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)

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. BTW, simp can stay unchanged. Use `ders` for CW2, not `ders2`!
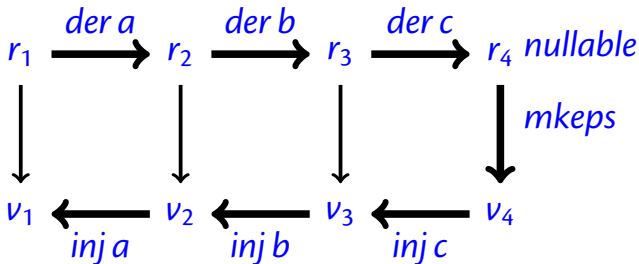
Last Week
Regexes and Values

Regular expressions and their corresponding values:

$$r ::= \mathbf{0}$$
$$| \quad \mathbf{1}$$
$$| \quad c$$
$$| \quad r_1 \cdot r_2$$
$$| \quad r_1 + r_2$$
$$| \quad r^*$$

$$v ::=$$
$$Empty$$
$$| \quad Char(c)$$
$$| \quad Seq(v_1, v_2)$$
$$| \quad Left(v)$$
$$| \quad Right(v)$$
$$| \quad Stars[v_1, \ldots v_n]$$

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

r_1 →$der\,a$→ r_2 →$der\,b$→ r_3 →$der\,c$→ r_4 *nullable*

mkeps

v_1 ←$inj\,a$← v_2 ←$inj\,b$← v_3 ←$inj\,c$← v_4

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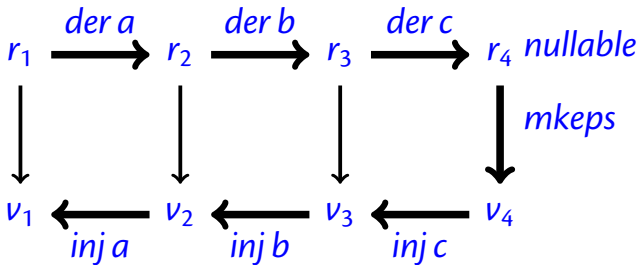
$$(b \cdot c) + (\mathbf{0} + \mathbf{1}) \mapsto (b \cdot c) + \mathbf{1}$$

Records

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

Records

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

- $nullable(x : r) \stackrel{\text{def}}{=} nullable(r)$

- $der\, c\, (x : r) \stackrel{\text{def}}{=} der\, c\, r$

- $mkeps(x : r) \stackrel{\text{def}}{=} Rec(x, mkeps(r))$

- $inj\, (x : r)\, c\, v \stackrel{\text{def}}{=} Rec(x, inj\, r\, c\, v)$

Records

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

- $nullable(x : r) \overset{\text{def}}{=} nullable(r)$

- $der\, c\, (x : r) \overset{\text{def}}{=} der\, c\, r$

- $mkeps(x : r) \overset{\text{def}}{=} Rec(x, mkeps(r))$

- $inj\, (x : r)\, c\, v \overset{\text{def}}{=} Rec(x, inj\, r\, c\, v)$

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

Environments

Obtaining the "recorded" parts of a value:

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

While Tokens

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

```
        "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(+)
```

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
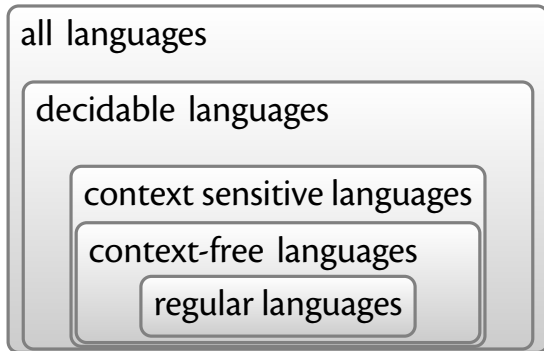- whether a variable can be declared twice in a scope

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

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

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

$E ::=$ *num_token*

$\quad | \; E \cdot + \cdot E$

$\quad | \; E \cdot - \cdot E$

$\quad | \; E \cdot * \cdot E$

$\quad | \; (\cdot E \cdot)$

Arithmetic Expressions

$$E ::= num\_token$$
$$| \; E \cdot + \cdot E$$
$$| \; E \cdot - \cdot E$$
$$| \; E \cdot * \cdot E$$
$$| \; (\cdot E \cdot)$$

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

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

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

Example Derivation

$$E ::= num\_token$$
$$| \; E \cdot + \cdot E$$
$$| \; E \cdot - \cdot E$$
$$| \; E \cdot * \cdot E$$
$$| \; (\cdot E \cdot)$$

$$E \rightarrow \quad E * E$$
$$\rightarrow \quad E + E * E$$
$$\rightarrow \quad E + E * E + E$$
$$\rightarrow^+ \; 1 + 2 * 3 + 4$$

Example Derivation

$$E ::= num\_token$$
$$\mid E \cdot + \cdot E$$
$$\mid E \cdot - \cdot E$$
$$\mid E \cdot * \cdot E$$
$$\mid (\cdot E \cdot)$$

$$
\begin{aligned}
E &\to \; E * E \\
&\to \; E + E * E \\
&\to \; E + E * E + E \\
&\to^+ \; 1 + 2 * 3 + 4
\end{aligned}
\qquad
\begin{aligned}
E &\to \; E + E \\
&\to \; E + E + E \\
&\to \; E + E * E + E \\
&\to^+ \; 1 + 2 * 3 + 4
\end{aligned}
$$

Context Sensitive Grammars

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

$$S ::= bSAA \mid \epsilon$$

$$A ::= a$$

$$bA ::= Ab$$

Context Sensitive Grammars

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

$$S \;::=\; bSAA \mid \epsilon$$

$$A \;::=\; a$$

$$bA \;::=\; Ab$$

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

Context Sensitive Grammars

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

$$S ::= bSAA \mid \epsilon$$

$$A ::= a$$

$$bA ::= Ab$$

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

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

Language of a CFG

Let G be a context-free grammar with start symbol 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:

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

- Terminals, because there are no rules for replacing them.

- Once generated, terminals are "permanent".

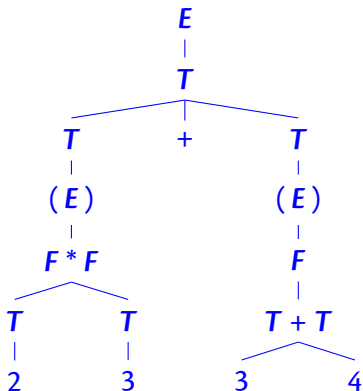- Terminals ought to be tokens of the language
  (but can also be strings).

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

$E ::= \textit{num\_token}$

$\quad | \; E \cdot + \cdot E$

$\quad | \; E \cdot - \cdot E$

$\quad | \; E \cdot * \cdot E$

$\quad | \; (\cdot E \cdot)$

Arithmetic Expressions

$$E ::= num\_token$$
$$| \; E \cdot + \cdot E$$
$$| \; E \cdot - \cdot E$$
$$| \; E \cdot * \cdot E$$
$$| \; (\cdot E \cdot)$$

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.

$$E ::= num\_token$$
$$| \; E \cdot + \cdot E$$
$$| \; E \cdot - \cdot E$$
$$| \; E \cdot * \cdot E$$
$$| \; (\cdot E \cdot)$$

1 + 2 * 3 + 4

'Dangling' Else

Another ambiguous grammar:

$$E \rightarrow \text{if } E \text{ then } E$$
$$| \quad \text{if } E \text{ then } E \text{ else } E$$
$$| \quad \ldots$$

```
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

$$\texttt{Num(123)} :: \mathit{rest} \implies \big\{ \big(\texttt{Num(123)}, \mathit{rest} \big) \big\}$$

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

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 part
- 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

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

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 $p \mathbin{\|} q$ 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: 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

```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
}
```

```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)
}
```
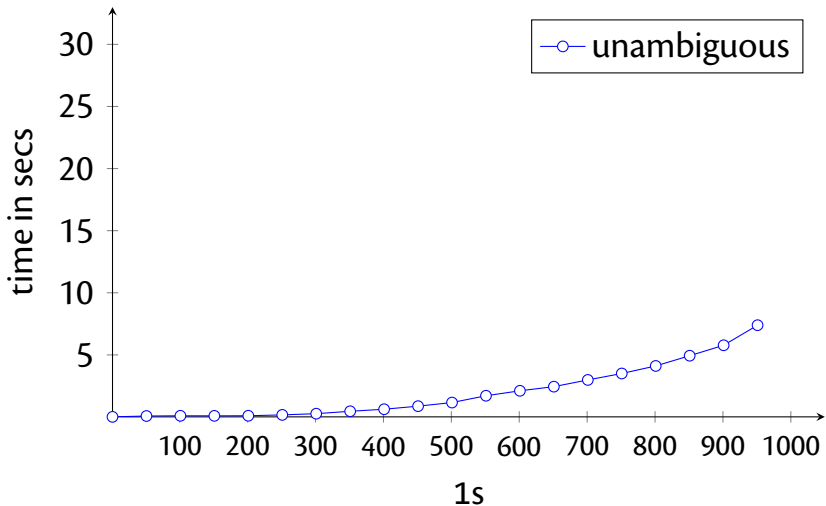
Two Grammars

Which languages are recognised by the following two grammars?

$$S \rightarrow 1 \cdot S \cdot S$$
$$\mid \epsilon$$

$$U \rightarrow 1 \cdot U$$
$$\mid \epsilon$$

Ambiguous Grammars

Ambiguous Grammars
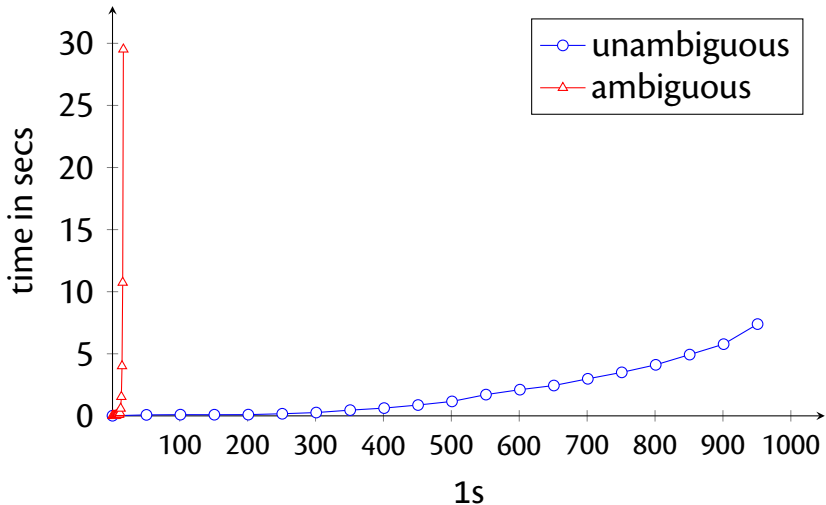
While-Language

$Stmt$ ::= skip

     | Id := $AExp$

     | if $BExp$ then $Block$ else $Block$

     | while $BExp$ do $Block$

$Stmts$ ::= $Stmt$; $Stmts$

     | $Stmt$

$Block$ ::= { $Stmts$ }

     | $Stmt$

$AExp$ ::= ...

$BExp$ ::= ...

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

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

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

Interpreter

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

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

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

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

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

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

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

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

Interpreter (2)

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

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

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

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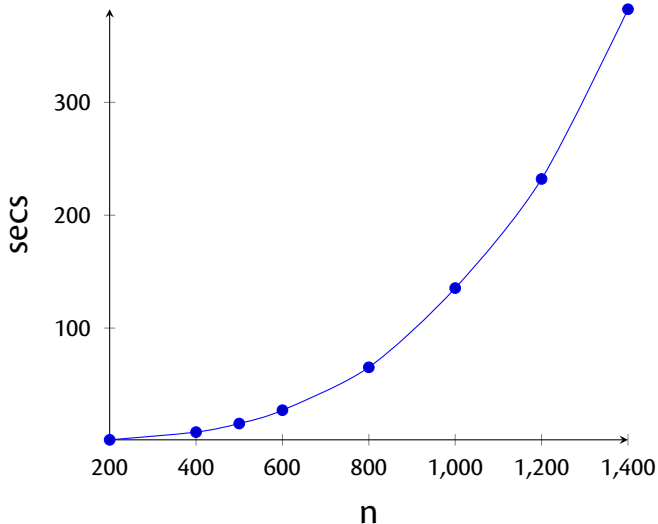$$\text{then } \text{eval}(\text{while } b \text{ do } cs, \text{eval}(cs, E))$$
$$\text{else } E$$

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

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...