Automata and Formal Languages (5)

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

Last Week
Regexes and Values

Regular expressions and their corresponding values:

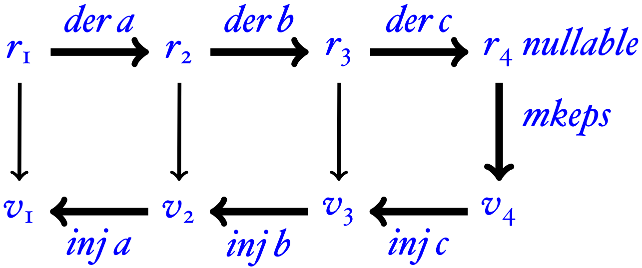
$$
\begin{array}{rcl}
r & ::= & \varnothing \\
  & \mid & \epsilon \\
  & \mid & c \\
  & \mid & r_1 \cdot r_2 \\
  & \mid & r_1 + r_2 \\
  & & \\
  & \mid & r^*
\end{array}
\qquad\qquad
\begin{array}{rcl}
v & ::= & \\
  & & \mathit{Empty} \\
  & \mid & \mathit{Char}(c) \\
  & \mid & \mathit{Seq}(v_1,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

r_1: $a \cdot (b \cdot c)$

r_2: $\epsilon \cdot (b \cdot c)$

r_3: $(\varnothing \cdot (b \cdot c)) + (\epsilon \cdot c)$

r_4: $(\varnothing \cdot (b \cdot c)) + ((\varnothing \cdot c) + \epsilon)$



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

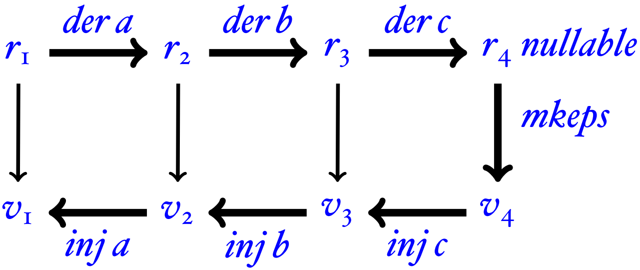
$|v_2|$: bc

$|v_3|$: 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.



$$(\varnothing \cdot (b \cdot c)) + ((\varnothing \cdot c) + \epsilon) \mapsto \epsilon$$

Records

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

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

Environments

Obtaining the "recorded" parts of a value:

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

While Tokens

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

”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(+)
```

Two Rules

- Longest match rule ("maximal munch rule"): The longest initial substring matched by any regular expression is taken as next token.
- Rule priority: For a particular longest initial substring, the first regular expression that can match determines the token.

- Regular expression for email addresses

$$(\text{name: } [a\text{-}z0\text{-}9\_\,.-]^{+})\cdot @ \cdot$$
$$(\text{domain: } [a\text{-}z0\text{-}9\,.-]^{+})\cdot . \cdot$$
$$(\text{top\_level: } [a\text{-}z\,.]^{\{2,6\}})$$

`christian.urban@kcl.ac.uk`

- result environment:

$$[(\mathit{name} : \texttt{christian.urban}),$$
$$(\mathit{domain} : \texttt{kcl}),$$
$$(\mathit{top\_level} : \texttt{ac.uk})]$$

Coursework

$$nullable([c_1 c_2 \ldots c_n]) \stackrel{\text{def}}{=} ?$$
$$nullable(r^+) \stackrel{\text{def}}{=} ?$$
$$nullable(r^?) \stackrel{\text{def}}{=} ?$$
$$nullable(r^{\{n,m\}}) \stackrel{\text{def}}{=} ?$$
$$nullable(\sim r) \stackrel{\text{def}}{=} ?$$

$$der\, c\, ([c_1 c_2 \ldots c_n]) \stackrel{\text{def}}{=} ?$$
$$der\, c\, (r^+) \stackrel{\text{def}}{=} ?$$
$$der\, c\, (r^?) \stackrel{\text{def}}{=} ?$$
$$der\, c\, (r^{\{n,m\}}) \stackrel{\text{def}}{=} ?$$
$$der\, c\, (\sim r) \stackrel{\text{def}}{=} ?$$

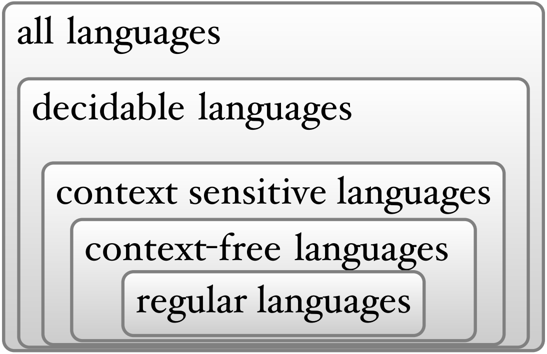
Regular Languages

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

$$A \rightarrow rhs$$

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

$$A \rightarrow rhs$$

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

We also allow rules

$$A \rightarrow rhs_1 | rhs_2 | \ldots$$

Palindromes

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

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

Palindromes

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

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

or

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

Palindromes

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

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

or

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

Can you find the grammar rules for matched parentheses?

Arithmetic Expressions

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

Arithmetic Expressions

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

`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 $X \rightarrow rhs$
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

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

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

Context Sensitive Grammars

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

$$
\begin{array}{lcl}
S & \rightarrow & bSAA \mid \epsilon \\
A & \rightarrow & a \\
bA & \rightarrow & Ab
\end{array}
$$

Context Sensitive Grammars

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

$$
\begin{array}{lcl}
S & \rightarrow & bSAA \mid \epsilon \\
A & \rightarrow & a \\
bA & \rightarrow & Ab
\end{array}
$$

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

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:

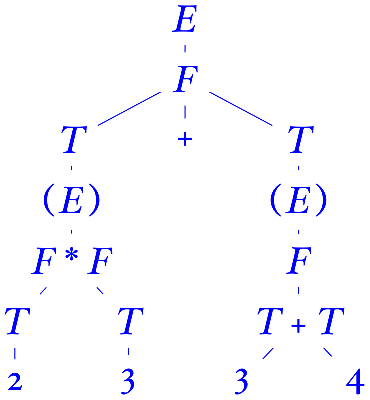
$$\{c_1 \ldots c_n \mid \forall i.\, c_i \in T \wedge S \rightarrow^* c_1 \ldots c_n\}$$

- 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

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

`(2*3)+(3+4)`



Arithmetic Expressions

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

Arithmetic Expressions

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

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.

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

`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

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 parts
- then combine the results:

$$((\text{output}_1, \text{output}_2), \text{unparsed part})$$

$$\begin{aligned}\{((o_1, o_2), u_2) \mid \\ (o_1, u_1) \in p(\text{input}) \wedge \\ (o_2, u_2) \in q(u_1)\}\end{aligned}$$

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 $p \mid\mid q$ 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 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

```
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)
}
```

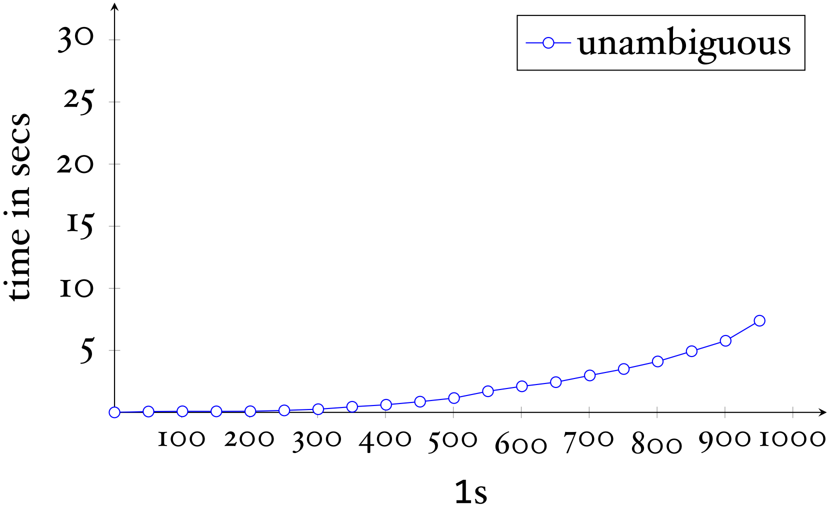
Two Grammars

Which languages are recognised by the following two grammars?

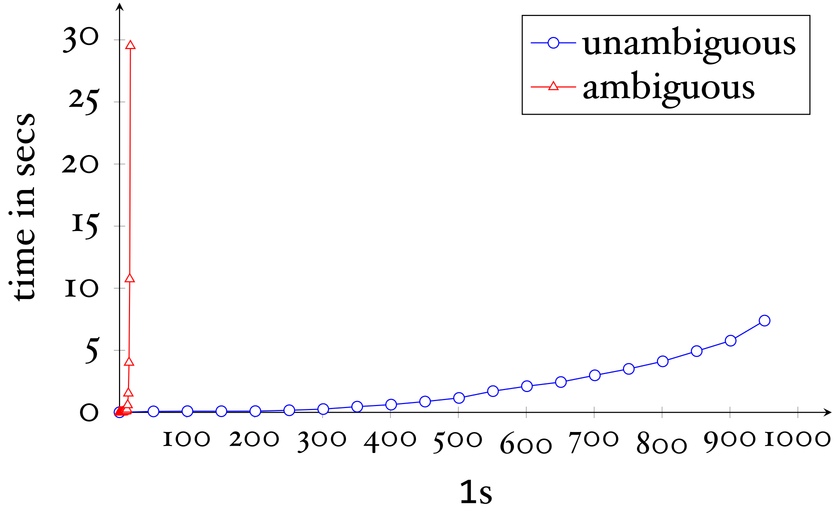
$$\begin{array}{rcl} S & \rightarrow & 1 \cdot S \cdot S \\ & | & \epsilon \end{array}$$

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

Ambiguous Grammars



Ambiguous Grammars



While-Language

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

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)`