

We take the common nonterminals *digit*, *digits*, *lower*, *upper*, *alphanum*, and *alphanums* to be predefined with their obvious meaning (plurality such as *digits* implies at least one digit). Then the grammar for the language is:

- *Class may extend an object or default to extending Object*

$\langle \text{class} \rangle \Rightarrow$
 $\quad \mathbf{class} \langle \text{class id} \rangle \langle \text{extend} \rangle \langle \text{scope} \rangle \langle \text{class decls} \rangle$
 $\langle \text{extend} \rangle \Rightarrow$
 $\quad \epsilon$
 $\quad | \mathbf{extends} \langle \text{class id} \rangle$

- *A class can be split up into five sections \Rightarrow private protected public members refinements and the main*

$\langle \text{class decls} \rangle \Rightarrow$
 $\quad \langle \text{class decl} \rangle$
 $\quad | \langle \text{class decls} \rangle \langle \text{class decl} \rangle$
 $\langle \text{class decl} \rangle \Rightarrow$
 $\quad \langle \text{refinement} \rangle$
 $\quad | \langle \text{access group} \rangle$
 $\quad | \langle \text{main} \rangle$

- *Refinements are named method dot refinement*

$\langle \text{refinement} \rangle \Rightarrow$
 $\quad \mathbf{refinement} \langle \text{scope} \rangle \langle \text{refines} \rangle$
 $\langle \text{refines} \rangle \Rightarrow$
 $\quad \langle \text{refine} \rangle$
 $\quad | \langle \text{refines} \rangle \langle \text{refine} \rangle$
 $\langle \text{refine} \rangle \Rightarrow$
 $\quad \langle \text{refine sig} \rangle \langle \text{scope} \rangle \langle \text{statements} \rangle$
 $\langle \text{refine sig} \rangle \Rightarrow$
 $\quad \langle \text{return type} \rangle \langle \text{var id} \rangle . \langle \text{var id} \rangle \langle \text{params} \rangle$

- *Access groups contain all the members of a class*

$\langle \text{access group} \rangle \Rightarrow$
 $\quad \langle \text{access type} \rangle \langle \text{scope} \rangle \langle \text{members} \rangle$
 $\langle \text{access type} \rangle \Rightarrow$
 $\quad \mathbf{private}$
 $\quad | \mathbf{protected}$
 $\quad | \mathbf{public}$
 $\langle \text{members} \rangle \Rightarrow$
 $\quad \langle \text{member} \rangle$

| $\langle \text{members} \rangle \langle \text{member} \rangle$
 $\langle \text{member} \rangle \Rightarrow$
 $\langle \text{ivar} \rangle$
 | $\langle \text{method} \rangle$
 | $\langle \text{init} \rangle$
 $\langle \text{ivar} \rangle \Rightarrow$
 $\langle \text{var decl} \rangle$
 $\langle \text{method} \rangle \Rightarrow$
 $\langle \text{method sig} \rangle \langle \text{scope} \rangle \langle \text{statements} \rangle$
 $\langle \text{method sig} \rangle \Rightarrow$
 $\langle \text{return type} \rangle \langle \text{var id} \rangle$
 | $\langle \text{params} \rangle$
 $\langle \text{init} \rangle \Rightarrow$
 $\langle \text{init sig} \rangle \langle \text{scope} \rangle \langle \text{statements} \rangle$
 $\langle \text{init sig} \rangle \Rightarrow$
 init $\langle \text{params} \rangle$

- *The main is special – it belongs to no object and is what can be used to start an execution*

$\langle \text{main} \rangle \Rightarrow$
 $\langle \text{main sig} \rangle \langle \text{scope} \rangle \langle \text{statements} \rangle$
 $\langle \text{main sig} \rangle \Rightarrow$
 main
 | **main** (**String**[] $\langle \text{var id} \rangle$)

- *Finally the meat and potatoes*

$\langle \text{statements} \rangle \Rightarrow$
 $\langle \text{statement} \rangle$
 | $\langle \text{statements} \rangle \langle \text{statement} \rangle$
 $\langle \text{statement} \rangle \Rightarrow$
 $\langle \text{local} \rangle$
 | $\langle \text{refine stmt} \rangle$
 | $\langle \text{assignment} \rangle$
 | $\langle \text{conditional} \rangle$
 | $\langle \text{loop} \rangle$
 | $\langle \text{expression} \rangle$

- *Local variables can be uninitialized (end up null) or initialized*

$\langle \text{local} \rangle \Rightarrow$
 $\langle \text{var decl} \rangle$
 | $\langle \text{var decl} \rangle := \langle \text{expression} \rangle$

- *Refine statements need not return type information*

$\langle \text{refine stmt} \rangle \Rightarrow$
 $\quad \mathbf{refine} \ \langle \text{specialize} \rangle$

- *Assignment works as in most languages – lvalues receive the results of expressions*

$\langle \text{assignment} \rangle \Rightarrow$
 $\quad \langle \text{lvalue} \rangle := \langle \text{expression} \rangle$
 $\langle \text{lvalue} \rangle \Rightarrow$
 $\quad \langle \text{var id} \rangle$
 $\quad | \ \mathbf{this} \ . \ \langle \text{var id} \rangle$
 $\quad | \ \langle \text{var id} \rangle \ . \ \langle \text{var id} \rangle$

- *If and else are the same as in every other language*

$\langle \text{conditional} \rangle \Rightarrow$
 $\quad \langle \text{if} \rangle$
 $\quad | \ \langle \text{if} \rangle \langle \text{else} \rangle$
 $\langle \text{if} \rangle \Rightarrow$
 $\quad \mathbf{if} \ (\ \langle \text{expression} \rangle \) \langle \text{scope} \rangle \langle \text{statements} \rangle$
 $\langle \text{else} \rangle \Rightarrow$
 $\quad \mathbf{else} \langle \text{scope} \rangle$

- *C and java style loop constructs*

$\langle \text{loop} \rangle \Rightarrow$
 $\quad \langle \text{loop head} \rangle \langle \text{scope} \rangle \langle \text{statements} \rangle$
 $\langle \text{loop head} \rangle \Rightarrow$
 $\quad \mathbf{for} \ (\ \langle \text{assignment} \rangle \ , \ \langle \text{expression} \rangle \ , \ \langle \text{assignment} \rangle \)$
 $\quad | \ \mathbf{while} \ (\ \langle \text{expression} \rangle \)$

- *An expression is anything that can result in a value – note that assignment is not an expression (should we change?)*

$\langle \text{expression} \rangle \Rightarrow$
 $\quad \langle \text{invocation} \rangle$
 $\quad | \ \langle \text{arithmetic} \rangle$
 $\quad | \ \langle \text{array expression} \rangle$
 $\quad | \ \langle \text{test} \rangle$
 $\quad | \ \langle \text{instantiate} \rangle$
 $\quad | \ \langle \text{refine expr} \rangle$
 $\quad | \ \langle \text{literal} \rangle$
 $\quad | \ \langle \text{var id} \rangle$
 $\quad | \ (\ \langle \text{expression} \rangle \)$
 $\quad | \ \mathbf{null}$

- *Method invocations always have a receiver*

$\langle \text{invocation} \rangle \Rightarrow$
 $\langle \text{receiver} \rangle . \langle \text{invoke} \rangle$
 $\langle \text{receiver} \rangle \Rightarrow$
 $\quad \mathbf{this}$
 $\quad | \langle \text{var id} \rangle$
 $\langle \text{invoke} \rangle \Rightarrow$
 $\quad \langle \text{var id} \rangle ()$
 $\quad | \langle \text{var id} \rangle (\langle \text{args} \rangle)$

- *Basic arithmetic can and will be done!*

$\langle \text{arithmetic} \rangle \Rightarrow$
 $\quad \langle \text{expression} \rangle \langle \text{bin op} \rangle \langle \text{expression} \rangle$
 $\quad | \langle \text{unary op} \rangle \langle \text{expression} \rangle$
 $\langle \text{bin op} \rangle \Rightarrow$
 $\quad +$
 $\quad | -$
 $\quad | *$
 $\quad | /$
 $\quad | \%$
 $\langle \text{unary op} \rangle \Rightarrow$
 $\quad -$

- *Build an array on the fly (should we leave this out? TODO – ARRAY ASSIGNMENT)*

$\langle \text{array expression} \rangle \Rightarrow$
 $\quad [\langle \text{args} \rangle]$

- *Common boolean predicates*

$\langle \text{test} \rangle \Rightarrow$
 $\quad \langle \text{expression} \rangle \langle \text{bin pred} \rangle \langle \text{expression} \rangle$
 $\quad | \langle \text{unary pred} \rangle \langle \text{expression} \rangle$
 $\quad | \mathbf{refinable} (\langle \text{var id} \rangle)$
 $\langle \text{bin pred} \rangle \Rightarrow$
 $\quad \mathbf{and}$
 $\quad | \mathbf{or}$
 $\quad | \mathbf{xor}$
 $\quad | <$
 $\quad | <=$
 $\quad | =$
 $\quad | !=$
 $\quad | >=$
 $\quad | >$

$\langle \text{unary pred} \rangle \Rightarrow$
 $!$

- *Making something*

$\langle \text{instantiate} \rangle \Rightarrow$
 $\langle \text{object instantiate} \rangle$
 $| \langle \text{array instantiate} \rangle$
 $\langle \text{object instantiate} \rangle \Rightarrow$
 $\mathbf{new} \langle \text{class id} \rangle$
 $| \mathbf{new} \langle \text{class id} \rangle (\langle \text{args} \rangle)$
 $\langle \text{array instantiate} \rangle \Rightarrow$
 $\mathbf{new} \langle \text{type} \rangle [\langle \text{digits} \rangle]$

- *Refinement takes a specific specialization and notes the required return type*

$\langle \text{refine expr} \rangle \Rightarrow$
 $\mathbf{refine} \langle \text{specialize} \rangle \mathbf{to} \langle \text{type} \rangle$
 $\langle \text{specialize} \rangle \Rightarrow$
 $\langle \text{var id} \rangle ()$
 $| \langle \text{var id} \rangle (\langle \text{args} \rangle)$

- *Literally necessary*

$\langle \text{literal} \rangle \Rightarrow$
 $\langle \text{int lit} \rangle$
 $| \langle \text{bool lit} \rangle$
 $| \langle \text{float lit} \rangle$
 $| \langle \text{string lit} \rangle$
 $\langle \text{float lit} \rangle \Rightarrow$
 $\langle \text{digits} \rangle . \langle \text{digits} \rangle$
 $\langle \text{int lit} \rangle \Rightarrow$
 $\langle \text{sign} \rangle \langle \text{digits} \rangle$
 $\langle \text{bool lit} \rangle \Rightarrow$
 \mathbf{true}
 $| \mathbf{false}$
 $\langle \text{string lit} \rangle \Rightarrow$
 $\text{“} \langle \text{string escape seq} \rangle \text{”}$

- *Params and args are as expected*

$\langle \text{params} \rangle \Rightarrow$
 ϵ
 $| (\langle \text{paramlist} \rangle)$
 $\langle \text{paramlist} \rangle \Rightarrow$
 $\langle \text{var decl} \rangle$

$\mid \langle \text{paramlist} \rangle , \langle \text{var decl} \rangle$
 $\langle \text{args} \rangle \Rightarrow$
 $\langle \text{expression} \rangle$
 $\mid \langle \text{args} \rangle , \langle \text{expression} \rangle$

- *All the basic stuff we've been saving up until now*

$\langle \text{var decl} \rangle \Rightarrow$
 $\langle \text{type} \rangle \langle \text{var id} \rangle$
 $\langle \text{return type} \rangle \Rightarrow$
 \mathbf{unit}
 $\mid \langle \text{type} \rangle$
 $\langle \text{type} \rangle \Rightarrow$
 $\langle \text{class id} \rangle$
 $\mid \langle \text{type} \rangle []$
 $\langle \text{class id} \rangle \Rightarrow$
 $\langle \text{upper} \rangle$
 $\mid \langle \text{upper} \rangle \langle \text{alphanums} \rangle$
 $\langle \text{var id} \rangle \Rightarrow$
 $\langle \text{lower} \rangle$
 $\mid \langle \text{lower} \rangle \langle \text{alphanums} \rangle$