Exceptions, Functions

Guillaume Melquiond

Cours MPRI 2-36-1 “Preuve de Programme”

9 janvier 2012
Reminder of the Last 2 Lectures

- Simple IMP programs:

- Hoare logic:
  - deduction rules for triples $\{\text{Pre}\} e \{\text{Post}\}$
  - notions of validity and safety (progress)

- Weakest precondition computation:
  - $\{\text{Pre}\} e \{\text{Post}\}$ valid if $\text{Pre} \Rightarrow \text{WP}(e, \text{Post})$
  - notion of preservation by reduction.

- Extension: labels.
Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),

- Hoare logic:
  - deduction rules for triples \{Pre\} e \{Post\},
  - notions of validity and safety (progress).

- Weakest precondition computation:
  - \{Pre\} e \{Post\} valid if \( Pre \Rightarrow WP(e, Post) \),
  - notion of preservation by reduction.

- Extension: labels.
Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.
Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

- Hoare logic:
Reminder of the Last 2 Lectures

- **Simple IMP programs:**
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

- **Hoare logic:**
  - deduction rules for triples $\{Pre\} e \{Post\}$,
Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

- Hoare logic:
  - deduction rules for triples $\{\text{Pre}\} e \{\text{Post}\}$,
  - notions of validity and safety (progress).
Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

- Hoare logic:
  - deduction rules for triples $\{Pre\} e \{Post\}$,
  - notions of validity and safety (progress).

- Weakest precondition computation:
Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

- Hoare logic:
  - deduction rules for triples $\{Pre\} e \{Post\}$,
  - notions of validity and safety (progress).

- Weakest precondition computation:
  - $\{Pre\} e \{Post\}$ valid if $Pre \Rightarrow WP(e, Post)$,
Reminder of the Last 2 Lectures

- **Simple IMP programs:**
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

- **Hoare logic:**
  - deduction rules for triples $\{Pre\}e\{Post\}$,
  - notions of validity and safety (progress).

- **Weakest precondition computation:**
  - $\{Pre\}e\{Post\}$ valid if $Pre \Rightarrow WP(e, Post)$,
  - notion of preservation by reduction.
Reminder of the Last 2 Lectures

- Simple IMP programs:
  - basic datatypes (e.g., bool, int),
  - global variables and let-in bindings,
  - program = single expression with side effects.

- Hoare logic:
  - deduction rules for triples $\{Pre\}e\{Post\}$,
  - notions of validity and safety (progress).

- Weakest precondition computation:
  - $\{Pre\}e\{Post\}$ valid if $Pre \Rightarrow WP(e, Post)$,
  - notion of preservation by reduction.

- Extension: labels.
Next Extensions

- Mutable local variables.
- Exceptions.
- Functions (call by value).
Outline

Local Variables

Exceptions

Functions
Mutable Local Variables

We extend the syntax of expressions with

\[ e ::= \text{let ref } id = e \text{ in } e \]

Example: isqrt revisited

```plaintext
val x, res : ref int

isqrt:
    res := 0;
    let ref sum = 1 in
    while sum ≤ x do
        res := res + 1; sum := sum + 2 * res + 1
    done
```
Operational Semantics

\[ \Sigma, \Pi, e \sim \Sigma', \Pi', e' \]

\( \Pi \) no longer contains just immutable variables.

\[ \Sigma, \Pi, e_1 \sim \Sigma', \Pi', e'_1 \]

\( \Sigma, \Pi, \text{let ref } x = e_1 \text{ in } e_2 \sim \text{let ref } x = e'_1 \text{ in } e_2 \)

\[ \Sigma, \Pi, \text{let ref } x = v \text{ in } e \sim \Sigma, \Pi\{(x, \text{Here}) \mapsto v\}, e \]
Operational Semantics

\[ \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \]

\( \Pi \) no longer contains just immutable variables.

\[ \Sigma, \Pi, e_1 \rightsquigarrow \Sigma', \Pi', e'_1 \]

\[ \Sigma, \Pi, \text{let ref } x = e_1 \text{ in } e_2 \rightsquigarrow \text{let ref } x = e'_1 \text{ in } e_2 \]

\[ \Sigma, \Pi, \text{let ref } x = v \text{ in } e \rightsquigarrow \Sigma, \Pi\{(x, \text{Here}) \mapsto v\}, e \]

\( x \) local variable

\[ \Sigma, \Pi, x := v \rightsquigarrow \Sigma, \Pi\{(x, \text{Here}) \mapsto v\}, e \]

And labels too.
Mutable Local Variables: WP rules

Exercise: propose rules for \( \text{WP(} \text{let ref } x = e_1 \text{ in } e_2, Q) \), \( \text{WP(} x := e, Q) \), and \( \text{WP(} L : e, Q) \).
Mutable Local Variables: WP rules

\[
\text{WP}(\text{let ref } x = e_1 \text{ in } e_2, Q) = \text{WP}(e_1, \text{WP}(e_2, Q)[x \leftarrow \text{result}])
\]

\[
\text{WP}(x := e, Q) = \text{WP}(e, Q[x \leftarrow \text{result}])
\]

\[
\text{WP}(L : e, Q) = \text{WP}(e, Q)[x@L \leftarrow x, \text{for all } x@L]
\]
Outline

Local Variables

Exceptions

Functions
Exceptions

We extend the syntax of expressions with

\[ e ::= \text{raise } exn \]
\[ \mid \text{try } e \text{ with } exn \Rightarrow e \]

with \( exn \) a set of exception identifiers.
Operational Semantics

Propagation of thrown exceptions:

\[ \Sigma, \Pi, (\text{let } x = \text{raise } \text{exn in } e) \leadsto \Sigma, \Pi, \text{raise } \text{exn} \]
Operational Semantics

Propagation of thrown exceptions:

$$\Sigma, \Pi, (\text{let } x = \text{raise } exn \text{ in } e) \leadsto \Sigma, \Pi, \text{raise } exn$$

Reduction of try-with:

$$\Sigma, \Pi, e \leadsto \Sigma', \Pi', e'$$

$$\Sigma, \Pi, (\text{try } e \text{ with } exn \Rightarrow e'') \leadsto \Sigma', \Pi', (\text{try } e' \text{ with } exn \Rightarrow e'')$$
Operational Semantics

Propagation of thrown exceptions:

$$\Sigma, \Pi, (\text{let } x = \text{raise exn in } e) \rightsquigarrow \Sigma, \Pi, \text{raise exn}$$

Reduction of try-with:

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

$$\Sigma, \Pi, (\text{try } e \text{ with } \text{exn } \Rightarrow e'') \rightsquigarrow \Sigma', \Pi', (\text{try } e' \text{ with } \text{exn } \Rightarrow e'')$$

Normal execution:

$$\Sigma, \Pi, (\text{try } v \text{ with } \text{exn } \Rightarrow e') \rightsquigarrow \Sigma, \Pi, v$$
Operational Semantics

Propagation of thrown exceptions:

$$\Sigma, \Pi, (\text{let } x = \text{raise } exn \text{ in } e) \rightsquigarrow \Sigma, \Pi, \text{raise } exn$$

Reduction of try-with:

$$\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$$

$$\Sigma, \Pi, (\text{try } e \text{ with } exn \Rightarrow e'') \rightsquigarrow \Sigma', \Pi', (\text{try } e' \text{ with } exn \Rightarrow e'''')$$

Normal execution:

$$\Sigma, \Pi, (\text{try } v \text{ with } exn \Rightarrow e') \rightsquigarrow \Sigma, \Pi, v$$

Exception handling:

$$\Sigma, \Pi, (\text{try raise } exn \text{ with } exn \Rightarrow e) \rightsquigarrow \Sigma, \Pi, e$$

$$exn \neq exn'$$

$$\Sigma, \Pi, (\text{try raise } exn \text{ with } exn' \Rightarrow e) \rightsquigarrow \Sigma, \Pi, \text{raise } exn$$
Hoare Triples

Hoare triple modified to allow **exceptional post-conditions**:

\[
\{ P \} e \{ Q \ | \ exn_i \Rightarrow R_i \} 
\]
Hoare Triples

Hoare triple modified to allow exceptional post-conditions:

$$\{P\} e \{Q \mid \text{exn}_i \Rightarrow R_i\}$$

Validity: if $e$ is executed in a state where $P$ holds, it does not block and

- if it terminates normally with value $v$ in state $\Sigma$, then $Q[\text{result} \leftarrow v]$ holds in $\Sigma$;
Hoare Triples

Hoare triple modified to allow exceptional post-conditions:

\[ \{ P \} e \{ Q \mid exn_i \Rightarrow R_i \} \]

Validity: if \( e \) is executed in a state where \( P \) holds, it does not block and

- if it terminates normally with value \( v \) in state \( \Sigma \), then \( Q[\text{result} \leftarrow v] \) holds in \( \Sigma \);
- if it terminates with exception \( exn \) in state \( \Sigma \), then there exists \( i \) such that \( exn = exn_i \) and \( R_i \) holds in \( \Sigma \).
Hoare Triples

Hoare triple modified to allow exceptional post-conditions:

\[
\{ P \} e \{ Q \mid \text{exn}_i \Rightarrow R_i \}
\]

Validity: if \( e \) is executed in a state where \( P \) holds, it does not block and

- if it terminates normally with value \( v \) in state \( \Sigma \), then \( Q[\text{result} \leftarrow v] \) holds in \( \Sigma \);
- if it terminates with exception \( \text{exn} \) in state \( \Sigma \), then there exists \( i \) such that \( \text{exn} = \text{exn}_i \) and \( R_i \) holds in \( \Sigma \).

Note: if \( e \) terminates with an exception not in the set \( \{ \text{exn}_i \} \), the triple is not valid.
Function $WP$ modified to allow exceptional post-conditions too:

$$WP(e, Q, exn_i \Rightarrow R_i)$$

Implicitly, $R_k = False$ for any $exn_k \not\in \{exn_i\}$.
WP Rules

Function WP modified to allow exceptional post-conditions too:

$$\text{WP}(e, Q, \text{exn}_i \Rightarrow R_i)$$

Implicitly, $$R_k = False$$ for any $$\text{exn}_k \not\in \{\text{exn}_i\}$$.

Extension of WP for simple expressions:

$$\text{WP}(x := t, Q, \text{exn}_i \Rightarrow R_i) = Q[\text{result} \leftarrow (), x \leftarrow t]$$

$$\text{WP}(\text{assert } R, Q, \text{exn}_i \Rightarrow R_i) = R \land Q$$
WP Rules

Extension of WP for composite expressions:

\[
WP(\text{let } x = e_1 \text{ in } e_2, Q, \text{exn}_i \Rightarrow R_i) = \\
WP(e_1, WP(e_2, Q, \text{exn}_i \Rightarrow R_i)[\text{result } \leftarrow x], \text{exn}_i \Rightarrow R_i)
\]

\[
WP(\text{if } t \text{ then } e_1 \text{ else } e_2, Q, \text{exn}_i \Rightarrow R_i) = \\
\text{if } t \text{ then } WP(e_1, Q, \text{exn}_i \Rightarrow R_i) \\
\text{else } WP(e_2, Q, \text{exn}_i \Rightarrow R_i)
\]

\[
WP\left(\text{while } c \text{ invariant } l \text{ variant } v, \prec \text{ do } e, Q, \text{exn}_i \Rightarrow R_i\right) = l \land \forall x_1, \ldots, x_k, \\
(l \land \text{if } c \text{ then } WP(L: e, l \land v \prec v@L, \text{exn}_i \Rightarrow R_i) \\
\text{else } Q)[w_i \leftarrow x_i]
\]

where \(w_1, \ldots, w_k\) is the set of assigned variables in expressions and \(x_1, \ldots, x_k\) are fresh logic variables.
Exercise: propose rules for \( \text{WP}(\text{raise } \text{exn}, Q, \text{exn}_i \Rightarrow R_i) \) and \( \text{WP}(\text{try } e_1 \text{ with } \text{exn} \Rightarrow e_2, Q, \text{exn}_i \Rightarrow R_i) \).
WP Rules

\[ WP(\text{raise } \text{exn}_k, Q, \text{exn}_i \Rightarrow R_i) = R_k \]

\[ WP(\text{try } e_1 \text{ with } \text{exn} \Rightarrow e_2), Q, \text{exn}_i \Rightarrow R_i) = \]

\[ WP\left( e_1, Q, \left\{ \begin{array}{l}
\text{exn} \Rightarrow WP(e_2, Q, \text{exn}_i \Rightarrow R_i) \\
\text{exn}_i \backslash \text{exn} \Rightarrow R_i
\end{array} \right\} \right) \]
Outline

Local Variables

Exceptions

Functions
Functions

Program structure:

\[
\begin{align*}
\text{prog} & ::= \text{decl}* \\
\text{decl} & ::= \text{vardecl} \mid \text{fundecl} \\
\text{vardecl} & ::= \text{val id : ref basetype}
\end{align*}
\]
Functions

Program structure:

\[
\begin{align*}
prog & ::= \ decl^* \\
\ decl & ::= \ vardecl \mid \ fundecl \\
\ vardecl & ::= \ val \ id : \ ref \ basetype \\
\ fundecl & ::= \ function \ id( (param,)^* ) : \ basetype \\
& \hspace{1cm} \ contract \ body \ e \\
\ param & ::= \ id : \ basetype \\
\ contract & ::= \ requires \ t \ writes \ (id,)^* \ ensures \ t
\end{align*}
\]
Functions

Program structure:

\[
\begin{align*}
prog & ::= \ decl^* \\
\decl & ::= \ vardecl \mid \ fundecl \\
\vardecl & ::= \ val \ id : \ ref \ \text{basetype} \\
\fundecl & ::= \ \text{function} \ id( (\text{param,})^* ) : \text{basetype} \\
& \quad \text{contract body e} \\
\param & ::= \ id : \ \text{basetype} \\
\contract & ::= \ \text{requires} \ t \ \text{writes} \ (id,)^* \ \text{ensures} \ t 
\end{align*}
\]

Function definition:

- Contract:
  - pre-condition,
  - post-condition (label \textit{Old} available),
  - assigned variables: clause \textit{writes}.
- Body: expression.
Example: isqrt

**function** isqrt(x:int): int
  **requires** x \geq 0
  **ensures** result \geq 0 \land
  sqr(result) \leq x < sqr(result + 1)

**body**
  let ref res = 0 in
  let ref sum = 1 in
  while sum \leq x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
  res
Example using *Old* label

```flexible
val res: ref int

procedure incr(x:int)
    requires true
    writes res
    ensures res = res@Old + x

body
res := res + x
```
Typing

Definition $d$ of function $f$:

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  requires $Pre$
  writes $\vec{w}$
  ensures $Post$
  body $Body$
Typing

Definition $d$ of function $f$:

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  requires $Pre$
  writes $\vec{w}$
  ensures $Post$
body $Body$

Well-formed definitions:

\[
\Gamma' = \{ x_i : \tau_i \mid 1 \leq i \leq n \} \cdot \Gamma
\]
\[
\Gamma' \vdash Pre, Post : \text{formula}
\]
\[
\vec{w}_g \subseteq \vec{w} \text{ for each call } g
\]
\[
y \in \vec{w} \text{ for each assign } y
\]
\[
\Gamma \vdash d : \text{wf}
\]

where $\Gamma$ contains the global declarations.
Typing

Definition $d$ of function $f$:

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  requires $Pre$
  writes $\vec{w}$
  ensures $Post$
  body $Body$

Well-typed function calls:

\[ \Gamma \vdash t_i : \tau_i \quad \Gamma \vdash f(t_1, \ldots, t_n) : \tau \]

Note: $t_i$ are immutable expressions.
Operational Semantics

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
requires $Pre$
writes $\vec{w}$
ensures $Post$
body $Body$

$$\Pi' = \{x_i \mapsto \llbracket t_i \rrbracket_{\Sigma, \Pi}\} \quad \Sigma, \Pi' \models Pre$$

$$\Sigma, \Pi, f(t_1, \ldots, t_n) \rightsquigarrow \Sigma, \Pi, (Old : frame \Pi', Body, Post)$$
frame is a dummy operation that keeps track of the local variables of the callee:

\[
\Sigma, \Pi, e \leadsto \Sigma', \Pi', e'
\]

\[
\Sigma, \Pi'', (\text{frame } \Pi, e, P) \leadsto \Sigma', \Pi'', (\text{frame } \Pi', e', P)
\]

It also checks that the post-condition holds at the end:

\[
\Sigma, \Pi' \models P[\text{result } \leftarrow v]
\]

\[
\Sigma, \Pi, (\text{frame } \Pi', v, P) \leadsto \Sigma, \Pi, v
\]
WP Rule of Function Call

function \( f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \)
  requires \( Pre \)
  writes \( \vec{w} \)
  ensures \( Post \)
body \( Body \)

\[
\text{WP}(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \wedge \\
\forall \vec{v}, \ (Post[x_i \leftarrow t_i, w_j \leftarrow v_j, w_j@Old \leftarrow w_j] \Rightarrow Q[w_j \leftarrow v_j])
\]
Example: isqrt(42)

Exercise: prove that \{true\}isqrt(42)\{result = 6\} holds.

```
function isqrt(x:int): int
    requires x ≥ 0
    ensures result ≥ 0 ∧
        sqr(result) ≤ x < sqr(result + 1)
body
    let ref res = 0 in
    let ref sum = 1 in
    while sum ≤ x do
        res := res + 1;
        sum := sum + 2 * res + 1
    done;
    res
```
Example: Incrementation

Exercise: Prove that $\{ res = 6 \} incr(36) \{ res = 42 \}$ holds.

```plaintext
val res: ref int

procedure incr(x:int)
    requires true
    writes res
    ensures res = res@Old + x
```
Soundness of WP

Assuming that for each function defined as

function \( f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \)
requires \( Pre \)
writes \( \vec{w} \)
ensures \( Post \)
body \( Body \)

we have

- variables assigned in \( Body \) belong to \( \vec{w} \),
- \( \models Pre \Rightarrow WP(Body, Post)[w_i@Old \leftarrow w_i] \) holds,

then for any formulas \( P \) and \( Q \) and any expression \( e \),
\( \{ P \} e \{ Q \} \) is a valid triple if \( \models P \Rightarrow WP(e, Q) \).
Soundness Proof

To prove soundness of WP rules:

1. If $\Sigma, \Pi \models WP(e, Q)$ and $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$, then $\Sigma', \Pi' \models WP(e', Q)$.

   By structural induction on $e$. 
Soundness Proof

To prove soundness of WP rules:

1. If $\Sigma, \Pi \models WP(e, Q)$ and $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$, then $\Sigma', \Pi' \models WP(e', Q)$.
   By structural induction on $e$.

2. If $\Sigma, \Pi \models WP(e, Q)$ and $e$ is not a value, then there exists $\Sigma', \Pi', e'$ such that $\Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e'$.
   By predicate induction on $\rightsquigarrow$. 

Monotony lemma:
Given an expression $e$ and its assigned variables $\vec{w}$, if $\Sigma, \Pi \models \forall \vec{v}, (P \Rightarrow Q)[w_i \leftarrow v_i]$, then $\Sigma, \Pi \models WP(e, P) \Rightarrow WP(e, Q)$. 

Soundness Proof

To prove soundness of WP rules:

1. If \( \Sigma, \Pi \models WP(e, Q) \) and \( \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \), then \( \Sigma', \Pi' \models WP(e', Q) \).

   By structural induction on \( e \).

2. If \( \Sigma, \Pi \models WP(e, Q) \) and \( e \) is not a value, then there exists \( \Sigma', \Pi', e' \) such that \( \Sigma, \Pi, e \rightsquigarrow \Sigma', \Pi', e' \).

   By predicate induction on \( \rightsquigarrow \).

Monotony lemma:

Given an expression \( e \) and its assigned variables \( \vec{w} \), if \( \Sigma, \Pi \models \forall \vec{w}, (P \Rightarrow Q)[w_i \leftarrow v_i] \), then \( \Sigma, \Pi \models WP(e, P) \Rightarrow WP(e, Q) \).
Functions Raising Exceptions

A generalized contract has the form

function $f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau$
  requires $Pre$
  raises $E_1 \cdots E_k$
  writes $\vec{w}$
  ensures $Post \mid E_1 \rightarrow Post_1 \mid \cdots \mid E_k \rightarrow Post_k$

In the WP, the implication $Post[\ldots] \Rightarrow Q$ must be replaced by a conjunction of implications:

$$(Post[\ldots] \Rightarrow Q) \land \bigwedge_{i}(Post_i[\ldots] \Rightarrow R_i)$$
Example: Exact Square Root

exception NotSquare

function isqrt(x:int): int
  requires true
  raises NotSquare
  ensures result ≥ 0 ∧ sqr(result) = x
    | NotSquare → forall n:int. sqr(n) ≠ x

body
  if x < 0 then raise NotSquare;
  let ref res = 0 in
  let ref sum = 1 in
  while sum ≤ x do
    res := res + 1;
    sum := sum + 2 * res + 1
  done;
  if res * res ≠ x then raise NotSquare;
  res
Recursive Functions: Termination

If a function is recursive, termination of call can be proved, provided that the function is annotated with a variant.

function \( f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau \)
requires \( Pre \)
variant \( var \) for \( \prec \)
writes \( \vec{w} \)
ensures \( Post \)
body \( Body \)

WP for function call:

\[
WP(f(t_1, \ldots, t_n), Q) = Pre[x_i \leftarrow t_i] \land var[x_i \leftarrow t_i] \prec var@Init \land \\
\forall \vec{y}, (Post[x_i \leftarrow t_i][w_j \leftarrow y_j][w_j@Old \leftarrow w_j] \Rightarrow Q[w_j \leftarrow y_j])
\]

with \( Init \) a label assumed to be present at the start of \( Body \).
Example: Division

Exercise: find adequate specifications.

```
function div(x:int,y:int): int
  requires ?
  variant ?
  writes ?
  ensures ?
```
Example: McCarthy’s 91 Function

\[ f_{91}(n) = \text{if } n \leq 100 \text{ then } f_{91}(f_{91}(n + 11)) \text{ else } n - 10 \]

Exercise: find adequate specifications.

```function f_{91}(n:\text{int}): \text{int} 
  \text{requires} ?
  \text{variant} ?
  \text{writes} ?
  \text{ensures} ?
\text{body}
  \text{if } n \leq 100 \text{ then } f_{91}(f_{91}(n + 11)) \text{ else } n - 10
```