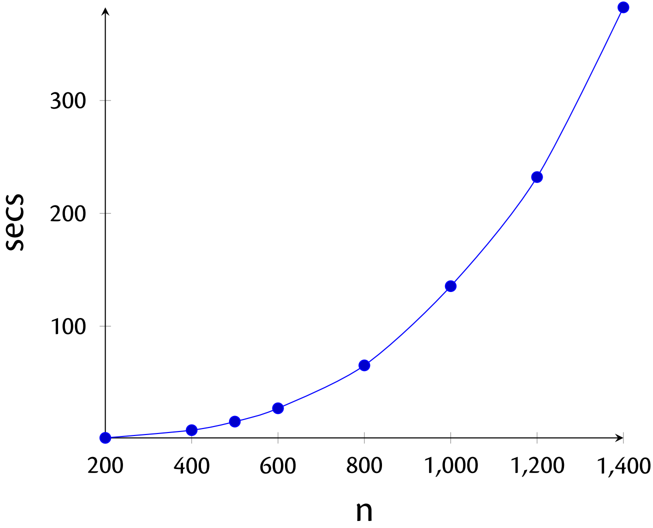
$$
\begin{array}{l}
\quad\quad \text{if eval}(b, E) \\
\quad\quad \text{then eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E)) \\
\quad\quad \text{else } E
\end{array}
$$

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

Test Program

??

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...