1. Review of Data Structures in rePL
   - Records in rePL
   - Syntactic Sugar

2. Exceptions in rePL

3. Denotational Semantics of rePL

4. Pass-by-name and Pass-by-need

5. A Virtual Machine for rePL
Records in rePL

Records in rePL are bracket-enclosed sequences of property-value associations.

In rePL we can represent a pair containing 10 and 20 as a record of the form

[First:10, Second:20]
Accessing Records Using “Dot”

let p = [First:10, Second:20]
in p.First + p.Second
end
Syntax of Records in rePL

\[
\begin{align*}
E_1 & \quad E_n \\
\hline
\text{[}q_1 : E_1, \ldots, q_n : E_n\text{]} & \quad E \\
\end{align*}
\]

where \(q, q_1, \ldots, q_n\) denote properties
More Record Operators

The operator

\[ E \text{ hasproperty } q \]

returns \textit{true}, if the record resulting from \( E E \) has property \( q \), and \textit{false} otherwise.

Examples:

\([\text{Red}:0, \text{Blue}:127, \text{Green}:255]\) hasproperty Green

\([\text{Red}:0, \text{Blue}:127, \text{Green}:255]\) hasproperty Yellow
The Empty Record

The empty record [] does not have any properties.

The operator empty checks whether its argument is the empty record.

Examples:

empty []

empty [SomeProperty: 1]
Syntax

\[
E \\
p[E]
\]

where \( p \in \{\\backslash, \text{empty}\} \)
Syntactic Sugar for rePL

We write $E_1::E_2$ as abbreviation for $[\text{First}:E_1, \text{Second}:E_2]$.

The operator $::$ is right-associative.

We can write $10 :: 20 :: 30$ instead of $10 :: (20 :: 30)$. 
Lists

A list is either empty—in which case it is represented by the empty record \([\,]\)—or a pair, whose second component is also a list.

The elements of the list are the first components of the pairs that make up the list.

Example:

\[10 :: 20 :: 30 :: 40 :: []\]
Constructing Lists

```rePL
let even = recfun even i counter done ->
    if counter=done then []
    else i :: (even i+2 counter+1 done)
end
end
in let evennumbers = fun n -> (even 2 0 n) end
    in ...
end
end
```

The expression (evennumbers 3) returns the list `2 :: 4 :: 6 :: []`. 
Length of Lists

The following function computes the length of a given list.

```plaintext
recfun length xs ->
    if empty xs then 0
    else 1+(length xs.Second)
end
end
```
Mapping Lists

recfun map xs f -> if empty xs then []
    else (f xs.First) :: (map xs.Second f)
end

Example:

(map 1 :: 2 :: 3 :: [] fun x -> x * x end)

returns 1 :: 4 :: 9 :: [].

CS 4215: Programming Language Implementation 08—The Language rePL
1. Review of Data Structures in rePL

2. Exceptions in rePL
   - Motivation
   - Syntax of Exception Handling
   - Built-in Exceptions
   - Programmer-defined Exceptions

3. Denotational Semantics of rePL

4. Pass-by-name and Pass-by-need

5. A Virtual Machine for rePL
Motivation

Errors arise from

- Division by zero
- Invalid record access
- ...

Motivation

Syntax of Exception Handling
Built-in Exceptions
Programmer-defined Exceptions
Handling Exceptions

try (evaluate input)
catch e
with if e hasproperty DivisionByZero
    then (evaluate (readNewUserInput))
else ..
end
end
Syntax of Exception Handling

\[
\text{try } E_1 \text{ catch } x \text{ with } E_2 \text{ end}
\]
Built-in Exceptions

Division by zero leads to an exception of the form `DivisonByZero:true`. Invalid record access leads to an exception of the form `InvalidRecordAccess:true`.

Examples:

4711 / 0

[].SomeProperty
Let the Programmer Throw Exceptions

\[
E \\
\cdashrule[0.4pt]{2.5cm}
\text{throw } E \text{ end}
\]
Example

```plaintext
if percentage > 100
then throw [PercentageExceeds100: true, PercentageValue: percentage]
end
else ... end
```

The `percentageExceeds100` exception can then be caught by a surrounding expression and handled appropriately.
Review of Data Structures in rePL

Exceptions in rePL

Denotational Semantics of rePL
  - rePL0: simPL plus Records
  - rePL1: rePL0 plus Exceptions

Pass-by-name and Pass-by-need

A Virtual Machine for rePL
## Semantic Domains for rePL0

<table>
<thead>
<tr>
<th>Sem. domain</th>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bool</strong></td>
<td>{true, false}</td>
<td>ring of booleans</td>
</tr>
<tr>
<td><strong>Int</strong></td>
<td>{..., −2, −1, 0, 1, 2, ...}</td>
<td>ring of integers</td>
</tr>
<tr>
<td><strong>EV</strong></td>
<td>(\text{Bool} + \text{Int} + {\perp} + \text{Fun} + \text{Rec})</td>
<td>expressible values</td>
</tr>
<tr>
<td><strong>DV</strong></td>
<td>(\text{Bool} + \text{Int} + \text{Fun} + \text{Rec})</td>
<td>denotable values</td>
</tr>
<tr>
<td><strong>Id</strong></td>
<td>alphanumeric string</td>
<td>identifiers</td>
</tr>
<tr>
<td><strong>Env</strong></td>
<td>\text{Id} \mapsto \text{DV}</td>
<td>environments</td>
</tr>
<tr>
<td><strong>Fun</strong></td>
<td>(\text{DV} \star \cdots \star \text{DV} \mapsto \text{EV})</td>
<td>function values</td>
</tr>
<tr>
<td><strong>Rec</strong></td>
<td>\text{Id} \mapsto \text{DV}</td>
<td>records</td>
</tr>
</tbody>
</table>
Rules for rePL0

\[ \Delta \models E_1 \rightarrow v_1 \quad \cdots \quad \Delta \models E_n \rightarrow v_n \]
\[ \Delta \models [q_1 : E_1, \ldots, q_n : E_n] \rightarrow f \]
where \( f = \emptyset[q_1 \leftarrow v_1] \)
\[ \cdots [q_n \leftarrow v_n] \]
Rules for rePL0

\[ \Delta \vdash E \rightarrow v \]

\[ \Delta \vdash E \cdot q \rightarrow v' \]

where \( v' = v(q) \)
Rules for rePL0

\[
\begin{align*}
\Delta \models E \Rightarrow v & \quad \text{if } \text{dom}(v) = \emptyset \\
\Delta \models \text{empty } E \Rightarrow \text{true} & \\
\Delta \models E \Rightarrow v & \quad \text{if } \text{dom}(v) \neq \emptyset \\
\Delta \models \text{empty } E \Rightarrow \text{false}
\end{align*}
\]
Rules for rePL0

\[ \Delta \models E \rightarrow v \]

\[ \Delta \models E \text{ hasproperty } q \rightarrow true \]

\[ \Delta \models E \rightarrow v \]

\[ \Delta \models E \text{ hasproperty } q \rightarrow false \]
### Semantic Domains for rePL1

<table>
<thead>
<tr>
<th>Sem. domain</th>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bool</strong></td>
<td>{true, false}</td>
<td>ring of booleans</td>
</tr>
<tr>
<td><strong>Int</strong></td>
<td>{\ldots, -2, -1, 0, 1, 2, \ldots}</td>
<td>ring of integers</td>
</tr>
<tr>
<td><strong>EV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Exc</strong> + <strong>Fun</strong> + <strong>Rec</strong></td>
<td>expressible values</td>
</tr>
<tr>
<td><strong>DV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Fun</strong> + <strong>Rec</strong></td>
<td>denotable values</td>
</tr>
<tr>
<td><strong>Id</strong></td>
<td>alphanumeric string</td>
<td>identifiers</td>
</tr>
<tr>
<td><strong>Env</strong></td>
<td><strong>Id</strong> (\leadsto) <strong>DV</strong></td>
<td>environments</td>
</tr>
<tr>
<td><strong>Fun</strong></td>
<td><strong>DV</strong> (\ast) (\ast) <strong>DV</strong> (\leadsto) <strong>EV</strong></td>
<td>function values</td>
</tr>
<tr>
<td><strong>Rec</strong></td>
<td><strong>Id</strong> (\leadsto) <strong>DV</strong></td>
<td>records</td>
</tr>
<tr>
<td><strong>Exc</strong></td>
<td><strong>Rec</strong></td>
<td>exceptions</td>
</tr>
</tbody>
</table>
Rules for rePL1

\[
\Delta \models E_1 \rightarrow e \\
\hline
\Delta \models E_1 + E_2 \rightarrow e
\]

if \( e \in \textbf{Exc} \)

\[
\Delta \models E_1 \rightarrow v \\
\Delta \models E_2 \rightarrow e
\]

\[
\Delta \models E_1 + E_2 \rightarrow e
\]

if \( v \not\in \textbf{Exc} \) and \( e \in \textbf{Exc} \)

\[
\Delta \models E_1 \rightarrow v_1 \\
\Delta \models E_2 \rightarrow v_2
\]

\[
\Delta \models E_1 + E_2 \rightarrow v_1 + v_2
\]

if \( v_1, v_2 \not\in \textbf{Exc} \)
Rules for rePL1

\[
\begin{align*}
\Delta \vdash E_1 \rightarrow e & \quad \text{if } e \in \text{Exc} \\
\Delta \vdash E_1 / E_2 \rightarrow e & \\
\Delta \vdash E_1 \rightarrow v & \quad \Delta \vdash E_2 \rightarrow e & \quad \text{if } v \notin \text{Exc} \text{ and } e \in \text{Exc} \\
\Delta \vdash E_1 / E_2 \rightarrow e & \\
\Delta \vdash E_1 \rightarrow v_1 & \quad \Delta \vdash E_2 \rightarrow v_2 & \quad \text{if } v_1, v_2 \notin \text{Exc} \text{ and } v_2 \neq 0 \\
\Delta \vdash E_1 / E_2 \rightarrow v_1 / v_2 &
\end{align*}
\]
Rules for rePL1

\[ \Delta \models E_1 \rightarrow v_1 \quad \Delta \models E_2 \rightarrow 0 \]

\[ \Delta \models E_1 / E_2 \rightarrow e \]

if \( v_1 \notin \text{Exc} \) and

where \( e = \{\text{DivisionByZero: true}\} \), and \( e \in \text{Exc} \)
Rules for rePL1

\[
\begin{align*}
\Delta \vdash E \rightarrow e & \quad \text{if } e \in \text{Exc} \\
\Delta \vdash E \cdot q \rightarrow e \\
\Delta \vdash E \rightarrow v & \quad \text{if } q \in \text{dom}(v) \text{ and where } v' = v(q) \\
\Delta \vdash E \cdot q \rightarrow v' \\
\Delta \vdash E \rightarrow v & \quad \text{if } q \notin \text{dom}(v) \text{ and where } \\
\Delta \vdash E \cdot q \rightarrow e & \quad e = \text{[InvalidRecordAccess: true]}, \text{ and } e \in \text{Exc}
\end{align*}
\]
Rules for rePL1

\[
\Delta \vdash E \rightarrow v
\]

---

\[
\Delta \vdash \text{throw } E \text{ end } \rightarrow e
\]

if \( v \in \text{Rec} \cup \text{Exc} \) and

where \( e = v, \ e \in \text{Exc} \)
1. Review of Data Structures in rePL
2. Exceptions in rePL
3. Denotational Semantics of rePL
4. Pass-by-name and Pass-by-need
   - Pass-by-name
   - Pass-by-need
   - Examples
5. A Virtual Machine for rePL
Pass-by-name

- Pass-by-name is an evaluation strategy where arguments of functions are passed immediately, instead of evaluating them.
- Pass-by-need is an optimization of pass-by-name.
Pass-by-name

- Pass argument expressions without evaluating them
- Evaluate them when needed
- What environment to use?
Thunks

Thunks are expressions together with an environment.
## Semantic Domains

<table>
<thead>
<tr>
<th>Sem. domain</th>
<th>Definition</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bool</strong></td>
<td>{true, false}</td>
<td>ring of booleans</td>
</tr>
<tr>
<td><strong>Int</strong></td>
<td>{\ldots, -2, -1, 0, 1, 2, \ldots}</td>
<td>ring of integers</td>
</tr>
<tr>
<td><strong>EV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Exc</strong> + <strong>Fun</strong> + <strong>Rec</strong></td>
<td>expressible values</td>
</tr>
<tr>
<td><strong>DV</strong></td>
<td><strong>Bool</strong> + <strong>Int</strong> + <strong>Fun</strong> + <strong>Rec</strong> + <strong>Thunk</strong></td>
<td>denotable values</td>
</tr>
<tr>
<td><strong>Id</strong></td>
<td>alphanumeric string</td>
<td>identifiers</td>
</tr>
<tr>
<td><strong>Env</strong></td>
<td><strong>Id</strong> ⇝ <strong>DV</strong></td>
<td>environments</td>
</tr>
<tr>
<td><strong>Fun</strong></td>
<td><strong>DV</strong> ∗ ⋯ ∗ <strong>DV</strong> ⇝ <strong>EV</strong></td>
<td>function values</td>
</tr>
<tr>
<td><strong>Rec</strong></td>
<td><strong>Id</strong> ⇝ <strong>DV</strong></td>
<td>records</td>
</tr>
<tr>
<td><strong>Exc</strong></td>
<td><strong>Rec</strong></td>
<td>exceptions</td>
</tr>
<tr>
<td><strong>Thunk</strong></td>
<td>rePL ∗ <strong>Env</strong></td>
<td>thunks</td>
</tr>
</tbody>
</table>
Evaluation of Application

\[ \Delta \vdash E \rightarrow f \]

\[ \Delta \vdash (E \ E_1 \ldots \ E_n) \rightarrow f((E_1, \Delta), \ldots, (E_n, \Delta)) \]
Evaluation of Identifiers

\[ \Delta' \vdash E \rightarrow v \]
\[ \Delta \vdash x \rightarrow v \]

if \( \Delta(x) \) is thunk of the form \((E, \Delta')\).

\[ \Delta \vdash x \rightarrow \Delta(x) \]

if \( \Delta(x) \) is not a thunk.
Pass-by-need

- Observation: In pass-by-name, the same thunk could be evaluated multiple times.
- Pass-by-need is an optimization of pass-by-name that avoids multiple evaluation of the same thunk.
- Pass-by-need enjoys popularity in functional programming (Haskell, Miranda).
Infinite Lists

```plaintext
let cons = fun x y -> x :: y end
in let makeints = recfun makeints i ->
    (cons i (makeints i + 1))
end
in let allints = (makeints 0)
in  allints.Second.Second.First
end
end
```
Squaring Infinite Lists

let allints = (makeints 0)
in
  let allsquares = (map allints fun x -> x * x end) in allsquares.Second.Second.Second.First
end
end
Review of Data Structures in rePL
Exceptions in rePL
Denotational Semantics of rePL
Pass-by-name and Pass-by-need
A Virtual Machine for rePL

1. Review of Data Structures in rePL
2. Exceptions in rePL
3. Denotational Semantics of rePL
4. Pass-by-name and Pass-by-need
5. A Virtual Machine for rePL
   - Records
   - Exceptions
   - Implementing Records Efficiently
New Instructions for Record Construction

New instructions:

\[ s \]

\[ \text{LDPS } q.s \]  \hspace{2cm}  \[ s \]

\[ \text{RCDS } i.s \]

where \( q \) is a property and \( i \) is an integer.
Compilation of Record Construction

\[
E_1 \leftrightarrow s_1 \quad \ldots \quad E_n \leftrightarrow s_n
\]

\[
[q_1:E_1, \ldots, q_n:E_n] \leftrightarrow \text{LDPS } q_1.s_1 \ldots \text{LDPS } q_n.s_n.\text{RCDS } n
\]
Execution of LDPS

\[ s(pc) = \text{LDPS } q \]

\[ (os, pc, e, rs) \Rightarrow_s (q.os, pc + 1, e, rs) \]
Execution of RCDS

Consider LDPS $q_1.s_1$\ldots LDPS $q_n.s_n.RCDS \ n$ resulting from $[q_1 : E_1, \ldots, q_n : E_n]$.

After LDPS $q_1.s_1$\ldots LDPS $q_n.s_n$, instruction RCDS $n$ finds association list on operand stack.

$$s(pc) = RCDS \ n$$

$$((v_n.q_n, \ldots, v_1.q_1.os, pc, e, rs) \Rightarrow_s ((q_1, v_1), \ldots, (q_n, v_n)).os, pc + 1, e, rs)$$
Operations on Records

Operations: empty, ".", hasproperty

New instructions:

\[
\begin{align*}
\text{EMPTY} &. s \\
\text{DOT} &. s \\
\text{HASP} &. s
\end{align*}
\]
Compilation of Record Operations

\[
E \rightarrow s \\
\text{empty } E \rightarrow s.\text{EMPTY} \\
E \rightarrow s \\
E \text{ hasproperty } q \rightarrow s.\text{LDPS } q.\text{HASP}
\]

\[
E \rightarrow s \\
E . q \rightarrow s.\text{LDPS } q.\text{DOT}
\]
Executions of Record Operations

\[
s(pc) = \text{EMPTY} \\
\begin{array}{l}
\text{if } v = \emptyset \\
(v.os, pc, e, rs) \leadsto_s (true.os, pc + 1, e, rs)
\end{array}
\]

\[
s(pc) = \text{EMPTY} \\
\begin{array}{l}
\text{if } v \neq \emptyset \\
(v.os, pc, e, rs) \leadsto_s (false.os, pc + 1, e, rs)
\end{array}
\]
Executions of Record Operations

\[
\begin{align*}
\text{s}(pc) &= \text{DOT} \\
\begin{array}{l}
\text{if } v(q) = v' \\
(q.v.os, pc, e, rs) \Rightarrow_s (v'.os, pc + 1, e, rs)
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{s}(pc) &= \text{HASP} \\
\begin{array}{l}
\text{if } \exists v_i . (q, v_i) \in v \\
(q.v.os, pc, e, rs) \Rightarrow_s (true.os, pc + 1, e, rs)
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{s}(pc) &= \text{HASP} \\
\begin{array}{l}
\text{if } \forall v_i . (q, v_i) \in v \\
(q.v.os, pc, e, rs) \Rightarrow_s (false.os, pc + 1, e, rs)
\end{array}
\end{align*}
\]
Built-in Exceptions

Division by zero and record access throw exceptions. Idea: place instructions for raising these two exceptions at the end of the instruction sequence.

\[
E \rightarrow s_1
\]

[divisionByZero:true] \rightarrow s_2

[invalidRecordAccess:true] \rightarrow s_3

\[E_{s_1}.\text{DONE}.s_2.\text{THROW}.s_3.\text{THROW}\]

beginning address of \( s_2 \): \( addr_{\text{divisionByZero}} \)

beginning address of \( s_3 \): \( addr_{\text{invalidRecordAccess}} \)
Primitive Operations Throwing Exceptions

\[
s(pc) = \text{DIV}\\
\]

\[
(0.i_1.os, pc, e, rs) \Rightarrow_s (os, addr_{\text{divisionByZero}}, e, rs)\\
\]

\[
s(pc) = \text{DOT}\\
\]

\[
(q.v.os, pc, e, rs) \Rightarrow_s (os, addr_{\text{invalidRecordAccess}}, e, rs)\\
\]

if \( \forall v.(q, v') \in v \)
Programmer-defined Exception Throws

\[ E \rightarrow s \]

\[ \text{throw } E \text{ end } \rightarrow s.\text{THROW} \]
Use runtime stack to keep track of the `catch...with...` part of `try` expressions

Exception from `try` part will pop stackframes, until it finds the appropriate `catch...with...` part
Translation of try Statement

\[
E_1 \rightarrow s_1 \quad E_2 \rightarrow s_2
\]

\[
\text{try } E_1 \text{ catch } x \text{ with } E_2 \text{ end} \quad \rightarrow \\
(\text{TRY } x \mid s_1 \mid + 3).s_1.\text{ENDTRY}.(\text{GOTOR } s_2 \mid + 1).s_2
\]
Execution of TRY Instruction

\[ s(pc) = \text{TRY} \times i \]

\[
(os, pc, e, rs) \Rightarrow_s (os, pc + 1, e, (\text{catch}, x, pc + i, os, e).rs)
\]
 Execution of ENDTRY Instruction

\[ s(pc) = \text{ENDTRY} \]

\[ (os, pc, e, (\text{catch}, x, pc', os, e).rs) \xRightarrow{s} (os, pc + 1, e, rs) \]
Throwing of an Exception

\[ s(pc) = \text{THROW} \]

\[
(o.s, p.c, e, (p.c', o.s', e').r.s) \Rightarrow_s (o.s, p.c, e, r.s)
\]

\[ s(pc) = \text{THROW} \]

\[
(v.o.s, p.c, e, (\text{catch}, x, p.c', o.s', e').r.s) \Rightarrow_s (o.s', p.c', e'[x \leftarrow v], r.s)
\]
Problems with Records in RVM

- representation of records as a set of pairs is inefficient
- properties are strings
Observations

- Properties always appear literally
- Records are always constructed with [...], which explicitly lists all properties
Representing Properties

- compiler constructs set $Q$ of all properties in a given $E$
- compiler calculates a bijection $idp$ between $Q$ and $[0 \ldots |Q| - 1]$
- compiler replaces every occurrence of a property $q$ in an instruction by $idp(q)$
Compilation of Record Operations (revisited)

\[ E \rightarrow s \]

\[ E . q \rightarrow s . \text{LDCI} \ idp(q) . \text{DOT} \]

\[ E \rightarrow s \]

\[ E \text{ hasproperty} \ q \rightarrow s . \text{LDCI} \ idp(q) . \text{HASP} \]
Record Construction

- All records are constructed by [\ldots]
- The compiler calculates a bijection $idr$ between the set $R$ of all property sets of records and $[0 \ldots |R| - 1]$.
- Associate with each property $q$ of each record its alphabetical position in the corresponding property set:
  $p(idr(\{q_1, \ldots, q_n\}), idp(q))$, starting with 0. If a record with index $m$ does not have a property with index $n$, we set $p(m, n) = -1$. 
Example

- Let us say compiler assigns the number 13 to the set of properties \( \{a, b\} \) (thus \( idr(\{a, b\} = 13) \))
- the number 55 to the property \( a \)
  \( idp(a) = 55 \)
- the number 77 to the property \( b \)
  \( idp(b) = 77 \)
- the position of property \( a \) in \( [a:5\ b:7] \) is \( p(13, 55) = 0 \)
- For any property identifier \( n \neq 55, 77 \): \( p(13, n) = -1. \)
Representing Records

Represent a record with properties $q_1, \ldots, q_n$ as a pair consisting of identifier $idr\{q_1, \ldots, q_n\}$ and array that maps the alphabetical position of each $q$ in the corresponding property list.
Example

In the example above, since \( p(13,55) = 0 \) and \( p(13,77) = 1 \), we can represent the record \([a:5\ b:7]\) by the pair \((13,[0:5,1:7])\).
New Translation of Record Construction

\[ E_1 \rightarrow s_1 \quad \cdots \quad E_n \rightarrow s_n \]

\[ [q_1:E_1, \ldots, q_n:E_n] \rightarrow \]

\[ \text{LDCI } idp(q_1).s_1 \ldots \text{LDCI } idp(q_n).s_n.\text{RCD } n \text{ idr}\{q_1, \ldots, q_n\} \]
Efficient Records in RVM

• compiler passes the table $p$ to RVM
• RCD constructs an array, whose indices corresponding to the record properties are given by $p$. 
New Execution of Record Construction

\[ s(pc) = \text{RCD } n \ m \]

\[
(v_n.i_n \ldots \ldots v_1.i_1.os, pc, e, rs) \\
\xrightarrow{s} \\
((m, \{(p(m, idp(q_1)), v_1), \ldots, (p(m, idp(q_n)), v_n)\}).os, pc + 1, e, rs)
\]
New Execution of Record Operations

\[ s(pc) = \text{DOT} \]

if

\[ (i.(m,a).os, pc, e, rs) \Rightarrow_s (a(j).os, pc + 1, e, rs) \]

\[ p(m, i) = j, j \geq 0 \]

\[ s(pc) = \text{DOT} \]

if

\[ (i.(m,a).os, pc, e, rs) \Rightarrow_s (os, addr_{\text{invalidRecordAccess}}, e, rs) \]

\[ p(m, i) = -1 \]
New Execution of Record Operations

\[
s(pc) = \text{HASP}
\]

\[
\implies_{s} (i.(m, a).os, pc, e, rs) \implies_s (true.os, pc + 1, e, rs)
\]

\[
s(pc) = \text{HASP}
\]

\[
\implies_{s} (i.(m, a).os, pc, e, rs) \implies_s (false.os, pc + 1, e, rs)
\]

\[
p(m, i) = -1
\]
Summary of Record Implementation

Constant time record access achieved by:

- representing properties by integers using idp
- mapping record property sets to integers using idr
- record access through arrays using lookup table p
Overview of Next Lecture

- Imperative Programming: The language imPL