

Languages-beta: MiniJava-Dynamics *

The P_LanCompS Project

MiniJava-Dynamics.cbs | PLAIN | PRETTY

OUTLINE

1 Programs

2 Declarations

- Classes
- Variables
- Types
- Methods
- Formals

3 Statements

4 Expressions

5 Lexemes

Language "MiniJava"

1 Programs

Syntax $P : \text{program} ::= \text{main-class class-declaration}^*$

$MC : \text{main-class} ::=$ `'class' identifier '{'`
`'public' 'static' 'void' 'main' '(' 'String' '[' ']' identifier ')' '{'`
`statement`
`'}'`
`'}'`

Semantics $\text{run}[\![P : \text{program}]\!] : \Rightarrow \text{null-type}$

Rule $\text{run}[\![\text{'class' } ID_1 \text{'{'}$
 $\text{'public' 'static' 'void' 'main' '(' 'String' '[' ']' } ID_2 \text{'{'}$
 S
 $\text{'}'$
 $\text{'}'$
 $CD^*]\!] =$
 $\text{scope}(\text{recursive}(\text{bound-names}[\![CD^*]\!], \text{declare-classes}[\![CD^*]\!]),$
 $\text{execute}[\![S]\!])$

ID_1 and ID_2 are not referenced in S or CD^*

*Suggestions for improvement: plancomps@gmail.com.
Reports of issues: <https://github.com/plancomps/CBS-beta/issues>.

2 Declarations

Classes

Syntax $CD : \text{class-declaration} ::= \text{'class' identifier ('extends' identifier)? '{'}$
var-declaration*
method-declaration*
}'

Semantics $\text{bound-names} \llbracket CD^* : \text{class-declaration}^* \rrbracket : \Rightarrow \text{sets(ids)}$
Rule $\text{bound-names} \llbracket \text{'class' } ID_1 \text{'{' } VD^* MD^* \text{'}} \rrbracket = \{\text{id} \llbracket ID_1 \rrbracket\}$
Rule $\text{bound-names} \llbracket \text{'class' } ID_1 \text{'extends' } ID_2 \text{'{' } VD^* MD^* \text{'}} \rrbracket = \{\text{id} \llbracket ID_1 \rrbracket\}$
Rule $\text{bound-names} \llbracket \rrbracket = \{\}$
Rule $\text{bound-names} \llbracket CD CD^+ \rrbracket =$
 $\text{set-unite}(\text{bound-names} \llbracket CD \rrbracket, \text{bound-names} \llbracket CD^+ \rrbracket)$

Semantics $\text{declare-classes} \llbracket CD^* : \text{class-declaration}^* \rrbracket : \Rightarrow \text{envs}$
Rule $\text{declare-classes} \llbracket \text{'class' } ID_1 \text{'{' } VD^* MD^* \text{'}} \rrbracket =$
 $\{\text{id} \llbracket ID_1 \rrbracket \mapsto$
 class(
 thunk closure
 reference object(
 fresh-atom,
 $\text{id} \llbracket ID_1 \rrbracket,$
 $\text{declare-variables} \llbracket VD^* \rrbracket),$
 $\text{declare-methods} \llbracket MD^* \rrbracket)\}$
Rule $\text{declare-classes} \llbracket \text{'class' } ID_1 \text{'extends' } ID_2 \text{'{' } VD^* MD^* \text{'}} \rrbracket =$
 $\{\text{id} \llbracket ID_1 \rrbracket \mapsto$
 class(
 thunk closure
 reference object(
 fresh-atom,
 $\text{id} \llbracket ID_1 \rrbracket,$
 $\text{declare-variables} \llbracket VD^* \rrbracket,$
 $\text{dereference force class-instantiator bound id} \llbracket ID_2 \rrbracket),$
 $\text{declare-methods} \llbracket MD^* \rrbracket,$
 $\text{id} \llbracket ID_2 \rrbracket)\}$
Rule $\text{declare-classes} \llbracket \rrbracket = \text{map()}$
Rule $\text{declare-classes} \llbracket CD CD^+ \rrbracket =$
 $\text{collateral}(\text{declare-classes} \llbracket CD \rrbracket, \text{declare-classes} \llbracket CD^+ \rrbracket)$

Variables

Syntax $VD : \text{var-declaration} ::= \text{type identifier ';'}$

Semantics $\text{declare-variables} \llbracket VD^* : \text{var-declaration}^* \rrbracket : \Rightarrow \text{envs}$

Rule $\text{declare-variables} \llbracket T \ ID \ ;' \rrbracket =$
 $\{ \text{id} \llbracket ID \rrbracket \mapsto$
 $\quad \text{allocate-initialised-variable}(\text{type} \llbracket T \rrbracket, \text{initial-value} \llbracket T \rrbracket) \}$

Rule $\text{declare-variables} \llbracket \rrbracket = \text{map}(\)$

Rule $\text{declare-variables} \llbracket VD \ VD^+ \rrbracket =$
 $\text{collateral}(\text{declare-variables} \llbracket VD \rrbracket, \text{declare-variables} \llbracket VD^+ \rrbracket)$

Types

Syntax $T : \text{type} ::=$ `'int' '[' ']'`
`| 'boolean'`
`| 'int'`
`| identifier`

Semantics $\text{type} \llbracket T : \text{type} \rrbracket : \Rightarrow \text{types}$

Rule $\text{type} \llbracket \text{'int' '[' ']'} \rrbracket = \text{vectors}(\text{variables})$

Rule $\text{type} \llbracket \text{'boolean'} \rrbracket = \text{booleans}$

Rule $\text{type} \llbracket \text{'int'} \rrbracket = \text{integers}$

Rule $\text{type} \llbracket ID \rrbracket = \text{pointers}(\text{objects})$

Semantics $\text{initial-value} \llbracket T : \text{type} \rrbracket : \Rightarrow \text{minijava-values}$

Rule $\text{initial-value} \llbracket \text{'int' '[' ']'} \rrbracket = \text{vector}(\)$

Rule $\text{initial-value} \llbracket \text{'boolean'} \rrbracket = \text{false}$

Rule $\text{initial-value} \llbracket \text{'int'} \rrbracket = 0$

Rule $\text{initial-value} \llbracket ID \rrbracket = \text{pointer-null}$

Methods

Syntax $MD : \text{method-declaration} ::=$ `'public' type identifier '(' formal-list? ')'` `'{'`
`var-declaration*`
`statement*`
`'return' expression ';' ;`
`'}'`

Type methods
 $\rightsquigarrow \text{functions}(\text{tuples}(\text{references}(\text{objects}), \text{minijava-values}^*), \text{minijava-values})$

Semantics $\text{declare-methods}[\![MD^* : \text{method-declaration}^*]\!] : \Rightarrow \text{envs}$

Rule $\text{declare-methods}[\![\text{'public' } T ID \text{'(' } FL^? \text{')' ' \{ ' } VD^* S^* \text{'return' } E \text{' ;' ' \}' }]\!] =$

$\{\text{id}[\![ID]\!] \mapsto$
 $\quad \text{function closure scope(}$
 $\quad \quad \text{collateral(}$
 $\quad \quad \quad \text{match(}$
 $\quad \quad \quad \quad \text{given,}$
 $\quad \quad \quad \quad \text{tuple(}$
 $\quad \quad \quad \quad \quad \text{pattern abstraction}$
 $\quad \quad \quad \quad \quad \quad \{\text{"this"} \mapsto$
 $\quad \quad \quad \quad \quad \quad \quad \text{allocate-initialised-variable(pointers(objects), given)),}$
 $\quad \quad \quad \quad \text{bind-formals}[\![FL^?]\!]),$
 $\quad \quad \quad \text{object-single-inheritance-feature-map}$
 $\quad \quad \quad \text{checked dereference first tuple-elements given,}$
 $\quad \quad \quad \text{declare-variables}[\![VD^*]\!]),$
 $\quad \quad \text{sequential(execute}[\![S^*]\!], \text{evaluate}[\![E]\!]))\}$

Rule $\text{declare-methods}[\![]\!] = \text{map}(\)$

Rule $\text{declare-methods}[\![MD MD^+]\!] =$

$\text{collateral}(\text{declare-methods}[\![MD]\!], \text{declare-methods}[\![MD^+]\!])$

Formals

Syntax $FL : \text{formal-list} ::= \text{type identifier (' , ' formal-list)}^?$

Semantics $\text{bind-formals}[\![FL^? : \text{formal-list}^?]\!] : \Rightarrow \text{patterns}^*$

Rule $\text{bind-formals}[\![T ID]\!] =$

$\text{pattern abstraction}$
 $\quad \{\text{id}[\![ID]\!] \mapsto$
 $\quad \quad \text{allocate-initialised-variable}(\text{type}[\![T]\!], \text{given})\}$

Rule $\text{bind-formals}[\![T ID \text{' , ' } FL]\!] = \text{bind-formals}[\![T ID]\!], \text{bind-formals}[\![FL]\!]$

Rule $\text{bind-formals}[\![]\!] = ()$

3 Statements

Syntax $S : \text{statement} ::= \text{' \{ ' statement}^* \text{'}}$

$| \text{'if' ' (' expression ') ' statement 'else' statement}$
 $| \text{'while' ' (' expression ') ' statement}$
 $| \text{'System' ' . ' out ' . ' println ' (' expression ') ' ;'}$
 $| \text{identifier ' = ' expression ' ;'}$
 $| \text{identifier ' [' expression '] ' = ' expression ' ;'}$

Semantics $\text{execute}[\![S^* : \text{statement}^*]\!] : \Rightarrow \text{null-type}$

Rule $\text{execute}[\![\{ S^* \}]\!] = \text{execute}[\![S^*]\!]$

Rule $\text{execute}[\![\text{if } (E) S_1 \text{ else } S_2]\!] =$
 $\text{if-true-else}(\text{evaluate}[\![E]\!], \text{execute}[\![S_1]\!], \text{execute}[\![S_2]\!])$

Rule $\text{execute}[\![\text{while } (E) S]\!] =$
 $\text{while-true}(\text{evaluate}[\![E]\!], \text{execute}[\![S]\!])$

Rule $\text{execute}[\![\text{System } . \text{out} . \text{println } (E) ;]\!] =$
 $\text{print}(\text{to-string } \text{evaluate}[\![E]\!], "\n")$

Rule $\text{execute}[\![ID = E ;]\!] =$
 $\text{assign}(\text{bound id } [ID], \text{evaluate}[\![E]\!])$

Rule $\text{execute}[\![ID [E_1] = E_2 ;]\!] =$
 $\text{assign}(\text{checked index}(\text{integer-add}(\text{evaluate}[\![E_1]\!], 1), \text{vector-elements assigned bound id } [ID], \text{evaluate}[\![E_2]\!]))$

Rule $\text{execute}[\![]] = \text{null}$

Rule $\text{execute}[\![S S^+]\!] = \text{sequential}(\text{execute}[\![S]\!], \text{execute}[\![S^+]\!])$

4 Expressions

Syntax $E : \text{expression} ::=$ expression $\&\&$ expression
 | expression $<$ expression
 | expression $+$ expression
 | expression $-$ expression
 | expression $*$ expression
 | expression $[$ expression $]$
 | expression $.$ length
 | expression $.$ identifier $($ expression-list? $)$
 | integer-literal
 | true
 | false
 | identifier
 | this
 | $\text{new int } [\text{expression}]$
 | $\text{new identifier } ()$
 | $!$ expression
 | $($ expression $)$

Type minijava-values
 \rightsquigarrow booleans | integers | vectors(variables) | pointers(objects)

Semantics $\text{evaluate}[\![E : \text{expression}]\!] : \Rightarrow \text{minijava-values}$

$\text{evaluate}[\![_]\!]$ is a well-typed function term only when $_$ is a well-typed MiniJava expression.

Rule `evaluate`[[E_1 ' && ' E_2]] =
 if-true-else(`evaluate`[[E_1]], `evaluate`[[E_2]], `false`)
Rule `evaluate`[[E_1 ' < ' E_2]] =
 integer-is-less(`evaluate`[[E_1]], `evaluate`[[E_2]])
Rule `evaluate`[[E_1 ' + ' E_2]] =
 integer-add(`evaluate`[[E_1]], `evaluate`[[E_2]])
Rule `evaluate`[[E_1 ' - ' E_2]] =
 integer-subtract(`evaluate`[[E_1]], `evaluate`[[E_2]])
Rule `evaluate`[[E_1 ' * ' E_2]] =
 integer-multiply(`evaluate`[[E_1]], `evaluate`[[E_2]])
Rule `evaluate`[[E_1 ' [' E_2 '] ']] =
 assigned checked index(
 integer-add(`evaluate`[[E_2]], 1),
 vector-elements `evaluate`[[E_1]])
Rule `evaluate`[[E ' . ' 'length']] =
 length vector-elements `evaluate`[[E]]
Rule `evaluate`[[E ' . ' ID ' (' $EL?$ ') ']] =
 give(
 `evaluate`[[E]],
 apply(
 lookup(
 class-name-single-inheritance-feature-map
 object-class-name checked dereference given,
 id[[ID]]),
 tuple(given, `evaluate-actuals`[[$EL?$]]))
)
Rule `evaluate`[[IL]] = integer-value[[IL]]
Rule `evaluate`[['true']] = `true`
Rule `evaluate`[['false']] = `false`
Rule `evaluate`[[ID]] = assigned bound id[[ID]]
Rule `evaluate`[['this']] = assigned bound "this"
Rule `evaluate`[['new' 'int' ' [' E '] ']] =
 vector(
 interleave-repeat(
 allocate-initialised-variable(integers, 0), 1, `evaluate`[[E]])
)
Rule `evaluate`[['new' ID ' (' ' ') ']] =
 force class-instantiator bound id[[ID]]
Rule `evaluate`[[' ! ' E]] = not `evaluate`[[E]]
Rule `evaluate`[[' (' E ') ']] = `evaluate`[[E]]

Syntax EL : expression-list ::= expression (' , ' expression-list) ?

Semantics `evaluate-actuals`[[$EL?$: expression-list ?]] : (\Rightarrow minijava-values) *

Rule `evaluate-actuals`[[E]] = `evaluate`[[E]]

Rule `evaluate-actuals`[[E ' , ' EL]] = `evaluate`[[E]], `evaluate-actuals`[[EL]]

Rule `evaluate-actuals`[[]] = ()

5 Lexemes

Lexis $ID : \text{identifier} ::= \text{letter} (\text{letter} \mid \text{digit} \mid \text{'_'})^*$

Semantics $\text{id} \llbracket ID : \text{identifier} \rrbracket : \Rightarrow \text{ids}$
 $= \text{"ID"}$

Lexis $IL : \text{integer-literal} ::= \text{digit}^+$
 $\text{letter} ::= \text{'a' - 'z'} \mid \text{'A' - 'Z'}$
 $\text{digit} ::= \text{'0' - '9'}$

Semantics $\text{integer-value} \llbracket IL : \text{integer-literal} \rrbracket : \Rightarrow \text{integers}$
 $= \text{decimal-natural "IL"}$