CEC Abstract Syntax Tree

Stephen A. Edwards
Columbia University
sedwards@cs.columbia.edu

Contents

1 The ASTNode Class

All AST nodes are derived from this class; the `Visitor` class takes an `ASTNode` as an argument.

2a ⟨*ASTNode class* 2a⟩≡ (27)

```
abstract "ASTNode : Node


  virtual Status welcome(Visitor&) = 0;
"
```

2 Symbols and Types

Symbols represent names in the Esterel source code, such as those for signals, functions, variables, and other modules.

2b ⟨*Symbol classes* 2b⟩≡ (27)

⟨*Symbol* 2c⟩
⟨*ModuleSymbol* 2d⟩
⟨*SignalSymbol* 3a⟩
⟨*BuiltinSignalSymbol* 3b⟩
⟨*TrapSymbol* 3c⟩
⟨*VariableSymbol* 3d⟩
⟨*ConstantSymbol* 4a⟩
⟨*BuiltinConstantSymbol* 4b⟩
⟨*TypeSymbol* 4c⟩
⟨*BuiltinTypeSymbol* 4d⟩
⟨*FunctionSymbol* 4e⟩
⟨*BuiltinFunctionSymbol* 4f⟩
⟨*ProcedureSymbol* 5a⟩
⟨*TaskSymbol* 5b⟩

2c ⟨*Symbol* 2c⟩≡ (2b)

```
abstract "Symbol : ASTNode
  string name;

  Symbol(string s) : name(s) {}"
```

Symbol representing a module.

2d ⟨*ModuleSymbol* 2d⟩≡ (2b)

```
class "ModuleSymbol : Symbol
  Module *module;

  ModuleSymbol(string s) : Symbol(s), module(0) {}"
```

SignalSymbol represents a signal. Pure signals have a NULL type. The `presence` variable is a boolean variable if the signal is not a sensor. The `value` variable is the variable for its value, if the signal is not pure. The `combine` field points to the "combine" function (e.g., combine integer with +) if there is one, and NULL otherwise.

3a ⟨*SignalSymbol* 3a⟩≡ (2b)

```
class "SignalSymbol : Symbol
  TypeSymbol *type;
  string direction; // input, output, inputoutput, sensor, return, local
  FunctionSymbol *combine; // combining function, if any
  Expression *initializer;
  VariableSymbol *presence;
  VariableSymbol *value;

  SignalSymbol(string n, TypeSymbol *t, string d, FunctionSymbol *f,
               Expression *e, VariableSymbol *p, VariableSymbol *v)
    : Symbol(n), type(t), direction(d), combine(f), initializer(e),
      presence(p), value(v) {}"
```

For the built-in signal "tick."

3b ⟨*BuiltinSignalSymbol* 3b⟩≡ (2b)

```
class "BuiltinSignalSymbol : SignalSymbol

  BuiltinSignalSymbol(string n, TypeSymbol *t, string d, FunctionSymbol *f,
                      VariableSymbol *p, VariableSymbol *v)
    : SignalSymbol(n, t, d, f, NULL, p, v) {}"
```

Symbol representing a trap. Pure traps have a NULL type. The `presence` field points to a boolean variable. The `value` field points to a variable if the trap has a value, 0 otherwise. The `code` field is used during dismantling and stores the completion code associated with this trap.

3c ⟨*TrapSymbol* 3c⟩≡ (2b)

```
class "TrapSymbol : Symbol
  TypeSymbol *type;
  Expression *initializer;
  int code;
  VariableSymbol *presence;
  VariableSymbol *value;

  TrapSymbol(string s, TypeSymbol *t, Expression *e, VariableSymbol *p,
             VariableSymbol *v)
    : Symbol(s), type(t), initializer(e), code(0), presence(p), value(v) {}"
```

3d ⟨*VariableSymbol* 3d⟩≡ (2b)

```
class "VariableSymbol : Symbol
  TypeSymbol *type;
  Expression *initializer;

  VariableSymbol(string n, TypeSymbol *t, Expression *e)
    : Symbol(n), type(t), initializer(e) {}"
```

A constant symbol has a name, type, and an optional initializing expression whose type must match that of the constant.

4a ⟨*ConstantSymbol* 4a⟩≡ (2b)
```
class "ConstantSymbol : VariableSymbol

  ConstantSymbol(string n, TypeSymbol *t, Expression *i)
     : VariableSymbol(n, t, i) {}"
```

4b ⟨*BuiltinConstantSymbol* 4b⟩≡ (2b)
```
class "BuiltinConstantSymbol : ConstantSymbol

  BuiltinConstantSymbol(string n, TypeSymbol *t, Expression *i)
     : ConstantSymbol(n, t, i) {}"
```

2.1 Type Symbols

Esterel's type system provides a way to import types from a host language. A `TypeSymbol` is just a name, while the function and procedure types are for representing functions (return a value) and procedures (do not return a value, but have pass-by-reference parameters).

4c ⟨*TypeSymbol* 4c⟩≡ (2b)
```
class "TypeSymbol : Symbol

   TypeSymbol(string s) : Symbol(s) {}"
```

A `BuiltinTypeSymbol` represents one of the five built-in types: boolean, integer, float, double, and string.

4d ⟨*BuiltinTypeSymbol* 4d⟩≡ (2b)
```
class "BuiltinTypeSymbol : TypeSymbol

   BuiltinTypeSymbol(string s) : TypeSymbol(s) {}"
```

A imported function, e.g., "function foo(integer) : boolean;"

4e ⟨*FunctionSymbol* 4e⟩≡ (2b)
```
class "FunctionSymbol : TypeSymbol
  vector<TypeSymbol*> arguments;
  TypeSymbol *result;

  FunctionSymbol(string s) : TypeSymbol(s), result(NULL) {}"
```

BuiltinFunctionSymbols are used in "combine" declarations or module renamings.

4f ⟨*BuiltinFunctionSymbol* 4f⟩≡ (2b)
```
class "BuiltinFunctionSymbol : FunctionSymbol

  BuiltinFunctionSymbol(string s) : FunctionSymbol(s) {}"
```

An imported procedure or task, e.g., “procedure bar(integer)(boolean)”

5a ⟨*ProcedureSymbol* 5a⟩≡ (2b)

```
class "ProcedureSymbol : TypeSymbol
  vector<TypeSymbol*> reference_arguments;
  vector<TypeSymbol*> value_arguments;

  ProcedureSymbol(string s) : TypeSymbol(s) {}"
```

5b ⟨*TaskSymbol* 5b⟩≡ (2b)

```
class "TaskSymbol : ProcedureSymbol

  TaskSymbol(string s) : ProcedureSymbol(s) {}"
```

3 Symbol Tables

A symbol table is basically a map from names (strings) to `Symbol`s.

6 ⟨*SymbolTable* 6⟩≡ (27)

```
class "SymbolTable : ASTNode
  SymbolTable *parent;
  typedef map<string, Symbol*> stmap;
  stmap symbols;

  SymbolTable() : parent(NULL) {}

  class iterator {
    stmap::iterator i;
  public:
    iterator(stmap::iterator ii) : i(ii) {}
    void operator ++(int) { i++; } // int argument denotes postfix
    void operator ++() { ++i; } // int argument denotes postfix
    bool operator !=(const iterator &ii) { return i != ii.i; }
    Symbol *operator *() { return (*i).second; }
  };

  iterator begin() { return iterator(symbols.begin()); }
  iterator end() { return iterator(symbols.end()); }

  bool local_contains(const string) const;
  bool contains(const string) const;
  void enter(Symbol *);
  Symbol* get(const string);
" "
bool SymbolTable::local_contains(const string s) const {
  return symbols.find(s) != symbols.end();
}

bool SymbolTable::contains(const string s) const {
  for ( const SymbolTable *st = this ; st ; st = st->parent )
    if (st->symbols.find(s) != st->symbols.end()) return true;
  return false;
}

void SymbolTable::enter(Symbol *sym) {
  assert(sym);
  assert(symbols.find(sym->name) == symbols.end());
  symbols.insert( std::make_pair(sym->name, sym) );
}

Symbol* SymbolTable::get(const string s) {
  map<string, Symbol*>::const_iterator i;
  for ( SymbolTable *st = this; st ; st = st->parent ) {
    i = st->symbols.find(s);
```

```
    if (i != st->symbols.end()) {
      assert((*i).second);
      assert((*i).second->name == s);
      return (*i).second;
    }
  }
  assert(0); // get should not be called unless contains returned true
}
"
```

FIXME: The `local_contains` method indicates whether a symbol with the given name is contained in this particular table table. The `contains` method also searches in containing scopes.

The `enter` method adds a symbol to the table. It assumes the table does not already contain a symbol with the same name.

The `get` method returns the symbol with the given name. It assumes the symbol is present in the table.

4 Expressions

7a ⟨*Expression classes* 7a⟩≡ (27)

```
⟨Expression 7b⟩

⟨Literal 8a⟩
⟨LoadVariableExpression 8b⟩
⟨LoadSignalExpression 8c⟩
⟨LoadSignalValueExpression 8d⟩
⟨LoadTrapExpression 8e⟩
⟨LoadTrapValueExpression 9a⟩

⟨UnaryOp 9b⟩
⟨BinaryOp 9c⟩
⟨FunctionCall 9d⟩
⟨Delay 10a⟩

⟨CheckCounter 10b⟩
```

Every `Expression` has a type.

7b ⟨*Expression* 7b⟩≡ (7a)

```
abstract "Expression : ASTNode
  TypeSymbol *type;

  Expression(TypeSymbol *t) : type(t) {}"
```

4.1 Literal

A literal is an integer, float, double, or string literal value. All are stored as strings to maintain precision.

8a ⟨*Literal* 8a⟩≡ (7a)

```
class "Literal : Expression
  string value;

  Literal(string v, TypeSymbol *t) : Expression(t), value(v) {}"
```

4.2 Variables, Signals, and Traps

`LoadVariableExpression` is a reference to a variable or constant. It is also used to reference the built-in boolean constants `true` and `false`.

8b ⟨*LoadVariableExpression* 8b⟩≡ (7a)

```
class "LoadVariableExpression : Expression
  VariableSymbol *variable;

  LoadVariableExpression(VariableSymbol *v)
    : Expression(v->type), variable(v) {}"
```

`LoadSignalExpression` returns the presence/absence of a signal. Used by present, etc. Its `type` is always the built-in boolean

8c ⟨*LoadSignalExpression* 8c⟩≡ (7a)

```
class "LoadSignalExpression : Expression
  SignalSymbol *signal;

  LoadSignalExpression(SignalSymbol *s)
    : Expression(s->type), signal(s) {}"
```

`LoadSignalValueExpression` returns the value of a valued signal, i.e., the `?` operator.

8d ⟨*LoadSignalValueExpression* 8d⟩≡ (7a)

```
class "LoadSignalValueExpression : LoadSignalExpression

  LoadSignalValueExpression(SignalSymbol *s) : LoadSignalExpression(s) {}"
```

`LoadTrapExpression` returns the presence/absence of a trap. Used by trap .. handle. `type` is always the built-in boolean.

8e ⟨*LoadTrapExpression* 8e⟩≡ (7a)

```
class "LoadTrapExpression : Expression
  TrapSymbol *trap;

  LoadTrapExpression(TrapSymbol *t)
   : Expression(t->type), trap(t) {}"
```

`LoadTrapValueExpression` returns the value of a trap, i.e., the `??` operator.

9a ⟨*LoadTrapValueExpression* 9a⟩≡ (7a)

```
class "LoadTrapValueExpression : LoadTrapExpression

  LoadTrapValueExpression(TrapSymbol *s) : LoadTrapExpression(s) {}"
```

4.3 Operators

Esterel has the usual unary and binary operators. The `op` field represents the actual type of the operator. Its value is the Esterel syntax for the operator, e.g., `<>` for not equal.

9b ⟨*UnaryOp* 9b⟩≡ (7a)

```
class "UnaryOp : Expression
  string op;
  Expression *source;

  UnaryOp(TypeSymbol *t, string s, Expression *e)
    : Expression(t), op(s), source(e) {}"
```

9c ⟨*BinaryOp* 9c⟩≡ (7a)

```
class "BinaryOp : Expression
  string op;
  Expression *source1;
  Expression *source2;

  BinaryOp(TypeSymbol *t, string s, Expression *e1, Expression *e2)
    : Expression(t), op(s), source1(e1), source2(e2) {}"
```

4.4 Function Call

This is a function call in an expression. Callee must be defined.

9d ⟨*FunctionCall* 9d⟩≡ (7a)

```
class "FunctionCall : Expression
  FunctionSymbol *callee;
  vector<Expression*> arguments;

  FunctionCall(FunctionSymbol *s)
    : Expression(s->result), callee(s) {}"
```

4.5 Delay

This is a delay, e.g., the argument of await 5 SECOND. The predicate is a pure signal expression that returns the built-in boolean. The count may be undefined. `is_immediate` is true for expressions such as "await immediate A." The `counter` variable is used when the delay is a counted one, and is 0 for immediate delays.

10a ⟨*Delay* 10a⟩≡ (7a)

```
class "Delay : Expression
  Expression *predicate;
  Expression *count;
  VariableSymbol *counter;
  bool is_immediate;

  Delay(TypeSymbol *t, Expression *e1, Expression *e2, VariableSymbol *v,
        bool i) : Expression(t), predicate(e1), count(e2), counter(v),
                  is_immediate(i) {}"
```

4.6 CheckCounter

This checks the counter alarm

10b ⟨*CheckCounter* 10b⟩≡ (7a)

```
class "CheckCounter : Expression
  VariableSymbol *counter;

  CheckCounter(TypeSymbol *t, VariableSymbol *c): Expression(t), counter(c) {}
"
```

5 Modules

10c ⟨*Module classes* 10c⟩≡ (27)

⟨*Module* 11⟩
⟨*InputRelation classes* 12a⟩
⟨*Modules* 12b⟩

Esterel places signals, types, variables/constants, functions, procedures, tasks, and traps in separate namespaces, so each has its own symbol table here except traps, which are only in scopes.

The `variables` symbol table holds `VariableSymbols` representing signal presence and value, trap status and values, counters, state variables, etc., all generated during the disamantling process.

The `max_code` field holds the highest completion code used anywhere in the module.

11 ⟨*Module* 11⟩≡ (10c)

```
class "Module : ASTNode
  ModuleSymbol *symbol;
  SymbolTable *types;
  SymbolTable *constants;
  SymbolTable *functions;
  SymbolTable *procedures;
  SymbolTable *tasks;
  SymbolTable *signals;
  SymbolTable *variables;
  vector<InputRelation*> relations;
  ASTNode *body;
  int max_code;

  Module() : max_code(0) {}
  Module(ModuleSymbol *);
  ~Module();
" "
Module::Module(ModuleSymbol *s) : symbol(s), body(NULL) {
  signals = new SymbolTable();
  constants = new SymbolTable();
  types = new SymbolTable();
  functions = new SymbolTable();
  procedures = new SymbolTable();
  tasks = new SymbolTable();
  variables = new SymbolTable();
}

Module::~Module() {
   delete signals;
   delete types;
   delete constants;
   delete functions;
   delete procedures;
   delete tasks;
   delete body;
   delete variables;
}"
```

Relations are constraints (either exclusion or implication) among two or more input signals.

12a ⟨*InputRelation classes* 12a⟩≡ (10c)

```
abstract "InputRelation : ASTNode"

class "Exclusion : InputRelation
  vector<SignalSymbol *> signals;"

class "Implication : InputRelation
  SignalSymbol *predicate;
  SignalSymbol *implication;

  Implication(SignalSymbol *ss1, SignalSymbol*ss2)
    : predicate(ss1), implication(ss2) {}"
```

12b ⟨*Modules* 12b⟩≡ (10c)

```
class "Modules : ASTNode
  SymbolTable module_symbols;
  vector<Module*> modules;

  void add(Module*);
" "
void Modules::add(Module* m) {
  assert(m);
  assert(m->symbol);
  assert(!module_symbols.contains(m->symbol->name));
  modules.push_back(m);
  module_symbols.enter(m->symbol);
}"
```

6 Statements

```
Abort
Assignment
Await
DoUpto
DoWatching
Emit
Every
Exec
Exit
Halt
If
Loop
LoopEach
Nothing
ParallelStatementList
Pause
Present
ProcedureCall
Repeat
Run
Signal
StatementList
Suspend
Sustain
TaskCall
Trap
Var
```

13 ⟨*Statement classes* 13⟩≡ (27)
⟨*Statement* 14a⟩

⟨*BodyStatement* 14b⟩
⟨*PredicatedStatement* 14c⟩
⟨*CaseStatement* 14d⟩

⟨*StatementList* 14e⟩
⟨*ParallelStatementList* 14f⟩

⟨*Basic Statements* 15a⟩
⟨*ProcedureCall* 15b⟩
⟨*Conditional Statements* 15c⟩
⟨*Iteration Statements* 16a⟩
⟨*Preemption Statements* 16b⟩
⟨*Task Statements* 17a⟩
⟨*Scope Statements* 17b⟩

14a ⟨*Statement* 14a⟩≡ (13)
```
abstract "Statement : ASTNode"
```

Helper statements are used as parts of other high-level statements or as base classes. A `BodyStatement` is simply one that contains another. A Boolean predicate expression controls the execution of the body of a `PredicatedStatement`. A `CaseStatement` is an abstract notion of a series of choices: if the first predicate is true, execute the first body, else check and execute the second, etc. If none hold, execute the optional default.

14b ⟨*BodyStatement* 14b⟩≡ (13)
```
abstract "BodyStatement : Statement
  Statement *body;

  BodyStatement(Statement *s) : body(s) {}"
```

14c ⟨*PredicatedStatement* 14c⟩≡ (13)
```
class "PredicatedStatement : BodyStatement
  Expression *predicate;

  PredicatedStatement(Statement *s, Expression *e)
    : BodyStatement(s), predicate(e) {}"
```

14d ⟨*CaseStatement* 14d⟩≡ (13)
```
abstract "CaseStatement : Statement
  vector<PredicatedStatement *> cases;
  Statement *default_stmt;

  CaseStatement() : default_stmt(0) {}
  PredicatedStatement *newCase(Statement *s, Expression *e) {
    PredicatedStatement *ps = new PredicatedStatement(s, e);
    cases.push_back(ps);
    return ps;
  }"
```

`StatementList` handles sequences of statements, i.e., those separated by `;`; `ParallelStatementList` handles sequences separated by `||`.

14e ⟨*StatementList* 14e⟩≡ (13)
```
class "StatementList : Statement
  vector<Statement *> statements;

  StatementList& operator <<(Statement *s) {
    assert(s);
    statements.push_back(s);
    return *this;
  }"
```

14f ⟨*ParallelStatementList* 14f⟩≡ (13)
```
class "ParallelStatementList : Statement
  vector<Statement *> threads;"
```

Nothing does nothing, pause delays a cycle, halt delays indefinitely, emit emits a signal, perhaps with a value, sustain emits a signal continously, and the assignment statement implements `:=`, assignment to a variable.

15a ⟨*Basic Statements* 15a⟩≡ (13)

```
class "Nothing : Statement"
class "Pause : Statement"
class "Halt : Statement"

class "StartCounter : Statement
  Expression *predicate;
  Expression *startvalue;
  VariableSymbol *counter;
  STNode *st;

  StartCounter(Expression *p, Expression *sv, VariableSymbol *v): predicate(p), startvalue(sv), counter(v

class "Emit : Statement
  SignalSymbol *signal;
  Expression *value;

  Emit(SignalSymbol *s, Expression *e) : signal(s), value(e) {}"

class "Sustain : Emit

  Sustain(SignalSymbol *s, Expression *e) : Emit(s, e) {}"

class "Assignment : Statement
  VariableSymbol *variable;
  Expression *value;

  Assignment(VariableSymbol *v, Expression *e) : variable(v), value(e) {}"
```

Procedure call is a statement that takes a procedure, a collection of pass-by-reference arguments, and a collection of pass-by-value arguments.

15b ⟨*ProcedureCall* 15b⟩≡ (13)

```
class "ProcedureCall : Statement
  ProcedureSymbol *procedure;
  vector<VariableSymbol*> reference_args;
  vector<Expression*> value_args;

  ProcedureCall(ProcedureSymbol *ps) : procedure(ps) {}"
```

Conditional statements test their expressions. Esterel draws a textual distinction between the two; I don't.

15c ⟨*Conditional Statements* 15c⟩≡ (13)

```
class "Present : CaseStatement"
class "If : CaseStatement"
```

16a ⟨*Iteration Statements* 16a⟩≡ (13)
```
class "Loop : BodyStatement

  Loop(Statement *s) : BodyStatement(s) {}"

class "Repeat : Loop
  Expression *count;
  bool is_positive;

  Repeat(Statement *s, Expression *e, bool p)
    : Loop(s), count(e), is_positive(p) {}"
```

The code in the `abort` statement is only used for weak aborts; it is the code used to signal normal termination of the body.

16b ⟨*Preemption Statements* 16b⟩≡ (13)
```
class "Abort : CaseStatement
  Statement *body;
  bool is_weak;
  int code;

  Abort(Statement *s, bool i) : body(s), is_weak(i), code(0) {}
  Abort(Statement *s, Expression *e, Statement *s1)
    : body(s), is_weak(false), code(0) {
    newCase(s1, e);
  }"

class "Await : CaseStatement"

class "LoopEach : PredicatedStatement

  LoopEach(Statement *s, Expression *e) : PredicatedStatement(s, e) {}"

class "Every : PredicatedStatement

  Every(Statement *s, Expression *e) : PredicatedStatement(s, e) {}"

class "Suspend : PredicatedStatement

  Suspend(Statement *s, Expression *e) : PredicatedStatement(s, e) {}"

class "DoWatching : PredicatedStatement
  Statement *timeout;

  DoWatching(Statement *s1, Expression *e, Statement *s2)
    : PredicatedStatement(s1, e), timeout(s2) {}"

class "DoUpto : PredicatedStatement

  DoUpto(Statement *s, Expression *e) : PredicatedStatement(s, e) {}"
```

17a ⟨*Task Statements* 17a⟩≡ (13)

```
class "TaskCall : ProcedureCall
  SignalSymbol *signal;
  Statement *body;

  TaskCall(TaskSymbol *ts) : ProcedureCall(ts), signal(0), body(0) {}
"

class "Exec : Statement
  vector <TaskCall *> calls;"
```

17b ⟨*Scope Statements* 17b⟩≡ (13)

```
abstract "ScopeStatement : BodyStatement
   SymbolTable *symbols;"

class "Trap : ScopeStatement
  vector<PredicatedStatement *> handlers;

  PredicatedStatement* newHandler(Expression *e, Statement *s) {
    PredicatedStatement *ps = new PredicatedStatement(s, e);
    handlers.push_back(ps);
    return ps;
  }"

class "Exit : Statement
  TrapSymbol *trap;
  Expression *value;

  Exit(TrapSymbol *t, Expression *e) : trap(t), value(e) {}"

class "Signal : ScopeStatement"

class "Var : ScopeStatement"
```

7 The Run Statement

18 ⟨*Run classes* 18⟩≡ (27)
```
abstract "Renaming : ASTNode
  string old_name;

Renaming(string s) : old_name(s) {}"

class "TypeRenaming : Renaming
  TypeSymbol *new_type;

  TypeRenaming(string s, TypeSymbol *t) : Renaming(s), new_type(t) {}"

class "ConstantRenaming : Renaming
  Expression *new_value;

  ConstantRenaming(string s, Expression *e) : Renaming(s), new_value(e) {}"

class "FunctionRenaming : Renaming
  FunctionSymbol *new_func;

  FunctionRenaming(string s, FunctionSymbol *f) : Renaming(s), new_func(f) {}"

class "ProcedureRenaming : Renaming
  ProcedureSymbol *new_proc;

  ProcedureRenaming(string s, ProcedureSymbol *p)
    : Renaming(s), new_proc(p) {}"

class "SignalRenaming : Renaming
  SignalSymbol *new_sig;

  SignalRenaming(string s, SignalSymbol *ss) : Renaming(s), new_sig(ss) {}"

class "Run : Statement
  string old_name;
  string new_name;
  vector<TypeRenaming *> types;
  vector<ConstantRenaming *> constants;
  vector<FunctionRenaming *> functions;
  vector<ProcedureRenaming *> procedures;
  vector<ProcedureRenaming *> tasks;
  vector<SignalRenaming *> signals;

  Run(string s) : old_name(s), new_name(s) {}"
```

8 Low-Level Statements

Low-level statements are produced by dismantling high-level Esterel statements. Semantically, they are meant to be similar to the IC format, but have a more traditional imperative langauge flavor. In particular, each is meant to have a straightforward translation into expressions, tests, and gotos.

var := *expr*
if (*expr*) { *stmts* } else { *stmts* }
Label:
goto *Label*

break *n*
continue
try { *stmts* } catch 2 { *stmts* } catch 3 { *stmts* } ...
resume { *stmts* } catch 1 { *stmts* } ...
thread { *stmts* } catch 1 { *stmts* } ...
parallel { *threads* } catch 1 { *stmts* } catch 2 { *stmts* } ...

fork *Label1*, *Label2*, ...
join

These statements were designed to express Esterel's facilities for exceptions, concurrency, and pausing between cycles. The *try* statement runs its body. Executing an enclosed *break* statement passes control to a matching *catch* clause. *Resume* is a *try* that restarts its body after the last *break 1* statement when a *continue* statement is executed within one of its *catch* clauses. *Parallel* is a *resume* that runs the *thread* statements in its body concurrently. *Fork* and *synchronize* are low-level initiators and collectors of concurrent behavior. Signal housekeepers read and write signals to and from the environment, and reset signal presence.

19a ⟨*low-level classes* 19a⟩≡ (27) 19b▷
```
class "IfThenElse : Statement
  Expression *predicate;
  Statement *then_part;
  Statement *else_part;

  IfThenElse(Expression *e) : predicate(e) , then_part(0), else_part(0) {}
  IfThenElse(Expression *e, Statement *s1, Statement *s2)
    : predicate(e) , then_part(s1), else_part(s2) {}"
```

19b ⟨*low-level classes* 19a⟩+≡ (27) ◁19a 20a▷
```
class "Goto : Statement
  Label *target;

  Goto(Label *s) : target(s) {}"
```

20a ⟨*low-level classes* 19a⟩+≡ (27) ◁19b 20b▷
```
class "Label : Statement
  string name;

  Label(string s) : name(s) {}"
```

The *try*, *resume*, and *parallel* statements all handle numeric exceptions. Handlers are presented by `CatchClauses`. The `code` is the completion code caught by the clause, and `label` is the label at the start of the body, used when dismantling the statement into a still-lower representation.

20b ⟨*low-level classes* 19a⟩+≡ (27) ◁20a 20c▷
```
class "Catch : BodyStatement
  int code;
  Label *label;

  Catch(Statement *s, int c, Label *l) : BodyStatement(s), code(c), label(l) {}
"
```

20c ⟨*low-level classes* 19a⟩+≡ (27) ◁20b 20d▷
```
abstract "Handler : Statement
  vector<Catch*> catches;
  Label *catch0;

  Handler(Label *c0) : catch0(c0) {}
  void newCatch(Statement *s, int c, Label *l) {
    catches.push_back(new Catch(s, c, l));
  }
"
```

The *try*, *break*, and *catch* statements work together as follows to implement Esterel's trap statements:

(a)	(b)	(c)	
trap T1 in	try {		
exit T1	break 2		`goto Catch2;`
			`goto Catch0;`
handle T1 do	} catch 2 {	`Catch2:`	
c := 1	c := 1		`c = 1;`
end	}	`Catch0:`	

(a) An Esterel *trap* statement with handler. (b) Its translation into low-level statements. The *exit* becomes a *break*. (b) Its translation into C. The *break* becomes a *goto*.

20d ⟨*low-level classes* 19a⟩+≡ (27) ◁20c 21a▷
```
class "Try : Handler
  Statement *body;

  Try(Label *c0, Statement *s) : Handler(c0), body(s) {}
"
```

The break statement raises a numeric exception (or completion code). `continue` is set for break 1 statements, which can resume in later cycles. `set_state` holds the statement, if any, that sets the state before branching to the catch clause. This field is set by the state assignment procedure and is used during the dismantling procedure.

21a ⟨*low-level classes* 19a⟩+≡ (27) ◁20d 21b▷
```
class "Break : Statement
  int level;
  Label *continue_target;
  Statement *set_state;

  Break(int l) : level(l), continue_target(0), set_state(0) {}"
```

The *resume* statement uses `break 1` to implement Esterel's ability to pause and abort groups of statements as follows:

```
abort      resume {                  goto Ent
                             Cont:  switch (s) {
                                    case 0: goto State0;
                                    case 1: goto State1;
                                    }
  pause      break 1         Ent:   s = 0; goto Catch1; State0:
  pause      break 1                s = 1; goto Catch1; State1:
                                    goto Catch0;
           } catch 1 {       Catch1:
             break 1                s1 = 0; goto Catch1o; State0o:
when A       if (!A) continue       if (!A) goto Cont;
           }                 Catch0:
  (a)           (b)                           (c)
```

(a) An Esterel *abort* statement with *pauses*. (b) Its translation into low-level statements. (c) Its translation into C. Note the unusual placement of labels on the right to make the translation line-to-line. The `Catch1o` and `State0o` labels belong to the *resume* statement that encloses this one (not shown).

21b ⟨*low-level classes* 19a⟩+≡ (27) ◁21a 21c▷
```
abstract "Continuable : Handler
  Label *continue_label;

  Continuable(Label *c0, Label *cont) : Handler(c0), continue_label(cont) {}
"
```

21c ⟨*low-level classes* 19a⟩+≡ (27) ◁21b 22a▷
```
class "Resume : Continuable
  Statement *body;
  Statement *branch;

  Resume(Label *c0, Label *cont, Statement *s)
    : Continuable(c0, cont), body(s), branch(0) {}
"
```

22a ⟨*low-level classes* 19a⟩+≡ (27) ◁21c 22b▷
```
class "Continue : Statement"
```

The *parallel* statement is like *resume* except that it runs the statements in its body (assumed to all be resume statements) concurrently. A *continue* in its handler restarts all the resumes. The `synchronize` statement is assumed to contain code that collects the completion codes from this *parallel*'s threads and dispatches control to its catch clause handlers.

22b ⟨*low-level classes* 19a⟩+≡ (27) ◁22a 22c▷
```
class "Thread : Resume
  VariableSymbol *exit_level;

  Thread(Label *c0, Label *cont, Statement *b, VariableSymbol *el)
     : Resume(c0, cont, b), exit_level(el) {}
"
```

22c ⟨*low-level classes* 19a⟩+≡ (27) ◁22b
```
class "Parallel : Continuable
  vector<Thread*> threads;
  Statement *synchronize;

  Parallel(Label *c0, Label *cont)
    : Continuable(c0, cont), synchronize(0) {}
  void newThread(Label *c0, Label *cont, Statement *b, VariableSymbol *el) {
    threads.push_back(new Thread(c0, cont, b, el));
  }
"
```

The *synchronize* statement is a multi-way branch used to handle completion codes generated by parallel statements. Its `decision_operand` field is null and its `default_target` field is also null.

FIXME: What should synchronize contain?

22d ⟨*low-level classe* 22d⟩≡
```
class "Synchronize : Statement

"
```

9 GRC Nodes

These follow the GRC format defined in Potop-Butcaru's thesis.

Successors may contain NULL nodes; these are used, e.g., to represent an unused continuation from a parallel synchronizer. Predecessors should all be non-NULL.

9.1 GRCNode

23a ⟨*GRC classes* 23a⟩≡ (27) 23b ▷

```
abstract "GRCNode : ASTNode
  vector<GRCNode*> predecessors;
  vector<GRCNode*> successors;

  virtual Status welcome(Visitor&) = 0;

  GRCNode& operator >>(GRCNode*);
" "
  GRCNode& GRCNode::operator >>(GRCNode * s) {
    successors.push_back(s);
    if (s) s->predecessors.push_back(this);
    return *this;
  }
"
```

9.2 STNodes

23b ⟨*GRC classes* 23a⟩+≡ (27) ◁ 23a 23c ▷

```
class "STNode : GRCNode"
```

23c ⟨*GRC classes* 23a⟩+≡ (27) ◁ 23b 23d ▷

```
class "STexcl : STNode"
```

23d ⟨*GRC classes* 23a⟩+≡ (27) ◁ 23c 23e ▷

```
class "STpar : STNode"
```

23e ⟨*GRC classes* 23a⟩+≡ (27) ◁ 23d 24a ▷

```
class "STref : STNode
  int type;

  STref(): type(0) {}

  int isabort() { return type == 1;}
  void setabort() { type = 1;}
  int issuspend() { return type == 2;}
  void setsuspend() { type = 2;}
"
```

24a ⟨*GRC classes* 23a⟩+≡ (27) ◁23e 24b▷
```
class "STleaf : STNode
  int type;

  STleaf(): type(0) {}

  int isfinal() { return type == 1;}
  void setfinal() { type = 1;}
"
```

9.3 additional flow control

24b ⟨*GRC classes* 23a⟩+≡ (27) ◁24a 24c▷
```
class "TopGRC : GRCNode"
```

24c ⟨*GRC classes* 23a⟩+≡ (27) ◁24b 24d▷
```
class "Nop : GRCNode
  int type;
  int code;

  Nop(): type(0), code(0) {}

  int isflowin() { return type == 1;}
  void setflowin() { type = 1;}
  // a shorcut Nop gives "up" flow to child 0
  int isshortcut() { return type == 2;}
  void setshortcut() { type = 2;}
"
```

24d ⟨*GRC classes* 23a⟩+≡ (27) ◁24c 24e▷
```
class "RecT1 : GRCNode"
```

24e ⟨*GRC classes* 23a⟩+≡ (27) ◁24d 24f▷
```
class "DefineSignal : GRCNode
  SymbolTable *signals;

  DefineSignal(Signal &s) : signals(s.symbols){}
"
```

24f ⟨*GRC classes* 23a⟩+≡ (27) ◁24e 25a▷
```
class "UndefineSignal : GRCNode
  SymbolTable *signals;

  UndefineSignal(Signal &s): signals(s.symbols) {}
"
```

9.4 Switch

Multi-way branch on the state of a thread.

25a ⟨*GRC classes* 23a⟩+≡ (27) ◁24f 25b▷

```
class "Switch : GRCNode
  STNode *st;
"
```

9.5 Test

An if-then-else statement.

25b ⟨*GRC classes* 23a⟩+≡ (27) ◁25a 25c▷

```
class "Test : GRCNode
  Expression *predicate;
  STNode *st;

  Test(Expression *e) : predicate(e) {}
"
```

9.6 Fork

Sends control to all its successors; just fan-out in the circuit.

25c ⟨*GRC classes* 23a⟩+≡ (27) ◁25b 25d▷

```
class "Fork : GRCNode
"
```

9.7 Sync and Terminate

A parallel synchronizer; mostly a placeholder. Its predecessors should all be Terminate nodes.

25d ⟨*GRC classes* 23a⟩+≡ (27) ◁25c 26a▷

```
class "Sync : GRCNode
  STNode *st;
  int depth;

  Sync() : depth(0) {}

  void setdepth() { depth = 1; }
  int isdepth() { return depth == 1; }
"
```

Terminates a thread with the given completion code. Should have a single successor, a `Sync` node.

26a ⟨*GRC classes* 23a⟩+≡ (27) ◁25d 26b▷
```
class "Terminate : GRCNode
  int code;

  Terminate(int c) : code(c) {}
"
```

9.8 Action

Perform an action such as emission or assignment. Should have a single successor.

26b ⟨*GRC classes* 23a⟩+≡ (27) ◁26a 26c▷
```
class "Action : GRCNode
  Statement *body;

  Action(Statement *s) : body(s) {}
"
```

9.9 Enter and Leave

The represent the activation and deactivation of a particular statement.

26c ⟨*GRC classes* 23a⟩+≡ (27) ◁26b 26d▷
```
class "Enter : GRCNode
  STNode *st;
"
```

26d ⟨*GRC classes* 23a⟩+≡ (27) ◁26c
```
class "Leave : GRCNode
  STNode *st;
"
```

10 The Shell Script

27 ⟨*AST.sh* 27⟩≡

```
#!/bin/sh

abstract() {
  class "$1" "$2" "abstract"
}

class() {
  # The classname is the string before the : on the first line
  classname=`echo "$1" | sed -n '1 s/ *:.*$//p'`
  # The parent's class name is the string after the : on the first line
  parent=`echo "$1" | sed -n '1 s/^.*: *//p'` ; # String after :
  # The fields come from the second line through the first empty line
  # Each is the identifier just before the semicolon
  fields=`echo "$1" | sed '/typedef/d' | sed -n '2,/^\$/ s/^.*[^a-zA-Z0-9_]\([a-zA-Z0-9_]*\);.*/\1/p'`
  # The body for the header file starts at the second line
  hppbody=`echo "$1" | sed -n '2,$p'`

  # Any additional methods are defined in the second argument

  #echo "[$classname]"
  #echo "[$parent]"
  #echo "[$fields]"
  #echo "[$hppbody]"

  forwarddefs="$forwarddefs
  class $classname;"

  # Define a default (zero-argument) constructor if one isn't already
  # defined in the body
  if (echo $hppbody | grep -q "$classname()"); then
    defaultconstructor=
  else
    defaultconstructor="$classname() {}
"
  fi

  if test -z "$3"; then
      visitorclassdefs="$visitorclassdefs
  virtual Status visit($classname& n) { assert(0); return Status(); }"
      welcome="
    IRCLASSDEFS;
  public:
    Status welcome(Visitor&);"
      welcomedef="
  IRCLASS($classname);
  Status $classname::welcome(Visitor &v) { return v.visit(*this); }"
  else
```

```
    welcome="public:"
    welcomedef=
  fi

  classdefs="$classdefs

  class $classname : public $parent {
    $welcome
    void read(XMListream &);
    void write(XMLostream &) const;
    $defaultconstructor
$hppbody
  };
"

  if test -n "$fields"; then
    writefields=`echo $fields | sed "s/ / << /g"`;
    writefields="
    w << $writefields;"
    readfields=`echo $fields | sed "s/ / >> /g"`;
    readfields="
    r >> $readfields;"
  else
    readfields=
    writefields=
  fi

  methoddefs="$methoddefs

  void $classname::read(XMListream &r) {
    $parent::read(r); $readfields
  }

  void $classname::write(XMLostream &w) const {
    $parent::write(w); $writefields
  }
$welcomedef
  $2
  "
}
```

⟨*ASTNode class* 2a⟩
⟨*Symbol classes* 2b⟩
⟨*SymbolTable* 6⟩
⟨*Expression classes* 7a⟩
⟨*Module classes* 10c⟩
⟨*Statement classes* 13⟩
⟨*Run classes* 18⟩
⟨*low-level classes* 19a⟩
⟨*GRC classes* 23a⟩

```
######################################################################

echo "#ifndef _AST_HPP
#  define _AST_HPP

/* Automatically generated by AST.sh -- do not edit */

#  include \"IR.hpp\"
#  include <string>
#  include <vector>
#  include <map>

namespace AST {
  using IR::Node;
  using IR::XMListream;
  using IR::XMLostream;
  using std::string;
  using std::vector;
  using std::map;

  class Visitor;
$forwarddefs

  union Status {
    int i;
    ASTNode *n;
    Status() {}
    Status(int ii) : i(ii) {}
    Status(ASTNode *nn) : n(nn) {}
  };

$classdefs

  class Visitor {
  public:
$visitorclassdefs
  };

}

#endif
" > AST.hpp

echo "/* Automatically generated by AST.sh -- do not edit */
#include \"AST.hpp\"
namespace AST {

$methoddefs
```

```
}
" > AST.cpp
```