MiniC Language Reference Card

conventions used in reference card

$\sigma, \sigma_1, \sigma_2$	boolean expressions
τ, π	general expressions
n, m	compile-time constant expressions
α_1, α_2	data types

module import and implemenation

package pointedName	define root path for importing modules relative to current dir.
include pointedName	include textfile
// *comment*	single line comment
/* *comment* */	block comment (mult. lines)
function f(*vdcl*) : α { *stat* }	function f with variable declarations *vdcl*, body statement *stat* and result type α

variable declarations *vdcl*::=

general syntax is a comma-separated list of single declarations *type* $x_1, \ldots, x_n$, e.g. **nat** x1,x2, **int** z1,z2

data types *type*::=

bool	booleans
nat	unsigned integers (machine dependent)
int	signed integers (machine dependent)
[n]α	array having n elements of type α
$\alpha_1 * \ldots * \alpha_n$	tuple type

literals

boolean constants are **false** and **true**; examples for un- signed integers are 0,1,2,3,... while signed integers are ...,−2,−1,−0,+0,+1,+2,+3,...

expressions

type casts

(**nat**) τ	interprets τ as type **nat**
(**int**) τ	interprets τ as type **int**
(**bool**) τ	interprets τ as type **bool**

constructing and accessing compound types

$\tau[\pi]$	array access
[|$\tau_0, \ldots, \tau_{n-1}$|]	array of n values
$\tau.n$	tuple access
(|$\tau_0, \ldots, \tau_{n-1}$|)	tuple of n values

equality

τ_1 == τ_2	equality
τ_1 != τ_2	inequality

numeric relations (for both nat and int)

τ_1 < τ_2	less than
τ_1 <= τ_2	less than or equal to
τ_1 > τ_2	greater than
τ_1 >= τ_2	greater than or equal to

boolean operators

! σ	**not** σ	negation
σ_1 & σ_2	σ_1 **and** σ_2	conjunction
σ_1 | σ_2	σ_1 **or** σ_2	disjunction
σ_1 ^ σ_2	σ_1 **xor** σ_2	exclusive or
σ_1 −> σ_2	σ_1 **imp** σ_2	implication
σ_1 <−> σ_2	σ_1 **eqv** σ_2	equivalence

arithmetic operators (for both nat and int)

$\tau + \pi$	addition
$\tau - \pi$	subtraction
$\tau * \pi$	multiplication
τ / π	division
τ % π	modulo
abs(τ)	absolute value

function call

$f(\tau_1, \ldots, \tau_n)$;	call function f with parameter expressions $\tau_1, \ldots, \tau_n$

statements *stat*::=

atomic statements

$\lambda = \tau$	single word assignment
$\lambda_1, \lambda_2 = \tau$	double word assignment
[*name* :] **assert**(σ);	assertion
sync	thread synchronisation

composed statements

if (σ) S_1 [**else** S_2]	conditional statement
S_1 S_2	sequential execution
{ α x; S }	declare variable x of type α with scope S
do S **while**(σ)	repeat S while σ holds
while (σ)S	while σ holds, repeat S
for (i=m .. n) S	unconditional loop
return τ	return value τ

remarks on function calls

the following restrictions apply

- no recursive functions: a function is not allowed to call itself, not even via other function calls
- arguments of scalar types are provided via call-by-value, arrays and tuples via call-by-reference (hence the latter are potentially overwritten by the function)