FACILE LANGUAGE REFERENCE

This document is a complete reference for the Facile programming language as implemented by the current Facile compiler. It describes Facile's lexical specification, architecture description features, types, and semantic statements and expressions. Note that Facile source files are passed through the C pre-processor (**cpp**) before being given to the Facile compiler, so C pre-processor directives (e.g., `#define` or `#include`) can be used in Facile source files.

TYPOGRAPHICAL CONVENTIONS. The following typographical conventions are used in this reference:

- `fixed-width` for Facile code, and for reserved words and punctuation in Facile grammar descriptions.

- *italic* for non-terminals in Facile grammar descriptions.

- **bold** for terminal symbols that can have many values (e.g., **name**, **int**, and **float**).

Lexical Specification

Facile is case sensitive. Whitespace—e.g., space, tab, newline—is ignored except where it is needed as a token separator. Comments are treated as white space and are either written as `//` to the end of line, or between `/*` and `*/` delimiters. Comments written as `/*` … `*/` do not nest.

Below are lists of all the reserved words and all punctuation used in Facile:

```
array       continue    fun         return      val
as          default     if          sem         var
break       else        in          struct      where
by          extern      inline      switch      while
case        fields      pat         token       xor
const       for         queue       type

(   )   ,   .   ..  :   ;   =   ?   [   ]   _   {   }
$   %   &   &&  *   +   -   /   <<  >>  ^   ^^  |   ||  ~
<   <=  ==  !=  >=  >
```

Function, variable, type, and attribute names are any alpha-numeric words that are not already defined as reserved words. Names are separated by whitespace or punctuation (except for the wildcard mark '_' which can be part of a name). They can not have a numeric digit as their first character, although numeric digits can be used later in a name. I.e., names are of the form `[A-Za-z_][A-Za-z_0-9]`*.

Literal values are integers (in decimal, octal, hexadecimal, binary, or character form), floating point constants, and strings. Decimal and octal constants are written as `[1-9][0-9]`* and `0[0-7]`* respectively, and are interpreted as 32-bit unsigned values. Hexadecimal and binary constants are written as `0x[0-9A-Fa-f]`* and `0b[01]`* respectively, and are interpreted as unsigned values with a bit-widths just large enough to encode all the digits appearing in the literal. E.g., 0xFF represents the 8-bit wide integer with value 255, and 0b11001 is the 5-bit wide integer

with value 25. Character literals are interpreted as 1-byte (8-bit wide) unsigned integers, and are written as `'`*c*, where *c* is any single character or escape sequence. Escape sequences are newline `\n`, tab `\t`, double-quote `\"`, backslash `\\`, or any ASCII character number written in hexadecimal as `\[0-9A-Fa-f]`$^{\{1,2\}}$`x` (e.g., `'A`, `'\n`, `'\13x`).

Floating point values are written as `-`$^{?}$`[0-9]`*`.[0-9]`*`([Ee][-+]`$^{?}$`[0-9]`$^{+}$`)`$^{?}$ and are interpreted as 8-byte (64-bit) IEEE floating point values. There must be at least one digit before or after the decimal point. The following are some possible floating point literals: `5.0`, `.15`, `-7.`, and `1.0e-9`.

String literals are written as sequences of characters and character escapes between double-quote delimiters (`"`). The recognized character escapes are newline `\n`, tab `\t`, double-quote `\"`, backslash `\\`, and any ASCII character number written in hexadecimal as `\[0-9A-Fa-f]`$^{\{1,2\}}$`x`. Strings cannot span multiple lines. To include a newline in a string literal, use the `\n` character escape.

Architecture Description

A target instruction set architecture (ISA) is described using token, field, pattern, and semantic declarations. Tokens define fixed width groupings of bits in an instruction stream (also called a token stream) and fields are contiguous sequences of bits within a token. Patterns are sets of conditions on the values stored in fields within a token stream and are used to distinguish the binary encodings of instructions in an ISA. Semantic declarations map pattern names to semantic code.

This semantic code usually simulates target instructions, but can be anything the programmer desires.

global_stmt	→ `token` **name** `[` **int** `]` *fields_opt* `;`	token declaration
fields_opt	→ `fields` *field_dec_list*	field declarations
	→	
fields_dec_list	→ *fields_dec_list* `,` *field_dec*	
	→ *field_dec*	
field_dec	→ **name** **int** `:` **int**	bit range field
	→ **name** **int**	single bit field

A `token` declaration statement defines a token with the given name and bit width. The token name is also defined as a field name that refers to the entire token. Additional fields are defined using an optional `fields` clause following the token definition. Field names are give in a comma separated list, and each name is associated with a single bit or a range of bits within the token. Token bits are numbered in big-endian order starting at 0 up to the token width minus 1—i.e., bit number 0 is the right most (least significant) bit and bit numbers increase to the left. For example, in a 32 bit token, the first byte (8-bits) contains bits 24:31, the second byte contains bits (16:23), etc.

global_stmt	→ `pat` *pnames* `=` *pat_exp* `;`	pattern name declaration
pnames	→ `[` *pname_list* `]`	
	→ **name**	
pname_list	→ *pname_list* *pname*	
	→ *pname*	

pname	→ **name**	
	→ _	

A `pat` declaration associates mnemonic names with patterns that describe the binary encodings of instructions. A pattern is represented as a collection of conditions on token fields in disjunctive normal form—i.e., an OR-list of AND-lists of conditions on token fields. Patterns are constructed with pattern expressions, called *pat_exp* in the grammar.

pat_exp	→ *pat_field op* **int**	single condition
	→ *pat_field* `in [` *pint_list* `]`	multiple condition
	→ *pat_exp* `||` *pat_exp*	OR two patterns
	→ *pat_exp* `&&` *pat_exp*	AND two pattern
	→ *pat_exp* `$` *pat_exp*	concatenate patterns
	→ **name**	named pattern
pat_field	→ **name** `?` **name** `(` *int_list* `)`	attributed field name
	→ **name**	field name
int_list	→ *int_list* `,` **int**	
	→ **int**	
pint_list	→ *pint_list pint*	
	→ *pint*	
pint	→ **int** `..` **int** `by` **int**	range of integers with step
	→ **int** `..` **int**	range of integers with step 1
	→ **int**	single integer

A single condition compares a token field to a single integer value using one of the comparison operators `<`, `<=`, `==`, `!=`, `>=`, or `>`. The `in` expression generates multiple comparisons OR'ed together that test if the token field is equal to any of the listed integers. A range of integers can be abbreviated using an ellipsis (`..`) with an optional step argument. For example, the pattern

expression `cond in [ 0x0 .. 0xf ]` is the same as OR'ing together 16 single comparisons, testing if the field `cond` is equal to the values 0 through 15.

Fields in a pattern expression can be referenced as field names defined in a previous `token` declaration, or as a sub-field of a named token field. The attribute operators `?bit` and `?bits` can be used in pattern expressions to select a single bit or a range of bits from the named token field. These attributes are describe more on page 26. When used in a pattern expression the `?bit` attribute operator can only be called with a literal integer operand.

The `||` operator concatenates the OR-lists of its two operands to generate a new pattern in disjunctive normal form. && returns the cross product of its operands, concatenating every AND-list from the left-hand side with ever AND-list from the right-hand side. The concatenation operator ($) behaves like the && operator, but $ also adds an offset to all the tokens in its right-hand side operand so they follow the left-hand side tokens in a matched token stream.

If a `pat` declaration is used to define multiple pattern names, then each pattern name is mapped to one element of the OR-list described by the pattern expression. The number of pattern names being defined must be the same as the number of elements in the OR-list described by the pattern expression, or it is an error. The wildcard pattern name (_) can be used to skip an element of the OR-list without associating it to a name.

To allow multiple pattern names to be defined with more than one OR'ed AND-list, previously declared pattern names are not expanded until after the new pattern names, listed in the current `pat` declaration, have been defined. Once defined, previously declared pattern names in the pattern expression are expanded, and the patterns associated with each new pattern name are re-normalized into disjunctive normal form.

global_stmt	→ `sem` *snames scope where_opt* `;`	instruction semantic declaration
snames	→ `[` *name_list* `]`	
	→ **name**	
name_list	→ *name_list* **name**	
	→ **name**	
scope	→ `{` *stmt_list* `}`	
where_opt	→ `where` *where_bind_list*	
	→	
where_bind_list	→ *where_bind_list* `,` *where_bind*	
	→ *where_bind*	
where_bind	→ **name** `=` *exp*	
	→ **name** `in [` *wexp_list* `]`	
wexp_list	→ *wexp_list wexp*	
	→ *wexp*	
wexp	→ *w_op*	
	→ **name**	
	→ *aexp*	

A `sem` declaration associates a list of previously declared pattern names with semantic code. These are the pattern names that correspond to instructions in a target ISA. Other pattern names, those without semantics, just describe miscellaneous conditions on token streams that do not nec-

essarily correspond to instructions. Note that token field names can be used in semantic code to reference the value of bit fields within the instruction, with the same restrictions as in pattern cases of a Facile switch statement (described on page 17).

Multiple pattern names (instructions) can be associated with semantic code using a single `sem` declaration. Each pattern name is bound to a different copy of the semantic code. An optional `where` clause parameterizes the semantic code, and these parameters can be bound to different expressions for each instruction being defined. The list of expressions associated with each parameter must have the same length as the list of instruction names being defined.

Although Facile usually forbids function pointers or pointers to operators, expressions in the list of values for parameters in a `where` clause are an exception. Elements in these lists can be Facile operators (*w_op* can be one of `!`, `%`, `&`, `&&`, `*`, `+`, `-`, `/`, `<<`, `>>`, `^`, `^^`, `|`, `||`, `~`, `<`, `<=`, `==`, `!=`, `>=`, `>`), function names, or an atomic expressions (*aexp* is defined on page 20). More complex expressions can be listed, but must be enclosed in parentheses, as per the definition of *aexp*.

Facile Types

Facile is a strongly typed language with a type system inspired by the programming language ML. As in ML, functions can be defined with incomplete—i.e., polymorphic—types. Polymorphic functions can be instantiated with different concrete types at different call sites. For example, the identity function (takes a single argument and returns it) has type $(\texttt{float}) \rightarrow \texttt{float}$ [1] when called with a floating-point argument, and type $(\texttt{ulong}) \rightarrow \texttt{ulong}$ when applied to a 32-bit

unsigned integer. Unlike ML, Facile also allows overloading. An overloaded function name can be bound to two or more function bodies, so long as the overloaded versions can be distinguished by the types of their parameters or return value.

Type expressions can optionally be associated with any semantic expression, variable declaration, or function parameter or return value. Normally no types need to be specified, and the Facile compiler infers complete type information from clues in the code. For example, literal constant `5` automatically has type `ulong`, and in the variable binding `val x = 5` the variable `x` also has type `ulong`. Occasionally though, Facile's type inference algorithm needs some help determining types and disambiguating overloaded functions and operators. When disambiguation is needed, types are specified by following an expression, declared variable name, or function parameter with a colon (`:`) and the appropriate type expression.

bind_stmt → `type` *type_param_list_opt* **name** *length_param_list_opt* `=` *type* `;`

type_param_list_opt → *type_param_list_opt* *type_param*
→

length_param_list_opt → `[` *lenght_param_list* `]`
→

length_param_list → *length_param_list* `,` *type_param*
→ *type_param*

type_param → **name**
→ _

---

1. Facile has no explicit function types. Where function types are need in this reference they are written as (*arg-type-list*) → *return-type* .

A `type` declaration defines a new type name. The type name can optionally be parameterized by one or more type parameters that stand for any type, or by length parameters that can get instantiated with literal integer values. An example of a `type` declaration without any parameters is `type cwp_t = unsigned[5]`, where the type `cwp_t` is defined to be the 5-bit unsigned integer type. An example of a parametrized type name declaration is `type T pair = (T,T)`, so the type `ulong pair` would be the same as type `(ulong,ulong)`.

Length parameters are useful in types built from arrays (to specify the array length) or from integer or floating point types (to specify the width). For example, to declare a type name for arrays of unsigned integers, the following declaration could be used:

```
type uarray[dim,width] = unsigned[width] array[dim];
```

The wildcard name (_) can be used in place of a type or length parameter that is not actually used in the type expression. For example, `type T my_queue[_] = T queue` defines `my_queue` so that it takes a length argument even though the length argument is not needed with Facile's built-in `queue` datatype.

type → *type_arg_list* **name** *type_lengths_opt*
→ *type_arg_list* `array` *type_lengths_opt*
→ *type_arg_list* `queue`
→ *atype*

atype	→ **name** *type_lengths_opt*	
	→ `struct {` *field_type_list_opt ellipsis_opt* `}`	structure type
	→ `(` *type* `,` *type_list* `)`	tuple type
	→ `(` *type* `)`	
	→ _	wildcard (polymorphic)
type_arg_list	→ *type_arg_list atype*	
	→ *atype*	
type_lengths_opt	→ `[` *type_length_list* `]`	
	→	
type_length_list	→ *type_length_list* `,` *type_length*	
	→ *type_length*	
type_length	→ **name**	
	→ **int**	
	→ _	

Several base types are predefined: The types `char`, `short`, `long`, and `llong` are all signed integer types with widths (in bits) 8, 16, 32, and 64 respectively. Types `bool`, `uchar`, `ushort`, `ulong`, and `ullong` are unsigned integers with widths (in bits) 1, 8, 16, 32, and 64 respectively. Types `float`, `double`, and `quad` are IEEE floating point types with widths (in bits) 32, 64, and 128 respectively. Other predefined type names are `void`, `string`, `stream` (token streams), `system` (the type of the system variable, described on page 21), `cc` (condition codes), and `elf` (ELF file descriptors[1]).

Parameterized type names are instantiated by listing the type arguments before the parameterized name and listing length arguments in square brackets after the type name. All type and length parameters of a parameterized type must be given values when the type is instantiated. Array

1. ELF file descriptors are defined in the system C header file <libelf.h>.

types are specified with the `array` reserved word, given one type argument (the element type) and one length argument (the array length). Multi-dimensional array types are written as arrays of arrays. Facile also supports a double-ended dynamic queue datatype, that grows and shrinks as needed at run-time. Queue types are specified using the `queue` reserved word preceded by one type argument specifying the queue element type. The queue type does not have any length arguments because a queue length can change at run-time.

There are two other predefined parameterized type names: The names `signed` and `unsigned` stand for signed and unsigned integers respectively, and each takes one length argument specifying the width (in bits) of the integer. Signed integers must have width 8, 16, 32, or 64. Unsigned integers can have any width from 1 to 64 inclusive.

The wildcard name (_) can be used to describe polymorphic types. The wildcard stands for an unknown type that will be instantiated later by type inference. For example, `unsigned[_]` is an unsigned type of any width.

atype	→ `struct {` *field_type_list_opt* *ellipsis_opt* `}`	structure type
	→ `(` *type* `,` *type_list* `)`	tuple type
field_type_list_opt	→ *field_type_list_opt* `,` *field_type*	
	→	
field_type	→ **name** `:` *type*	
type_list	→ *type_list* `,` *type*	
	→ *type*	

Array and queue types are examples of complex types. Other complex types can be constructed using Facile's tuple and structure type constructors. Tuple types are specified as comma separated lists of types in parentheses, with at least two elements in the list. Structures are specified using the `struct` reserved word followed by a list of field name/type pairs in curly brackets. Incomplete/polymorphic structure types are specified by including an ellipsis (`..`) at the end of the field list. Incomplete structure types are not the same as subtypes: Once an incomplete structure type is instantiated to be a complete type, the type is fixed and no subtyping is allowed.

Semantic Statements

Semantic statements include variable, function, and type name declarations, and a variety of control flow statements. Semantic declarations can be used in any scope—global or local—and define a name that can be used anywhere after the declaration statement within the current scope or in any subsequent sub-scope. Other statements are only allowed within a function body or in instruction semantic code. Any expression (described in the next section) can also be written as its own statement to ignore the expression's result value. Multiple statements can be grouped within curly brackets (`{}`), which delimit a new sub-scope. Variable names declared with a sub-scope or in any enclosing scope out to and including the global scope can be accessed.

global_stmt → *bind_stmt*

bind_stmt → `type` *type_arg_list_opt* **name** *length_arg_list_opt* `=` *type* `;`
→ *inline_opt* `fun` **name** `(` *arg_list_opt* `)` *type_restrict_opt scope*
→ *const_opt* `val` **name** *type_restrict_opt bind_exp_opt* `;`
→ `var` **name** *type_restrict_opt* `=` *exp* `;` reference variable

arg_list_opt → *arg_list* `,` *arg*
→

arg → `var` **name** *type_restrict_opt*
→ **name** *type_restrict_opt*

scope → `{` *stmt_list* `}`

stmt_list → *stmt_list* *stmt*
→ *stmt*

bind_exp_opt → `=` *exp*
→

inline_opt → `inline`
→

const_opt → `const`
→

type_restrict_opt → `:` *type*
→

Binding statements can appear in both the global scope and in local scopes, i.e., inside a function body or in the semantic code of a `sem` declaration. Type binding declarations are described on page 9 in the section on Facile types. The remaining binding statements define function and variable names. A `fun` declaration defines a function, a `val` declaration defines a variable, and a `var` declaration defines a reference to any lvalue[1]. Although functions and variable names must be declared before they are used, specifying types is optional if the type can be derived from the context in which the function or variable is used.

1. As in C, an lvalue is an expression that identifies a particular storage location. For example, an array lookup `A[5]` is an lvalue, but a computed value `x+1` is not.

A function binding includes a function name, a list of parameter names, and a function body. Function parameters can be passed by value or passed by reference. Reference parameters are preceded with the `var` reserved word. An optional type specification is allowed after each function parameter to specify the parameter's type, and between the parameter list and the function body to specify the function return type. If a function declaration is preceded by the `inline` reserved word, then all calls to the function are inlined. Regardless of the presence or absence of the `inline` flag, all calls to local functions (i.e., functions declared within a scope other than the global scope) are inlined.

Non-reference variables are defined using `val` declarations. Once declared, a variable can be used in subsequent statements of the same scope or sub-scopes. An optional initial value can be given in the declaration. A `val` declaration preceded by the `const` reserved word cannot be changed by subsequent assignments. Constant variables must have an initial value given in their declaration. Reference variables are defined using `var` declarations, and are interpreted as an alias to the value they are a reference for. A reference variable must be initialized with an lvalue expression.

stmt → *bind_stmt*
→ `if (` *exp* `)` *stmt*
→ `if (` *exp* `)` *stmt* `else` *stmt*
→ `while (` *exp* `)` *stmt*
→ `for (` **name** *type_restrict_opt* `in` *exp* `)` *scope*
→ `switch (` *exp* `) {` *case_list* `}`
→ `continue` *int_opt* `;`
→ `break` *int_opt* `;`
→ `return` *exp_opt* `;`
→ *exp* `;`
→ *scope*
→ `;`

int_opt → **int**
→

exp_opt → *exp*
→

Facile's control flow statements are `if`, `while`, `for`, `switch`, `continue`, `break`, and `return`. An `if` statement switches between two possible control flow paths based on the result of an expression with type `bool`. The `else` clause is optional. A `while` statements test a conditional expression of type `bool`, and loops until the condition evaluates to `false`. A `for` loop iterates through the elements of an array or queue. The variable named in a `for` loop is only defined within the associated scope, and it refers to the elements of the given array or queue, one element per iteration in ascending index order.

A `continue` statement causes the control flow to jump to the next iteration of an enclosing `while` or `for` loop. A `break` statement jumps to just after an enclosing loop. An optional integer literal can follow a `continue` or `break`, specifying which enclosing loop to continue or break from. A value of 1 specifies the innermost loop, 2 is the second innermost loop, etc. If omit-

ted, the default is to continue or break from the innermost loop. A `return` statement returns from the current function. If the function return type is anything other than `void`, then a return value must be given.

An expression can be written as a statement. This is useful for expressions with a `void` result type (e.g., assignment expressions and function calls with no return value). A statement can also be empty (just a `;` with no other code), or multiple statements grouped within curly brackets (`{}`). Every group of statements within curly brackets is a sub-scope. All variable, function, and type names defined in enclosing scopes can be used in a sub-scope provided they are not masked by other variable, function, or type names. Note that function and variable names can be overloaded, so a function name may not get masked by later bindings, unless the type of the new binding masks the type of bindings for the same function name in enclosing scopes.

stmt	→ `switch (` *exp* `) {` *case_list* `}`	
case_list	→ *case_list case_clause*	
	→ *case_clause*	
case_clause	→ `case` *match1* `:` *stmt_list*	normal case
	→ `pat` *pat_exp* `:` *stmt_list*	pattern case
	→ `default :` *stmt_list*	default case
match	→ *match* `:` *type*	type specification
	→ **name** `as` *match*	declare alias name
	→ `pat` *pat_exp*	pattern sub-case
	→ *match1*	

match1	→ **name**	declare variable or reference
	→ *int_match*	match an integer
	→ `(` *match* `,` *match_list* `)`	match a tuple
	→ `{` *match_field_list* *ellipsis_opt* `}`	match a record
	→ `(` *match* `)`	grouping
	→ `_`	wildcard
int_match	→ `-` **int**	negative signed integer
	→ `+` **int**	positive signed integer
	→ **int**	unsigned integer
match_list	→ *match_list* `,` *match*	
	→ *match*	
match_field_list	→ *match_field_list* `,` *match_field*	
	→ *match_field*	
match_field	→ `name =` *match*	
ellipsis_opt	→ `..`	
	→	

Facile `switch` statements are similar to case statement in ML, because they can extract individual elements of complex data structures and bind them to variable names, in addition to selecting between multiple cases based on data values. A Facile `switch` statement can also select between multiple patterns in a token stream. For example, instructions in an ISA could be decoded using a `switch` statement with pattern cases, and the Facile compiler internally transforms instructions defined with `pat` and `sem` declarations into a Facile `switch`.

Following the `switch` reserved word and the condition expression is a list of case clauses. Logically, a `switch` statement is evaluated by first evaluating its condition expression, then stepping through the list of case clauses in program order until one of the clauses matches the value of

the condition. Then the statements associated with the matching case clause are evaluated. The statements associated with a case clause do not fall through to the next case, as in C.

A case clause with the `case` reserved word is used to match data with types other than `stream` (the token stream type). These clauses can match simple signed and unsigned integer values, or deconstruct complex types to match the values of one or more fields. If variable names are given in a case clause's match expression and the case clause is matched when evaluating the switch statement, then the named variables are bound to their corresponding component of the condition expression. These variable bindings are only valid within the scope of the case clause. If the condition expression is an lvalue, then variable names are bound as reference variables, otherwise the variable names are bound to a copy of the selected data. The wildcard name (_) can be used in place of a variable name to match any value without binding it to a name.

A case clause with the `pat` reserved word matches data of type `stream`. The pattern expressions used here are the same as the pattern expressions used in `pat` declarations, described on page 4. The statements associated with a pattern case can use token field names to access bit fields within the matched token stream. Token field names are defined for the case clauses statements if the token is in the matched pattern and the field name uniquely identifies a sequence of bits in the token stream. Field names for token not in a matched pattern and field names that cannot be uniquely determined are not defined. Patterns in a token stream can also be matched as part of the match expression in a case clause with the `case` reserved word, but no field names will be defined.

A case clause with the `default` reserved word is the same as a case clause with the `case` reserved word and a wildcard (_) match expression (i.e., `case _ :` *stmt-list*). Case clauses following the default case are never matched. In general, case clauses are tested in the same order as they appear in the switch statement. Earlier case clauses may mask later case clauses if every value that would match in a later case is matched by earlier cases.

Semantic Expressions

Adding two values, calling a function, and assigning to a variable are all examples of expressions. All expressions that have type other than `void` produce one result value. `void` typed expressions have no result value.

exp	→ *exp* `:` *type*	type restriction
	→ *exp* *op2* *exp*	binary operator
	→ *aexp1*	
aexp1	→ *op1* *aexp1*	unary operator
	→ **name** `(` *exp_list_opt* `)`	function call
	→ *aexp1* `?` **name** *arg_list_opt*	attribute
	→ *aexp*	
aexp	→ **name**	variable name
	→ **int**	literal unsigned integer
	→ **float**	literal 64-bit float
	→ **string**	literal string
	→ `array` *length_arg_opt* `{` *exp_list* `}`	array
	→ `queue` *length_arg_opt* `{` *exp_list_opt* `}`	queue
	→ `struct {` *struct_field_list* `}`	structure
	→ `(` *exp* `,` *exp_list* `)`	tuple
	→ *aexp* `[` *exp* `]`	array/queue lookup
	→ *aexp* `.` **name**	record/tuple field selection
	→ `(` *exp* `)`	grouping

arg_list_opt → `(` *exp_list_opt* `)`
→

exp_list_opt → *exp_list*
→

exp_list → *exp_list* `,` *exp*
→ *exp*

length_arg_opt → `(` *exp* `)`
→

struct_field_list → *struct_field_list* `,` *struct_field*
→ *struct_field*

struct_field → **name** `=` *exp*

At the leaves of an expression tree are variable names and literal integer, floating-point, and string values. Note that literal integers are all unsigned and literal floating-point values are all 64-bits wide (type `double`). Literal values can be cast or converted to other types as needed. The names `true` and `false` are predefined to be the constant 1-bit unsigned integers 1 and 0 respectively. The name `system` is predefined as a value of type `system`, and provides access to various system related attributes, including getting/setting memory values and accessing the raw ELF file descriptor for a target executable.

An un-named array or queue value is constructed using the `array` or `queue` reserved word respectively, followed by an optional length argument and an initializer list. When constructing an array, the array length (if omitted) is inferred from the length of the initializer list. If an array length is given and the initializer list has exactly one element, then the array is initialized with a copy of the initializer value in every array element. Otherwise the array length and the initializer

list length must be the same. When constructing a queue, the length argument is only a hint to the compiler, and a queue's initial length is determined entirely from the length of its initializer list. The length argument in a queue constructor should be the expected maximum number of elements that will be stored in the queue, so Facile's compiler can generate efficient queue allocation code. Both array and queue elements are accessed by following an expression that evaluates to an array or queue with an index enclosed in square brackets. In array lookup expressions, the index value is an unsigned integer. In queue lookup expressions, the index value is a signed integer, and negative values index backward from the back end of the queue (e.g., `Q[-1]` is the last element in queue `Q`).

Structures are constructed with the `struct` reserved word followed by a list of field name/value pairs enclosed in curly brackets (`{}`). Unlike structure types and structure matching expressions in `switch` statements, structure values must be fully defined (i.e., no ellipsis). Tuple values are constructed by enclosing a list of two or more expressions in parentheses. Structure fields are accessed by following an expression that evaluates to a structure with a period (`.`) and a field name. Tuples fields are access in the same way but an unsigned integer from 1 to the tuple length is used, since tuples have no field names (e.g., `(10,20).1` evaluates to `10`).

Facile has several infix binary and prefix unary operators. Unary operators are `!` (boolean not), `~` (bit inversion), `-` (negation), and + (unsigned to signed type casting). The `!` operator can only be applied to values of type `bool` and returns `bool` values. The `~` operator can be applied to any unsigned integer and returns a value of the same type. The unary `-` operator can be applied to

any signed integer, unsigned integer, or floating-point value and returns a value with the same width. Negating an unsigned integer produces a signed integer of the same width. The unary + operator can only be applied to unsigned integers and casts its operand to a signed value with the same bit-width. Facile also pre-defines the function `sqrt` (called as a function, not as an operator) that takes one floating point argument and returns its square root.

The following table lists all of Facile's boolean operators, and their (possibly overloaded) types. These operators are listed in order of increasing precedence, with operators listed on the same line having the same precedence. All binary operators are left associative.

Operators	**Type(s)**
`=`	(α, α) → `void`
`&& || ^^`	(`bool`, `bool`) → `bool`
`< <= == != >= >`	(`unsigned[`α`]`, `unsigned[`α`]`) → `bool` (`signed[`α`]`, `signed[`α`]`) → `bool` (*fp*`[`α`]`, *fp*`[`α`]`) → `bool`[a]
`+ -`	(`stream`, `unsigned[`α`]`) → `stream` (`unsigned[`α`]`, `unsigned[`α`]`) → `unsigned[`α`]` (`signed[`α`]`, `signed[`α`]`) → `signed[`α`]` (*fp*`[`α`]`, *fp*`[`α`]`) → *fp*`[`α`]`
`* / %`	(`unsigned[`α`]`, `unsigned[`α`]`) → `unsigned[`α`]` (`signed[`α`]`, `signed[`α`]`) → `signed[`α`]` (*fp*`[`α`]`, *fp*`[`α`]`) → *fp*`[`α`]` (`*` and `/` only)
`& | ^`	(`unsigned[`α`]`, `unsigned[`α`]`) → `unsigned[`α`]`
`<< >>`	(`unsigned[`α`]`, `unsigned[_]`) → `unsigned[`α`]` (`signed[`α`]`, `unsigned[_]`) → `signed[`α`]` (`>>` only)

a. The name *fp* is used to describe a floating-point types with a width parameter. A special name is needed here, since no such name is predefined in Facile.

In addition to binary and unary operators, Facile also has attributes and attribute calls. Attributes follow an expression and are written as `?` and the attribute name. Attribute calls also have a list of arguments in parentheses. Attributes perform special operations that either support Facile's built-in data types (e.g., queues and condition codes) or require non-standard syntax (e.g., type casting and bit selection). Facile makes no distinction between attributes and attribute calls with no arguments. For example `PC?exec` is the same as `PC?exec()`. Below is a comprehensive list of all the attributes and attribute calls recognized by Facile:

- *exp*`?length` returns the length of an array or queue.

- *exp*`?width` returns the bit-width of a signed, unsigned, or floating-point value.

- *exp*`?start_pc` applies to expressions of type `system` (normally only the predefined variable `system`) and returns the start address of the target executable.

- *exp*`?start_sp` applies to expressions of type `system` (normally only the predefined variable `system`) and returns the initial stack pointer for the target executable.

- *exp*`?elf` applies to expressions of type `system` (normally only the predefined variable `system`) and returns the ELF file descriptor for the target executable.

- *exp*`?exec()` decodes the first instruction in the given token stream and evaluates its semantic code. Instruction patterns and semantics are declared using `pat` and `sem` declarations. All `sem` declarations must occur before the first call to `?exec` in a Facile source file.

- *exp*`?addr` returns the target byte address of the beginning of a token stream.

- *exp*`?cast(`*type*`)` changes the type of a signed integer, unsigned integer, or floating-point value to a signed integer, unsigned integer, or floating point type with the same width. The bits of a value are not changed, only its type.

- *exp*`?cvt(`*type*`)` converts a value of any signed integer, unsigned integer, or floating-point type to a value of any other signed integer, unsigned integer, or floating point type. Note that the bit representation may be changed, although the represented value is preserved up to the resolution of the new representation type.

- *exp*`?ext(`**int**`)` extends a signed or unsigned integer to the bit-width specified in the attribute argument by padding the most significant bits with 0s. The given width must be equal to or greater than the width of the integer being extended. The extended result is unsigned.

- *exp*`?sext(`**int**`)` sign extends a signed or unsigned integer to the bit-width specified in the attribute argument. The sign of the value being extended is inferred from its most significant bit (as in a 2s complement representation), even if the value has an unsigned type. The given

width must be equal to or greater than the width of the integer being extended. The extended result is unsigned.

- *exp*`?memory(`**int**`,`*exp*`)` applies to expressions of type `system` (normally only the pre-defined variable `system`) and provides access to a target executable's memory. The first attribute argument is a literal integer specifying the access width in bytes[1], and the second argument is the target virtual address. Note that the result of a `?memory` attribute call is an lvalue, so memory values can be both read and written.[2]

- `(`*exp op2 exp*`)?cc(`*exp*`)` computes the condition codes associated with a boolean operator expression. The normal expression value is returned, but condition codes are stored into the attribute call argument as a side effect. The attribute call argument must be an lvalue of type `cc`. Facile can compute condition codes for the floating-point subtraction operator (`-`) and the following integer operators: `+`, `-`, `&&`, `||`, `^^`, `&`, `|`, and `^`.

- *exp*`?bit(`*exp*`)` selects one bit from any signed, unsigned, floating-point, or condition code value. Which bit is selected is specified by the attribute call argument.

- *exp*`?bits(`**int**`,`**int**`)` and *exp*`?bits(`**int**`)` select one or more bits from any signed, unsigned, floating-point, or condition code value. The first version of this attribute call is called with the

1. FastSim's current host processor is a SPARC, so unaligned memory accesses produce a bus error signal (SIGBUS).
2. Some target memory is not mapped or is mapped read-only, as it would be in a real execution outside the simulator. Attempting any access to an address that is not mapped, or assigning to read-only memory produce a segmentation violation signal (SIGSEG).

index of the first and last bits to select. The second version selects a given number of bits starting from bit 0, the least significant bit. The attribute arguments are integer literals because the number of bits selected must be known by the compiler to determine the result type. Results are unsigned.

- *exp*`?push_back(`*exp*`)` and *exp*`?push_back()` push one new element onto the back end of a queue. If an attribute argument is given, then it is used to initialize the new queue element. This attribute call has no return value (return type `void`).

- *exp*`?push_front(`*exp*`)` and *exp*`?push_front()` push one new element onto the front end of a queue. If an attribute argument is given, then it is used to initialize the new queue element. This attribute call has no return value (return type `void`).

- *exp*`?pop_back(`*exp*`)` and *exp*`?pop_back()` pop element(s) off the back end of a queue. The first version takes an unsigned integer as its attribute call argument, and pops the given number of entries off the queue. This version has no return value (return type `void`). The second version pops one element and returns it. Attempting to pop elements from an empty queue produces a run-time error.

- *exp*`?pop_front(`*exp*`)` and *exp*`?pop_front()` pop element(s) off the front end of a queue. The first version takes an unsigned integer as its attribute call argument, and pops the given number of entries off the queue. This version has no return value (return type `void`).

The second version pops one element and returns it. Attempting to pop elements from an empty queue produces a run-time error.

- *exp*`?clear()` removes all elements from a queue. No value is returned (return type `void`).

- *exp*`?static` causes a fast-forwarding simulator to make the attributed expression result value run-time static. If the attributed value is an lvalue, then the `?static` attribute also makes the lvalue storage location run-time static. Note that, if a value is already static or run-time static, or if the simulator is not compiled with the fast-forwarding optimization, then this attribute is a no-op.

Facile pre-defines the function `assert`. `assert` takes a single argument of type bool. If the argument evaluates to false at run-time, then the simulator exits with an assertion failed error.

External Code

Facile code can access variables and call functions written in other languages, or just linked from separate object files. Conversely, external code can access facile variables and call facile functions. A Facile file's external interface is specified using external variable, function, and type declarations.

bind_stmt	→ `extern` **name** `:` *type* `;`	external variable
	→ `extern` **name** `(` *arg_type_list_opt* `)` `:` *type* `;`	external function
	→ `extern type` **name** *type_assign_opt* `;`	external type

arg_type_list_opt → *arg_type_list* `,` *targ*
→

targ → `var` *type*
→ *type*

type_assign_opt → `=` *type*
→

Variable and function names that either refer to symbols in another file or provide external access to values and code in a Facile file must be declared with an external variable or function declaration respectively. These declarations define the type of an external variable or the types of an external function's parameters and return value. To export a global variable or function defined in a Facile file, first declare the name with one of these external declaration, then bind the name with a `var` or `fun` declaration. Note that only global variables and functions can be exported. External function names cannot be overloaded or polymorphic.

External type declarations either provide a name for an external pointer type that can be stored in Facile variables but cannot be not dereferenced, or a Facile type that may be used by external code. To declare a type name for an external pointer type, declare an extern type name but omit the optional type assignment. Values of this type can then be stored in Facile variables, but they cannot be dereferenced and are always treated as dynamic values when fast-forwarding. To export a Facile type, declare an external type name and assign it to a Facile type. Exported Facile types are put in a C header file generated by the Facile compiler, and can be `#include`'ed into other C source files.

Simulator Layout

A Facile simulator must have a certain structure to work in the FastSim run-time environment. Primarily, a simulator must define a global variable called `init` and a global function called `main`. The `main` function is called repeatedly by the FastSim run-time environment. Each time `main` returns, FastSim calls it again. `main` must have one or more parameters, with any types except named external pointer types (since they cannot be memoized). The value of `main`'s arguments are used to index the memoization cache when fast-forwarding.

The argument values for each call to `main` are stored in `init`. Typically each call to `main` will update the `init` variable to set up arguments for the next call to `main`. If `main` has one argument, then `init` has the same type as `main`'s one argument. If `main` has multiple arguments, then `init` is a tuple, where each tuple field corresponds to one argument.

Call-by-reference parameters (declared with the `var` reserved word) can be used in the declaration of `main`. A reference parameter refers to the corresponding tuple field in the `init` variable. Hence the value in `init` can be modified by either assigning directly to `init` or by assigning to one of `main`'s reference parameters.