(Not Quite) Minijava

CMCS22620, Spring 2004

April 5, 2004

1 Syntax

program	→	*mainclass classdecl**
mainclass	→	**class** *identifier* { **public static void main (** **String []** *identifier* **)** *block* }
classdecl	→	**class** *identifier* (**extends** *identifier*)$_{\mathrm{opt}}$ { *instvardecl** *methoddecl** }
instvardecl	→	**public**$_{\mathrm{opt}}$ *type var* (**,** *var*)* **;**
var	→	*identifier* (**[]**)*
methoddecl	→	**public**$_{\mathrm{opt}}$ *rtyp identifier* **(** *formal* (**,** *formal*)* **)** *block*
rtyp	→	*type* | **void**
formal	→	*type var*
block	→	{ (*locvardecl* **;** | *statement*)* }
locvardecl	→	*type locvar* (**,** *locvar*)$_{\mathrm{opt}}$
locvar	→	*var* (**=** *init*)$_{\mathrm{opt}}$
init	→	*exp* | { *init* (**,** *init*)* }
statement	→	*block*
	|	**if (** *exp* **)** *statement* (**else** *statement*)$_{\mathrm{opt}}$
	|	**while (** *exp* **)** *statement*
	|	**do** *statement* **while (** *exp* **) ;**
	|	**for (** *forinit* **;** *exp*$_{\mathrm{opt}}$ **;** *explist*$_{\mathrm{opt}}$ **)** *statement*
	|	*exp* **;**
	|	**return** *exp*$_{\mathrm{opt}}$ **;**
	|	**;**
	|	*identifier* **:** *statement*
	|	**break** *identifier*$_{\mathrm{opt}}$ **;**
	|	**continue** *identifier*$_{\mathrm{opt}}$ **;**
forinit	→	*explist*$_{\mathrm{opt}}$
	|	*locvardecl*

explist	$\rightarrow$	*exp* (**,** *exp*)*
exp	$\rightarrow$	*exp binop exp*
	$\mid$	(*exp* **.**)$_{\text{opt}}$ *identifier* **(** *explist*$_{\text{opt}}$ **)**
	$\mid$	*integer*
	$\mid$	**true** $\mid$ **false** $\mid$ **this** $\mid$ **null**
	$\mid$	**new** *identifier* **( )**
	$\mid$	**new** *type* **[** *exp* **]**
	$\mid$	*uop exp*
	$\mid$	*exp* **?** *exp* **:** *exp*
	$\mid$	**(** *exp* **)**
	$\mid$	*lvalue*
	$\mid$	*lvalue* **=** *exp*
binop	$\rightarrow$	&& $\mid$ || $\mid$ < $\mid$ > $\mid$ <= $\mid$ >= $\mid$ == $\mid$!= $\mid$ + $\mid$ - $\mid$ *
uop	$\rightarrow$	! $\mid$ -
lvalue	$\mid$	*identifier*
	$\mid$	*exp* **.** *identifier*
	$\mid$	*exp* **[** *exp* **]**
type	$\rightarrow$	**int** $\mid$ **boolean** $\mid$ *identifier* $\mid$ *type* **[]**
identifier	$\rightarrow$	...
integer	$\rightarrow$	...

2 Classes and subtyping

The language has **no overloading**; each class has at most one method by any given name. Let B extend A. If A has a method `f` and B wants to override `f` with its own version, then the type of B's `f` must be a *subtype* of A's `f`.

For `f`'s type to be a subtype of `g`'s type, `f` and `g` must take equal number of arguments. If `f` has a result, then so must `g` (and vice versa), and the type of `f`'s result must be a subtype of the type of `g`'s result. Moreover, the type of each of `g`'s arguments must be a subtype of the type of the corresponding argument of `f`. (Notice the role reversal here! This is called *contravariance*.)

A class type B is a subtype of another class type A if B directly or indirectly extends A. Otherwise, an array type t`[]` is a subtype of another array type u`[]` if and only if t is a subtype of u. (This is, in some sense, a design bug in the Java language since it does not guarantee safety. To restore safety, assignment to arrays often require runtime type tests.)

3 Expressions

constants Minijava programs can use the following constants:

boolean **true**, **false**

numerical *integer*

array and object **null**

this refers to the "current" object—the object o in the method call o`.f(`$e_1, \ldots, e_k$`)` that invoked the current method

identifier an identifier n refers to the variable n that is currently in scope; if n is an instance variable (and therefore declared in the class corresponding to the current object), then n is equivalent to **this**`.`n; this construct is an *lvalue* and therefore can appear on the left-hand side of the assignment operator

selection the syntax o`.`n denotes access to an instance variable n in object o; which (of possibly many) instance variables by the name n is meant depends on the *compile-time* type of expression o; this construct is an *lvalue*

array length the syntax a`.length` refers to the length (number of elements) of an array a; although syntactically identical to selection and therefore an *lvalue*, this construct is read-only

array subscript the syntax a`[`i`]` refers to the i-th element of array a; this construct is an *lvalue*

self method call the syntax f`(`$e_1, \ldots, e_k$`)` (where f is an identifier) is equivalent to **this**`.`f`(`$e_1, \ldots, e_k$`)`

method invocation the syntax o`.`f`(`$e_1, \ldots, e_k$`)` invokes the method named f in object o; which (of possibly many) methods by the name f is meant depends on the class that o was created from, i.e., the *runtime* type of o

object creation an expression of the form **new** c`()` returns a freshly allocated object of runtime type c (c must be an identifier referring to a class); all instance variables of the object are cleared (integers become 0, booleans become **false**, arrays and objects become **null**)

array creation an expression of the form **new** t`[`l`]` returns a freshly allocated array whose elements are of type t and whose size (number of elements) is l; all elements are cleared (integers become 0, booleans become **false**, arrays and objects become **null**)

binary operations In general, binary operations have the form $e_1 \otimes e_2$ where $\otimes$ is one of:

- *short-circuiting logical and:* `||` — boolean arguments and results
- *short-circuiting logical or:* `&&` — boolean arguments and results
- *equality tests:* `==` `!=` — arbitrary (matching) argument types, boolean result; in case of arrays and objects the test is for object identity (memory address equality)
- *comparisons:* `<` `>` `<=` `>=` — integer operands, boolean result
- *addition and subtraction:* `+` `-` — integer operands and result
- *multiplication:* `*` — integer operands and result

These operators are listed in order of increasing precedence.

unary operations There are two unary operations:

- *boolean negation:* `!`e — boolean argument, boolean result
- *arithmetic negation:* `-`e — integer argument, integer result

conditional expression An expression of the form e_1`?`e_2`:`e_3 evaluates the boolean condition e_1 and depending on the outcome evaluates either e_2 (when **true**) or e_3 (when **false**) and returns the respective value; the expression that is not needed does not get evaluated; notice that e_1`&&`e_2 and e_1`||`e_2 are equivalent to e_1`?`e_2`:`**false** and e_1`?`**true**`:`e_2, respectively.

assignment l`=`e assigns the value of e into the location denoted by lvalue l; the result (which is the value that was assigned) has the same type as l.

4 Statements

blocks Blocks are sequences of statements and variable declarations enclosed in curly braces $\{\}$. The scope of each variable declaration extends from the point of declaration until the end of the block.

conditional The conditional statement has a condition, a "then" branch, and an optional "else" branch. It works like in C or Java.

while loop The `while`-loop has a condition and a body. It alternates between evaluating the condition and executing the statement (beginning with the condition) as long as the condition is found to be **true**. If during the execution of the loop a `continue` statement is executed, then control is passed directly directly to the condition, thus starting a new round.

do loop The `do`-loop is like the `while`-loop, except execution starts with the body, so the body gets executed at least once. A `continue` statement jumps back to the beginning of the body, thus starting a new round.

for loop The `for`-loop consists of an optional initializer, an optional condition, an optional update part, and a body. The initializer is evaluated once, possibly declaring some loop-local variables which are available in all three other parts. Then the loop starts executing beginning with the condition followed by the body, then the update part (if present), and finally back to the condition. A missing condition is treated as **true**. The update part consists of a list of expressions which get evaluated for effect. (Usually these are used to update loop variables originally declared or initialized by the initializer.) The loop stops after the condition is found to be **false** for the first time. A `continue` statement jumps back to the update part, thus starting a new round.

expression An expression can be used where a statement is expected by simply following it with a semicolon.

return The `return` statement terminates the current method. If the return type of the method is `void`, then `return` does not accept an expression. Otherwise `return` requires an expression which is used to specify the return value of the method.

empty A semicolon by itself is an empty statement which does nothing.

labeled Any statement s can follow l`:` where l is the name of a label. The label can be used by `break`- and `continue`-statements within s to refer to s.

break A `break`-statement without a label terminates the innermost enclosing loop (`while`, `do`, or `for`) within the current method. (Such a loop must exist for the `break` to be legal.) A `break`-statement carrying a label l terminates the innermost enclosing statement s that is labeled with l. If there is no such statement, then `break` l`;` is illegal.

continue All `continue`-statements refer to an enclosing loop statement within the current method. If no label was specified, that loop is the innermost enclosing loop (which has to exist); if label l was given, the statement refers to the innermost enclosing loop labeled with l (which has to exist). The precise semantics of `continue` depends on the kind of loop it refer to. See the description of `while`, `do`, and `for` for details.