Compilers and Formal Languages (5)

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)

Last Week
Regexes and Values

Regular expressions and their corresponding values:

$$
\begin{array}{rcl}
r & ::= & \mathbf{0} \\
  & | & \mathbf{1} \\
  & | & c \\
  & | & r_1 \cdot r_2 \\
  & | & r_1 + r_2 \\
  & & \\
  & | & r^*
\end{array}
\qquad
\begin{array}{rcl}
v & ::= & \\
  &   & \mathit{Empty} \\
  & | & \mathit{Char}(c) \\
  & | & \mathit{Seq}(v_1, v_2) \\
  & | & \mathit{Left}(v) \\
  & | & \mathit{Right}(v) \\
  & | & \mathit{Stars}\,[v_1, \ldots\, v_n]
\end{array}
$$

r_1: $a \cdot (b \cdot c)$
r_2: $\mathbf{1} \cdot (b \cdot c)$
r_3: $(\mathbf{0} \cdot (b \cdot c)) + (\mathbf{1} \cdot c)$
r_4: $(\mathbf{0} \cdot (b \cdot c)) + ((\mathbf{0} \cdot c) + \mathbf{1})$

$$\begin{tikzcd}
r_1 \arrow[r, "der\ a"] \arrow[d] & r_2 \arrow[r, "der\ b"] \arrow[d] & r_3 \arrow[r, "der\ c"] \arrow[d] & r_4\ nullable \arrow[d, "mkeps"] \\
v_1 & v_2 \arrow[l, "inj\ a"] & v_3 \arrow[l, "inj\ b"] & v_4 \arrow[l, "inj\ c"]
\end{tikzcd}$$

v_1: $Seq(Char(a), Seq(Char(b), Char(c)))$
v_2: $Seq(Empty, Seq(Char(b), Char(c)))$
v_3: $Right(Seq(Empty, Char(c)))$
v_4: $Right(Right(Empty))$

$|v_1|$: abc
$|v_2|$: bc
$|v_3|$: c
$|v_4|$: $[]$

Simplification

- If we simplify after the derivative, then we are builing the value for the simplified regular expression, but *not* for the original regular expression.

$$\begin{tikzcd}
r_1 \arrow[r, "der\,a"] \arrow[d] & r_2 \arrow[r, "der\,b"] \arrow[d] & r_3 \arrow[r, "der\,c"] \arrow[d] & r_4 \; nullable \arrow[d, "mkeps"] \\
v_1 & v_2 \arrow[l, "inj\,a"] & v_3 \arrow[l, "inj\,b"] & v_4 \arrow[l, "inj\,c"]
\end{tikzcd}$$

$$(b \cdot c) + (\mathbf{0} + \mathbf{1}) \mapsto (b \cdot c) + \mathbf{1}$$

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$
- $nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$
- $der\, c\, (x : r) \stackrel{\text{def}}{=} (x : der\, c\, r)$
- $mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$
- $inj\, (x : r)\, c\, Rec(x, v) \stackrel{\text{def}}{=} Rec(x, inj\, r\, c\, v)$

Records

- new regex: $(x : r)$ new value: $Rec(x, v)$
- $nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$
- $der\, c\, (x : r) \stackrel{\text{def}}{=} (x : der\, c\, r)$
- $mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$
- $inj\, (x : r)\, c\, Rec(x, v) \stackrel{\text{def}}{=} Rec(x, inj\, r\, c\, v)$

for extracting subpatterns $(z : ((x : ab) + (y : ba))$

Environments

Obtaining the "recorded" parts of a value:

$$
\begin{array}{lcl}
env(Empty) & \stackrel{\text{def}}{=} & [] \\
env(Char(c)) & \stackrel{\text{def}}{=} & [] \\
env(Left(v)) & \stackrel{\text{def}}{=} & env(v) \\
env(Right(v)) & \stackrel{\text{def}}{=} & env(v) \\
env(Seq(v_1, v_2)) & \stackrel{\text{def}}{=} & env(v_1) \mathbin{@} env(v_2) \\
env(Stars[v_1, \ldots, v_n]) & \stackrel{\text{def}}{=} & env(v_1) \mathbin{@} \ldots \mathbin{@} env(v_n) \\
env(Rec(x : v)) & \stackrel{\text{def}}{=} & (x : |v|) :: env(v)
\end{array}
$$

While Tokens

$$
\begin{aligned}
\texttt{WHILE\_REGS} \stackrel{\text{def}}{=} \Big((&\texttt{"k"} : \texttt{KEYWORD}) + \\
(&\texttt{"i"} : \texttt{ID}) + \\
(&\texttt{"o"} : \texttt{OP}) + \\
(&\texttt{"n"} : \texttt{NUM}) + \\
(&\texttt{"s"} : \texttt{SEMI}) + \\
(&\texttt{"p"} : (\texttt{LPAREN} + \texttt{RPAREN})) + \\
(&\texttt{"b"} : (\texttt{BEGIN} + \texttt{END})) + \\
(&\texttt{"w"} : \texttt{WHITESPACE})\Big)^*
\end{aligned}
$$

```
"if true then then 42 else +"
```

```
KEYWORD(if),
WHITESPACE,
IDENT(true),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
KEYWORD(then),
WHITESPACE,
NUM(42),
WHITESPACE,
KEYWORD(else),
WHITESPACE,
OP(+)
```

```
"if true then then 42 else +"
```

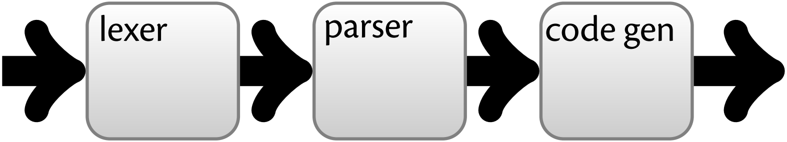
```
KEYWORD(if),
IDENT(true),
KEYWORD(then),
KEYWORD(then),
NUM(42),
KEYWORD(else),
OP(+)
```

Coursework 1: Submissions

- Scala (29)
- Haskell (1)
- Kotlin (1)
- Rust (1)

Please get in contact if you intend to do CW Strand 2. No zips please. Give definitions also on paper, if asked, and be truthful! BTW, simp can stay unchanged. Use `ders` for CW2, not `ders2`!

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

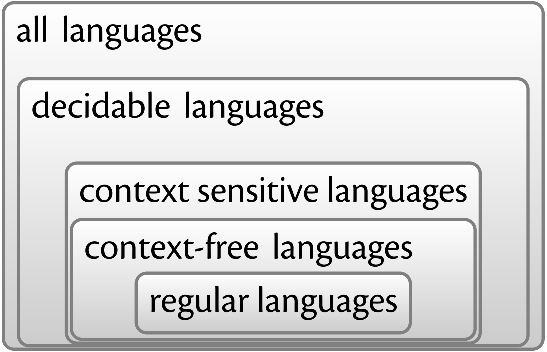
Regular Languages

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 $\{a, b\}$:

$$
\begin{aligned}
S &::= a \cdot S \cdot a\\
S &::= b \cdot S \cdot b\\
S &::= a\\
S &::= b\\
S &::= \epsilon
\end{aligned}
$$

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

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

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

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

Can you find the grammar rules for matched parentheses?

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

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

`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

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

Example Derivation

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

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

Example Derivation

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

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

Example Derivation

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

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

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:

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

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

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 $\mathbf{S}$. Then the language $L(G)$ is:

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

- 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).

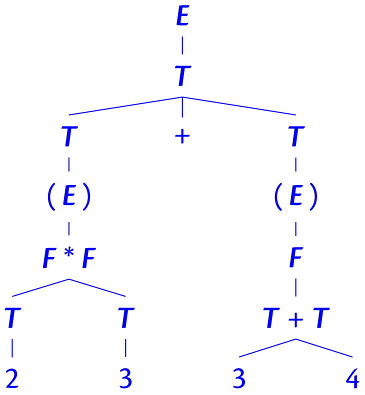
Parse Trees

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

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

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

(2*3)+(3+4)



Arithmetic Expressions

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

Arithmetic Expressions

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

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.

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

`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

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

- 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

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

Multiplication

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

Semantic Actions

Addition

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

Multiplication

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

Parenthesis

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

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)
}
```

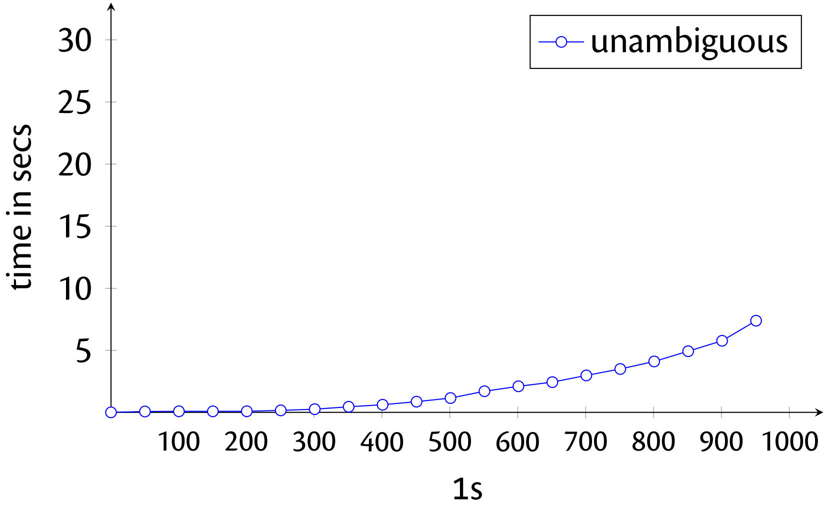
Two Grammars

Which languages are recognised by the following two grammars?

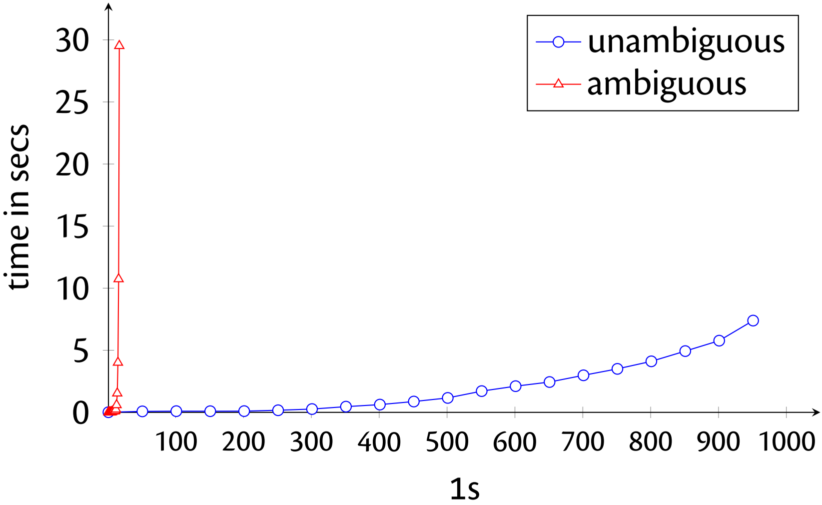
$$
\begin{array}{rcl}
S & \rightarrow & 1 \cdot S \cdot S \\
 & | & \epsilon
\end{array}
$$

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

Ambiguous Grammars



Ambiguous Grammars



While-Language

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

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

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

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

Interpreter

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

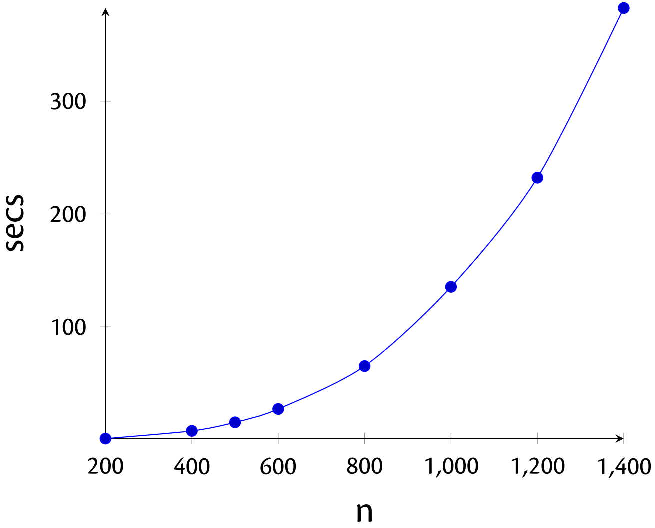
Interpreter (2)

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

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