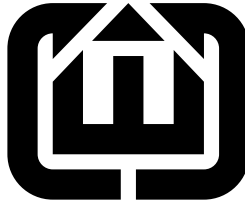


# CEC C Code Printer



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

## Contents

<b>1</b>	<b>The Printer Class</b>	<b>2</b>
<b>2</b>	<b>Name Management</b>	<b>4</b>
<b>3</b>	<b>GRC Node printers</b>	<b>4</b>
3.1	Test and Action . . . . .	4
3.2	Do-nothing nodes . . . . .	4
3.3	Nop . . . . .	5
3.4	Switch . . . . .	5
3.5	Enter . . . . .	5
3.6	Terminate . . . . .	7
3.7	Sync . . . . .	7
3.8	Fork . . . . .	8
<b>4</b>	<b>Statement Printers</b>	<b>8</b>
4.1	Emit and Exit . . . . .	8
4.2	DefineSignal . . . . .	9
4.3	Assign . . . . .	10
4.4	StartCounter . . . . .	10
4.5	CheckCounter . . . . .	10
<b>5</b>	<b>Expression Printers</b>	<b>11</b>
5.1	LoadSignalExpression and LoadSignalValueExpression . . . . .	11
5.2	LoadVariableExpression . . . . .	12
5.3	Unary and BinaryOp . . . . .	12
5.4	Literal . . . . .	13

5.5	Function Call . . . . .	14
5.6	Procedure Call . . . . .	15
<b>6</b>	<b>Overall declarations</b>	<b>15</b>
<b>7</b>	<b>Declarations for variables, functions, procedures, etc.</b>	<b>16</b>
<b>8</b>	<b>Output Functions</b>	<b>25</b>
<b>9</b>	<b>Reset inputs</b>	<b>26</b>
<b>10</b>	<b>I/O function printers</b>	<b>26</b>
<b>11</b>	<b>Structured Code Generation</b>	<b>28</b>
11.1	DFS node numbering . . . . .	29
11.2	Compute Reverse Dominators . . . . .	30
11.3	C Statements . . . . .	31
11.4	Statement synthesis . . . . .	33
<b>12</b>	<b>Utilities</b>	<b>36</b>
<b>13</b>	<b>Top-Level Files</b>	<b>37</b>

## 1 The Printer Class

```

2  <printer class 2>≡
    class Printer : public Visitor {
        typedef map<GRCNode *, int> CFGmap;
        typedef map<STNode *, int> STmap;
        CFGmap cfgmap;
        STmap stmap;

        vector<GRCNode*> nodes;
        CFGmap nodeNumber;
        map<GRCNode*, GRCNode*> ridom;

        map<GRCNode *, CStatement*> statementFor;

        static int nextLabel;
    public:
        std::ostream &o;
        Module &m;
        bool do_threevalued;
        GRCgraph *g;

        map<GRCNode *, string> labelFor;

        set<string> identifiers; // All C identifiers for avoiding name collisions

```

```

// C identifiers for various objects

typedef map<Counter *, string> CounterNames;
CounterNames counterVar;

typedef map<SignalSymbol *, string> SignalNames;
SignalNames presenceVar;
SignalNames valueVar;

typedef map<STexcl *, string> StateNames;
StateNames stateVar;

typedef map<Sync *, string> TerminationNames;
TerminationNames terminationVar;

typedef map<VariableSymbol *, string> VariableNames;
VariableNames variableVar;

Printer(std::ostream &, Module &, bool);
virtual ~Printer() {}

    <declarations 4a>
};

3a <definitions 3a>≡
    int Printer::nextLabel = 0;

3b <definitions 3a>+≡
Printer::Printer(std::ostream &o, Module &m, bool three_val)
    : o(o), m(m), do_threevalued(three_val)
{
    g = dynamic_cast<AST::GRCgraph*>(m.body);
    if (!g) throw IR::Error("Module is not in GRC format");

    // Enumerate selection tree and CFG nodes
    g->enumerate(cfgmap, stmap);

    // Enter C reserved words into the identifiers list to avoid collisions

    // Note: float and double aren't in this list because they are equivalent
    // to Esterel's types of the same name

    char *keywords[] = {
        "int", "break", "char", "continue", "if", "else",
        "struct", "for", "auto", "do", "extern", "while", "register", "switch",
        "static", "case", "goto", "default", "return", "entry", "sizeof", NULL
    };

    for (char **k = keywords ; *k != NULL ; k++) identifiers.insert(*k);
}

```

## 2 Name Management

Return a unique identifier for the given name. Enters the name into the `identifiers` set to make sure its unique.

```
4a <declarations 4a>≡
    string uniqueID(string);

4b <definitions 3a>+≡
    string Printer::uniqueID(string name)
    {
        string newname = name;

        char buf[10];
        int version = 1;

        while (contains(identifiers, newname)) {
            sprintf(buf, "%d", version++);
            newname = name + '_' + buf;
        }

        identifiers.insert(newname);
        return newname;
    }
```

## 3 GRC Node printers

The main method here is `printExpr`, which writes a C *expression* for the given node to the output stream. This expression often has side effects, such as an assignment, but for conditional nodes, it returns the value of the node, which can be used as an argument in, say, an if-then-else statement.

```
4c <declarations 4a>+≡
    void printExpr(ASTNode *n) { n->welcome(*this); }
```

### 3.1 Test and Action

These nodes contain expressions or statements that generate the real code.

```
4d <declarations 4a>+≡
    Status visit(Test &t) { printExpr(t.predicate); return Status(); }
    Status visit(Action &a) { printExpr(a.body); return Status(); }
```

### 3.2 Do-nothing nodes

These nodes are placeholders.

```
4e <declarations 4a>+≡
    Status visit(EnterGRC&) { o << "1 /* EnterGRC */"; return Status(); }
    Status visit(ExitGRC&) { o << "/* ExitGRC */"; return Status(); }
    Status visit(STSuspend&) { o << "/* STSuspend */"; return Status(); }
```

### 3.3 Nop

You can hide arbitrary code in a string in a Nop node and have it emitted.

```
5a <declarations 4a>+≡
    Status visit(Nop& n) { o << n.body; return Status(); }
```

### 3.4 Switch

A switch node by itself returns an expression for its state variable.

```
5b <declarations 4a>+≡
    Status visit(Switch &s) {
        STexcl *e = dynamic_cast<STexcl*>(s.st);
        assert(e);
        assert(contains(stateVar, e));
        o << stateVar[e];
        return Status();
    }
```

### 3.5 Enter

An enter node sets the value of its state depending on which child it is under. The visitor walks up the selection tree, starting at the selection tree node of the enter node, looking for the first exclusive node. The state value is simply the child number we came in on.

```
5c <declarations 4a>+≡
    Status visit(Enter &);
```

```

6  <definitions 3a>+≡
    Status Printer::visit(Enter &e)
    {
        STexcl *exclusive = 0;
        STNode *n = e.st;

        for (;;) {
            assert(n);

            STNode *parent = n->parent;

            // If we hit a parallel first, this Enter is unnecessary; do not generate
            // any code
            if (dynamic_cast<STpar*>(parent) != NULL) return Status();

            exclusive = dynamic_cast<STexcl*>(parent);
            if (exclusive != NULL) break; // found the exclusive node
            n = parent;
        }

        assert(exclusive != NULL);

        // Locate node n among the children of "parent"
        vector<STNode*>::iterator i = exclusive->children.begin();
        while (*i != n && i != exclusive->children.end()) i++;

        assert(i != exclusive->children.end());

        int childnum = i - exclusive->children.begin();

        assert(childnum >= 0);

        assert(contains(stateVar, exclusive));
        o << stateVar[exclusive] << " = " << childnum;

        return Status();
    }

```

### 3.6 Terminate

These update the termination level of sync node. This uses a clever encoding that allows a bitwise AND operation to perform the maximum calculation.

level	encoding	binary
0	-1	1111
1	-2	1110
2	-4	1100
3	-8	1000

7a *<declarations 4a>*+≡  
 Status visit(Terminate &);

7b *<definitions 3a>*+≡  
 Status Printer::visit(Terminate &t)  
 {  
 // If we have something other than a single data successor or it is not  
 // a Sync, return nothing.  
 if (t.dataSuccessors.size() != 1 || t.code == 0) {  
 o << "/\* Vacuous terminate \*/";  
 return Status();  
 }  
  
 Sync \*s = dynamic\_cast<Sync\*>(t.dataSuccessors.front());  
 if (s == NULL) return Status();  
  
 if ( contains(terminationVar, s) )  
 o << terminationVar[s] << " &= -(1 << " << t.code << " )";  
 return Status();  
 }

### 3.7 Sync

The sync node returns the value of its termination level. The encoding is a little unorthodox because of the trick used by Terminate nodes (see above):

level	result
0	0
1	1
2	3
3	7
4	15

7c *<declarations 4a>*+≡  
 Status visit(Sync &s) {  
 if ( contains(terminationVar, &s) )  
 o << '~' << terminationVar[&s];  
 return Status();  
 }

### 3.8 Fork

A fork node resets the termination level for its Sync node, if it has one.

```
8a <declarations 4a>+≡
    Status visit(Fork &f) {
        if (f.sync && contains(terminationVar, f.sync))
            o << terminationVar[f.sync] << " = -1";
        return Status();
    }
```

## 4 Statement Printers

### 4.1 Emit and Exit

These both assign an optional value, if present, then set their respective presence variables.

FIXME: Emit must be fixed to work with "combine" signals.

```
8b <declarations 4a>+≡
    Status visit(Emit &);
    Status visit(Exit &);

8c <definitions 3a>+≡
    Status Printer::visit(Emit &e)
    {
        assert(e.signal);
        if (e.signal->type != NULL) {
            assert(contains(valueVar, e.signal));
            o << "(";
            if (e.value->type->name == "string") {
                o << "strcpy(" << valueVar[e.signal] << ", ";
                printExpr(e.value);
                o << ")";
            } else {
                o << valueVar[e.signal] << " = ";
                printExpr(e.value);
            }
            o << "), (";
        }
        assert(contains(presenceVar, e.signal));
        if (e.unknown)
            o << presenceVar[e.signal] << "_unknown = 1";
        else
            o << presenceVar[e.signal] << " = 1";
        if (e.signal->type) o << ")";
        return Status();
    }
```



```

9a  <definitions 3a>+≡
    Status Printer::visit(Exit &e)
    {
        assert(e.trap);
        if (e.trap->type) {
            assert(contains(valueVar, e.trap));
            o << valueVar[e.trap] << " = ";
            printExpr(e.value);
            o << ", ";
        }
        assert(contains(presenceVar, e.trap));
        o << presenceVar[e.trap] << " = 1";
        return Status();
    }

```

## 4.2 DefineSignal

This resets the presence of a (local) signal.

```

9b  <declarations 4a>+≡
    Status visit(DefineSignal &d)
    {
        assert(contains(presenceVar, d.signal));
        o << presenceVar[d.signal] << " = 0";
        if (d.signal->initializer && d.is_surface) {
            o << ", ";
            assert(contains(valueVar, d.signal));
            if (d.signal->initializer->type->name == "string") {
                o << "strcpy(" << valueVar[d.signal] << ", ";
                printExpr(d.signal->initializer);
                o << ");";
            } else {
                o << valueVar[d.signal] << " = ";
                printExpr(d.signal->initializer);
            }
        }
        o << ',';
        return Status();
    }

```

### 4.3 Assign

```

10a  <declarations 4a>+≡
      Status visit(Assign &a) {
          assert(a.variable->type);
          assert(contains(variableVar, a.variable));

          if (a.variable->type->name == "string") {
              // Use strcpy for strings
              o << "strcpy(" << variableVar[a.variable] << ", ";
              printExpr(a.value);
              o << ")";
          } else if ( dynamic_cast<BuiltinTypeSymbol*>(a.variable->type) ) {
              // Use assignment for other built-in types
              o << variableVar[a.variable] << " = ";
              printExpr(a.value);
          } else {
              // Call _<typename>(&lvalue, rvalue) for user-defined types
              o << '_' << a.variable->type->name
                << "&" << variableVar[a.variable] << ", ";
              printExpr(a.value);
              o << ')';
          }
          return Status();
      }

```

### 4.4 StartCounter

This assigns the initial count value to the given counter.

```

10b  <declarations 4a>+≡
      Status visit(StartCounter &);

10c  <definitions 3a>+≡
      Status Printer::visit(StartCounter &s)
      {
          assert(s.counter);
          assert(contains(counterVar, s.counter));
          o << counterVar[s.counter] << " = ";
          printExpr(s.count);
          return Status();
      }

```

### 4.5 CheckCounter

This decrements the counter if its predicate is true and returns true if the counter has reached 0.

```

10d  <declarations 4a>+≡
      Status visit(CheckCounter &);

```

```

11a  <definitions 3a>+≡
      Status Printer::visit(CheckCounter &s)
      {
        assert(s.counter);
        assert(s.predicate);
        assert(contains(counterVar, s.counter));
        // FIXME: Is this safe?
        if (dynamic_cast<LoadVariableExpression*>(s.predicate)) {
          o << "0 == --" << counterVar[s.counter];
        } else {
          o << "0 == (";
          printExpr(s.predicate);
          o << " ? --" << counterVar[s.counter] << " : "
            << counterVar[s.counter] << ")";
        }
        return Status();
      }

```

## 5 Expression Printers

### 5.1 LoadSignalExpression and LoadSignalValueExpression

```

11b  <declarations 4a>+≡
      Status visit(LoadSignalExpression &e) {
        assert(contains(presenceVar, e.signal));
        o << presenceVar[e.signal];
        return Status();
      }

```

This is straightforward unless the signal being read is a sensor. We are only allowed to read a sensor (by calling its input function, e.g., `MODULE_S_SENSOR()`) once a cycle. We use its presence variable to track whether the sensor has been read or not, for something like `?SENSOR`, generating

```
( SENSOR ? SENSOR_v : (SENSOR = 1, SENSOR_v = MODULE_S_SENSOR()) )
```

```
12a <declarations 4a>+≡
    Status visit(LoadSignalValueExpression &e) {
        assert(e.signal);
        assert(contains(valueVar, e.signal));
        assert(contains(presenceVar, e.signal));
        if (e.signal->kind == SignalSymbol::Sensor) {
            o << "( " << presenceVar[e.signal] << " ? " << valueVar[e.signal]
              << " : (" << presenceVar[e.signal] << " = 1,"
              << valueVar[e.signal] << " = "
              << m.symbol->name << "_S_" << e.signal->name << "()) )";
        } else {
            o << valueVar[e.signal];
        }
        return Status();
    }
```

## 5.2 LoadVariableExpression

```
12b <declarations 4a>+≡
    Status visit(LoadVariableExpression &e) {
        assert(contains(variableVar, e.variable));
        o << variableVar[e.variable];
        return Status();
    }
```

## 5.3 Unary and BinaryOp

```
12c <definitions 3a>+≡
    Status Printer::visit(UnaryOp &op)
    {
        o << '(';
        string s = op.op;
        if (s == "not") s = "!";
        o << s;
        assert(op.source);
        printExpr(op.source);
        o << ')';
        return Status();
    }
```

```

13a  <definitions 3a>+≡
      Status Printer::visit(BinaryOp &op)
      {
          o << '(';
          assert(op.source1);
          printExpr(op.source1);
          string s = op.op;
          if (s == "mod") s = "%";
          else if (s == "=") s = "==";
          else if (s == "<>") s = "!=";
          else if (s == "and") s = "&&";
          else if (s == "or") s = "||";
          o << ' ' << s << ' ';
          assert(op.source2);
          printExpr(op.source2);
          o << ')';
          return Status();
      }

```

```

13b  <declarations 4a>+≡
      Status visit(UnaryOp &);
      Status visit(BinaryOp &);

```

## 5.4 Literal

```

13c  <declarations 4a>+≡
      Status visit(Literal &);

```

```

13d  <definitions 3a>+≡
      Status Printer::visit(Literal &l)
      {
          assert(l.type);
          if ( l.type->name == "string" ) {
              o << '\\';
              for ( string::iterator i = l.value.begin() ; i != l.value.end() ; i++ ) {
                  if (*i == '\\') o << '\\';
                  o << *i;
              }
              o << '\\';
          } else {
              o << l.value;
          }
          return Status();
      }

```

## 5.5 Function Call

Normal function calls are straightforward. Builtin functions are special: they are actually arithmetic or logical operators and therefore printed with an inline notation.

```

14a  <declarations 4a>+≡
      Status visit(FunctionCall &);

14b  <definitions 3a>+≡
      Status Printer::visit(FunctionCall &c)
      {
        assert(c.callee);
        if (dynamic_cast<BuiltinFunctionSymbol*>(c.callee)) {
          o << '(';
          switch (c.arguments.size()) {
            case 1:
              if (c.callee->name == "not") {
                o << '!';
              } else {
                o << c.callee->name << ' ';
              }
              printExpr(c.arguments.front());
              break;
            case 2:
              printExpr(c.arguments.front());
              if ( c.callee->name == "and" ) o << " && ";
              else if (c.callee->name == "or" ) o << " || ";
              else if (c.callee->name == "=" ) o << " == ";
              else if (c.callee->name == "<>" ) o << " != ";
              else o << ' ' << c.callee->name << ' ';
              printExpr(c.arguments[1]);
              break;
            default:
              // Not one or two arguments.  What function is this?
              assert(0);
              break;
          }
          o << ')';
        } else {
          o << c.callee->name << '(';
          for ( vector<Expression*>::iterator i = c.arguments.begin() ;
                i != c.arguments.end() ; i++ ) {
            printExpr(*i);
            if ( i != (c.arguments.end() - 1) ) o << ", ";
          }
          o << ')';
        }
        return Status();
      }

```

## 5.6 Procedure Call

```

15a  <declarations 4a>+≡
      Status visit(ProcedureCall &);

15b  <definitions 3a>+≡
      Status Printer::visit(ProcedureCall &c)
      {
        assert(c.procedure);
        o << c.procedure->name << '(';
        bool needComma = false;
        for ( vector<VariableSymbol*>::iterator i = c.reference_args.begin() ;
              i != c.reference_args.end() ; i++ ) {
          assert(*i);
          if (needComma) o << ", ";
          o << '&' << (*i)->name;
          needComma = true;
        }
        for ( vector<Expression*>::iterator i = c.value_args.begin() ;
              i != c.value_args.end() ; i++ ) {
          if (needComma) o << ", ";
          printExpr(*i);
          needComma = true;
        }
        o << ")";
        return Status();
      }

```

## 6 Overall declarations

This decides whether a `#include "basename.h"` is needed and prints it. `printDeclarations` calls this, so there should be no need otherwise.

```

15c  <declarations 4a>+≡
      virtual void printInclude(string);

```

```

16a  <definitions 3a>+≡
      void Printer::printInclude(string basename)
      {

          // Decide whether to #include "basename.h"
          // If there are any procedures, tasks, user-defined types, functions
          // or undefined constants, include it.
          bool needInclude = (m.procedures->size() != 0) || (m.tasks->size() != 0);

          if ( !needInclude )
              for ( SymbolTable::const_iterator i = m.types->begin() ;
                  i != m.types->end() ; i++ )
                  if ( dynamic_cast<BuiltinTypeSymbol*>(*i) == NULL ) {
                      needInclude = true;
                      break;
                  }

          if ( !needInclude )
              for ( SymbolTable::const_iterator i = m.constants->begin() ;
                  i != m.constants->end() ; i++ )
                  if ( dynamic_cast<BuiltinConstantSymbol*>(*i) == NULL ) {
                      ConstantSymbol *cs = dynamic_cast<ConstantSymbol*>(*i);
                      assert(cs);
                      if (cs->initializer == NULL) {
                          needInclude = true;
                          break;
                      }
                  }

          if ( !needInclude )
              for ( SymbolTable::const_iterator i = m.functions->begin() ;
                  i != m.functions->end() ; i++ )
                  if ( dynamic_cast<BuiltinFunctionSymbol*>(*i) == NULL ) {
                      needInclude = true;
                      break;
                  }

          if (needInclude)
              o << "#include \"" << basename << ".h\"\n";
      }

```

## 7 Declarations for variables, functions, procedures, etc.

```

16b  <declarations 4a>+≡
      virtual void printDeclarations(string);

```



```

17  <definitions 3a>+≡
    void Printer::printDeclarations(string basename)
    {

        // Although external types need no declarations, their names
        // are registered to check for later collisions

        o <<
            "#ifndef STRLEN"           "\n"
            "# define STRLEN 81"      "\n"
            "#endif"                  "\n"
            "#define _true 1"         "\n"
            "#define _false 0"       "\n"
            "typedef unsigned char boolean;" "\n"
            "typedef int integer;"     "\n"
            "typedef char* string;"    "\n"
            ;

        printInclude(basename);

        BuiltinConstantSymbol *truec =
            dynamic_cast<BuiltinConstantSymbol*>(m.constants->get(string("true")));
        assert(truec);
        variableVar[truec] = uniqueID("_true");
        BuiltinConstantSymbol *falsec =
            dynamic_cast<BuiltinConstantSymbol*>(m.constants->get(string("false")));
        assert(falsec);
        variableVar[falsec] = uniqueID("_false");

        identifiers.insert("STRLEN");

        // Verify all external type names are OK

        assert(m.types);
        for ( SymbolTable::const_iterator i = m.types->begin() ;
              i != m.types->end() ; i++ ) {
            TypeSymbol *s = dynamic_cast<TypeSymbol*>(*i);
            assert(s);
            if (contains(identifiers, s->name))
                throw IR::Error("Name of external type \"" + s->name +
                                "\"" + " already in use");
            uniqueID(s->name);
        }

        // Print input function declarations

        assert(m.signals);
        for ( SymbolTable::const_iterator i = m.signals->begin() ;
              i != m.signals->end() ; i++ ) {
            SignalSymbol *s = dynamic_cast<SignalSymbol*>(*i);

```

```

assert(s);
if (s->name != "tick" &&
    ( s->kind == SignalSymbol::Input ||
      s->kind == SignalSymbol::Inputoutput)) {
    assert(m.symbol);
    o << "void " << m.symbol->name << "_I_" << s->name << "(";
    if (s->type) {
        o << s->type->name;
    } else {
        o << "void";
    }
    o << ");\n";
}
}

// Print declarations for the tick and reset functions

o <<
"int " << m.symbol->name << "(void);"      "\n"
"int " << m.symbol->name << "_reset(void);" "\n";

// External declarations (constants, functions, procedures)

o << "#ifndef _NO_EXTERN_DEFINITIONS"      "\n";

// Uninitialized constants

o << "# ifndef _NO_CONSTANT_DEFINITIONS"  "\n";
assert(m.constants);
for ( SymbolTable::const_iterator i = m.constants->begin() ;
      i != m.constants->end() ; i++ ) {
    ConstantSymbol *s = dynamic_cast<ConstantSymbol*>(*i);
    assert(s);
    if (!s->initializer) {
        o << "#   ifndef _" << s->name << "_DEFINED\n";
        o << "#       ifndef " << s->name << "\n";
        assert(s->type);
        if (contains(identifiers, s->name))
            throw IR::Error("Name of constant \"" + s->name + "\" already in use");
        string var = uniqueID(s->name);
        variableVar[s] = var;
        o << "extern " << s->type->name << " " << var << ";\n";
        o << "#       endif\n";
        o << "#       endif\n";
    }
}
}
o << "# endif /* _NO_CONSTANT_DEFINITIONS */\n";

// Functions

```

```

o << "# ifndef _NO_FUNCTION_DEFINITIONS"    "\n";
assert(m.functions);
for ( SymbolTable::const_iterator i = m.functions->begin() ;
      i != m.functions->end() ; i++ ) {
  FunctionSymbol *s = dynamic_cast<FunctionSymbol*>(*i);
  assert(s);
  if (dynamic_cast<BuiltinFunctionSymbol*>(*i) == NULL ) {
    o << "#   ifndef _" << s->name << "_DEFINED\n";
    o << "#       ifndef " << s->name << "\n";
    if (contains(identifiers, s->name))
      throw IR::Error("Name of function \"" + s->name + "\" already in use");
    uniqueID(s->name);
    assert(s->result);
    o << "extern " << s->result->name << " " << s->name << "(";
    if (s->arguments.empty()) {
      o << "void";
    } else {
      for ( vector<TypeSymbol*>::const_iterator j = s->arguments.begin() ;
            j != s->arguments.end() ; j++ ) {
        assert(*j);
        o << (*j)->name;
        if ( j != s->arguments.end() - 1) o << ", ";
      }
    }
    o << ");\n";
    o << "#       endif\n";
    o << "#   endif\n";
  }
}
o << "# endif /* _NO_FUNCTION_DEFINITIONS */\n";

// Procedures

o << "# ifndef _NO_PROCEDURE_DEFINITIONS"    "\n";
assert(m.procedures);
for ( SymbolTable::const_iterator i = m.procedures->begin() ;
      i != m.procedures->end() ; i++ ) {
  ProcedureSymbol *s = dynamic_cast<ProcedureSymbol*>(*i);
  assert(s);
  o << "#   ifndef _" << s->name << "_DEFINED\n";
  o << "#       ifndef " << s->name << "\n";
  if (contains(identifiers, s->name))
    throw IR::Error("Name of procedure \"" + s->name + "\" already in use");
  uniqueID(s->name);
  o << "extern void " << s->name << "(";
  for ( vector<TypeSymbol*>::const_iterator j =
        s->reference_arguments.begin() ;
        j != s->reference_arguments.end() ; j++ ) {
    assert(*j);
    o << (*j)->name << "*";
  }
}

```

```

        if ( j != s->reference_arguments.end() - 1 ||
            !s->value_arguments.empty() )
            o << ", ";
    }
    for ( vector<TypeSymbol*>::const_iterator j = s->value_arguments.begin() ;
          j != s->value_arguments.end() ; j++ ) {
        assert(*j);
        o << (*j)->name;
        if ( j != s->value_arguments.end() - 1 ) o << ", ";
    }
    o << ");\n";
    o << "#      endif\n";
    o << "#      endif\n";
}
o << "#      endif /* _NO_PROCEDURE_DEFINITIONS */\n";

o << "#endif /* _NO_EXTERN_DEFINITIONS */\n\n";

// Initialized Constants

for ( SymbolTable::const_iterator i = m.constants->begin() ;
      i != m.constants->end() ; i++ ) {
    ConstantSymbol *s = dynamic_cast<ConstantSymbol*>(*i);
    assert(s);
    if (s->initializer && dynamic_cast<BuiltinConstantSymbol*>(*i) == NULL) {
        assert(s->type);
        if (contains(identifiers, s->name))
            throw IR::Error("Name of constant \"" + s->name + "\" already in use");
        string var = uniqueID(s->name);
        variableVar[s] = var;
        o << "static " << s->type->name << " " << s->name << " = ";
        printExpr(s->initializer);
        o << ";\n";
    }
}

// Variables for signal declarations

#ifdef USE_STRUCTS_FOR_SIGNALS
assert(m.signals);

// Define a struct holding all boolean presence variables

o << "static struct {\n";
unsigned int n_signals = 0;
for ( SymbolTable::const_iterator i = m.signals->begin() ;
      i != m.signals->end() ; i++ ) {
    SignalSymbol *s = dynamic_cast<SignalSymbol*>(*i);
    assert(s);
    if (s->name != "tick") {

```

```

        // All signals, sensors included, have presence variables
        string var = uniqueID(s->name);
        o << " unsigned int " << var << " : 1;\n";
        if (do_threevalued)
            o << " unsigned int " << var << "_unknown : 1;\n";
        presenceVar[s] = string("_s.") + var;
        ++n_signals;
    }
}
o << "} _s = { ";
for (unsigned int i = 0 ; i < n_signals ; i++) {
    o << " 0";
    if ( i < n_signals - 1 ) o << ", ";
}
o << " };\n";

// Define value variables for each valued signal

for ( SymbolTable::const_iterator i = m.signals->begin() ;
      i != m.signals->end() ; i++ ) {
    SignalSymbol *s = dynamic_cast<SignalSymbol*>(*i);
    assert(s);
    if (s->name == "tick") {
        // "tick" is a special built-in signal that is always present
        string var = uniqueID(s->name);
        presenceVar[s] = var;
        o << "#define " << var << " 1\n";
    }
    if (s->type) {
        // Has a type: need a value variable
        if (s->reincarnation) {
            // This is a reincarnation of an earlier signal: use its value variable
            // std::cerr << "Found reincarnation " << s->name << " of " << s->reincarnation->name << std::endl;
            assert(valueVar.find(s->reincarnation) != valueVar.end());
            valueVar[s] = valueVar[s->reincarnation];
        } else {
            string var = uniqueID(s->name + "_v");
            valueVar[s] = var;
            o << "static ";
            if (s->type->name == "string")
                o << "char " << var << "[STRLEN]";
            else
                o << s->type->name << " " << var;
            o << ";\n";
        }
    }
}
}
}

#else

```

```

assert(m.signals);
for ( SymbolTable::const_iterator i = m.signals->begin() ;
      i != m.signals->end() ; i++ ) {
    SignalSymbol *s = dynamic_cast<SignalSymbol*>(*i);
    assert(s);
    if (s->name == "tick") {
        // "tick" is a special built-in signal that is always present
        string var = uniqueID(s->name);
        presenceVar[s] = var;
        o << "#define " << var << " 1\n";
    } else {
        // All signals, sensors included, have presence variables
        string var = uniqueID(s->name);
        presenceVar[s] = var;
        o << "static boolean " << var << " = _false;\n";
        if (do_threevalued)
            o << "static boolean " << var << "_unknown = _false;\n";
    }
    if (s->type) {
        // Has a type: need a value variable
        if (s->reincarnation) {
            // This is a reincarnation of an earlier signal: use its value variable
            assert(valueVar.find(s->reincarnation) != valueVar.end());
            valueVar[s] = valueVar[s->reincarnation];
        } else {
            string var = uniqueID(s->name + "_v");
            valueVar[s] = var;
            o << "static ";
            if (s->type->name == "string")
                o << "char " << var << "[STRLEN]";
            else
                o << s->type->name << " " << var;
            o << ";\n";
        }
    }
}

#endif

// Variable declarations

assert(m.variables);
for ( SymbolTable::const_iterator i = m.variables->begin() ;
      i != m.variables->end() ; i++ ) {
    VariableSymbol *s = dynamic_cast<VariableSymbol*>(*i);
    assert(s);
    string var = uniqueID(s->name);
    variableVar[s] = var;
    o << "static ";
    if ( s->type->name == "string" )

```

```

        o << "char " << var << "[STRLEN]";
    else
        o << s->type->name << ' ' << var;
    if ( s->initializer ) {
        o << " = ";
        printExpr(s->initializer);
    }
    o << ";\n";
}

// State variable declarations

#ifdef USE_STRUCTS_FOR_STATES

o << "static struct {\n";
for ( STmap::const_iterator i = stmap.begin() ; i != stmap.end() ; i++ ) {
    STexcl *e = dynamic_cast<STexcl*>((*i).first);
    if (e) {
        char buf[15];
        sprintf(buf, "%d", stmap[e]);
        stateVar[e] = string("_state.") + string(buf);
        unsigned int bits = 1;
        while ( (1 << bits) < e->children.size() ) ++bits;
        o << " unsigned int " << buf << " : " << bits << ";\n";
    }
}
o << "} _state = { ";
bool needComma = false;
for ( STmap::const_iterator i = stmap.begin() ; i != stmap.end() ; i++ ) {
    STexcl *e = dynamic_cast<STexcl*>((*i).first);
    if (e) {
        // Initialization of states
        if (needComma) o << ", ";
        o << (e->children.size() - 1);
        needComma = true;
    }
}
o << "};\n";

#else

for ( STmap::const_iterator i = stmap.begin() ; i != stmap.end() ; i++ ) {
    STexcl *e = dynamic_cast<STexcl*>((*i).first);
    if (e) {
        char buf[15];
        sprintf(buf, "_state_%d", stmap[e]);
        string var = uniqueID(buf);
        stateVar[e] = var;
        o << "static unsigned char " << var;
        // Initialization of state of selection-tree root:

```

```

        // state = highest-numbered child
        if ( e == g->selection_tree )
            o << " = " << (e->children.size() - 1);
        o << ";\n";
    }
}
#endif

// Termination level variable declarations

for ( CFGmap::const_iterator i = cfgmap.begin() ;
      i != cfgmap.end() ; i++ ) {
    Sync *s = dynamic_cast<Sync*>((*i).first);
    if (s) {

        // Count the number of non-zero successors

        unsigned int successors = 0;
        for ( vector<GRCNode*>::iterator j = s->successors.begin() ;
              j != s->successors.end() ; j++ )
            if (*j) ++successors;
        if (successors > 1) {
            // If there is more than one non-NULL successor, generate a variable
            char buf[15];
            sprintf(buf, "_term_%d", cfgmap[s]);
            string var = uniqueID(buf);
            terminationVar[s] = var;
            o << "static int " << var << ";\n";
        }
    }
}

// Counter declarations

for ( vector<Counter*>::const_iterator i = m.counters.begin() ;
      i != m.counters.end() ; i++ ) {
    char buf[15];
    sprintf(buf, "_counter_%d", i-m.counters.begin() );
    string var = uniqueID(buf);
    counterVar[*i] = var;
    o << "static int " << var << ";\n";
}

#ifdef PRINT_OUTPUT_FUNCTION_DECLARATIONS
// Output function declarations

assert(m.signals);
for ( SymbolTable::const_iterator i = m.signals->begin() ;
      i != m.signals->end() ; i++ ) {

```



```

SignalSymbol *s = dynamic_cast<SignalSymbol*>(*i);
assert(s);
if (s->kind == SignalSymbol::Output ||
    s->kind == SignalSymbol::Inputoutput) {
    string name = m.symbol->name + "_0_" + s->name;
    o << "#ifndef " << name << "\n"
        "extern void " << name << "(";
    if (s->type) o << s->type->name;
    else o << "void";
    o << ");\n"
        "#endif\n";
}
}
#endif
}

```

## 8 Output Functions

Generate code that check the signal presence variables and call output functions as appropriate.

```

25a <declarations 4a>+≡
    virtual void outputFunctions();

25b <definitions 3a>+≡
    void Printer::outputFunctions()
    {
        assert(m.signals);
        for ( SymbolTable::const_iterator i = m.signals->begin() ;
            i != m.signals->end() ; i++ ) {
            SignalSymbol *s = dynamic_cast<SignalSymbol*>(*i);
            assert(s);
            if (s->kind == SignalSymbol::Output ||
                s->kind == SignalSymbol::Inputoutput) {
                assert(contains(presenceVar, s));
                o << " if (" << presenceVar[s] << ") { ";
                o << m.symbol->name << "_0_" << s->name << "(";
                if (s->type) {
                    assert(contains(valueVar, s));
                    o << valueVar[s];
                }
                o << "); " << presenceVar[s] << " = 0; }\n";
            }
        }
    }
}

```

## 9 Reset inputs

Generate code that resets all the inputs and sensor presence variables.

```

26a <declarations 4a>+≡
    virtual void resetInputs();

26b <definitions 3a>+≡
    void Printer::resetInputs()
    {
        assert(m.signals);
        for ( SymbolTable::const_iterator i = m.signals->begin() ;
              i != m.signals->end() ; i++ ) {
            SignalSymbol *s = dynamic_cast<SignalSymbol*>>(*i);
            assert(s);
            if (s->name != "tick" &&
                ( s->kind == SignalSymbol::Input ||
                  s->kind == SignalSymbol::Inputoutput ||
                  s->kind == SignalSymbol::Sensor )) {
                o << " ";
                assert(contains(presenceVar, s));
                o << presenceVar[s] << " = 0;\n";
            }
        }
    }

```

## 10 I/O function printers

FIXME: This does not support “combine” functions.

```

26c <declarations 4a>+≡
    virtual void ioDefinitions();

```

```

27  <definitions 3a>+≡
    void Printer::ioDefinitions()
    {
        // Print input signal function definitions

        assert(m.signals);
        for ( SymbolTable::const_iterator i = m.signals->begin() ;
              i != m.signals->end() ; i++ ) {
            SignalSymbol *s = dynamic_cast<SignalSymbol*>(*i);
            assert(s);

            if (s->name != "tick" &&
                ( s->kind == SignalSymbol::Input ||
                  s->kind == SignalSymbol::Inputoutput)) {
                assert(contains(presenceVar, s));
                assert(m.symbol);
                o << "void " << m.symbol->name << "_I_" << s->name << "(";
                if (s->type) {
                    o << s->type->name << " _v";
                } else {
                    o << "void";
                }
                o << ") {\n"
                  " " << presenceVar[s] << " = 1;\n";
                if (s->type) {
                    assert(contains(valueVar, s));
                    if (s->type->name == "string") {
                        o << " strcpy(" << valueVar[s] << ", _v);\n";
                    } else {
                        // FIXME: This doesn't work with combine
                        o << " " << valueVar[s] << " = _v;\n";
                    }
                }
                o << "}\n";
            }
        }
    }
}

```

## 11 Structured Code Generation

This prints C code for an acyclic CFG using the algorithm described in Stephen A. Edwards, An Esterel Compiler for Control-Dominated Systems, *IEEE Transactions on CAD*, 21(2), February 2002. It first constructs a reverse immediate dominator tree to determine where to terminate block-structured statements such as if-else and switch. Then it uses a recursive procedure to construct a simple abstract syntax tree for the generated code. Finally, this tree is walked to generate the final code.

The node passed the `printStructuredCode` should be the *exit* node for the CFG to be printed, i.e., it should have no successors.

```

28a <declarations 4a>+≡
    void printStructuredCode(GRCNode *, unsigned int = 0);

28b <definitions 3a>+≡
    void Printer::printStructuredCode(GRCNode *exit_node, unsigned int indent)
    {
        assert(exit_node);
        assert(exit_node->successors.size() == 0);

        // Number the nodes in a depth-first order

        nodes.clear();
        nodeNumber.clear();

        dfsVisit(exit_node);

        <compute reverse dominators 30>

        GRCCode *entry_node = nodes.front();

        statementFor.clear();
        CStatement *root = synthesize(entry_node, nodes.back(), false);

        CStatement::printer = this;
        for ( ; root ; root = root->next ) {
            // std::cerr << "Printing node " << cfmmap[root->node] << "\n";
            root->print(indent);
        }

        delete root;
    }

```

## 11.1 DFS node numbering

This method computes the postorder numbering of nodes required by the dominator algorithm.

```

29a  <declarations 4a>+≡
      void dfsVisit(GRCNode*);

29b  <definitions 3a>+≡
      void Printer::dfsVisit(GRCNode *n)
      {
        if (!n || nodeNumber.find(n) != nodeNumber.end()) return;
        nodeNumber[n] = -1; // Mark as being visited, but do not know number yet

        for (vector<GRCNode*>::const_iterator i = n->predecessors.begin() ;
             i != n->predecessors.end() ; i++)
          dfsVisit(*i);

        nodeNumber[n] = nodes.size();
        nodes.push_back(n);

        // std::cerr << "Assigned node " << cfgmap[n] << " = " << nodeNumber[n] << '\n';
      }

```

## 11.2 Compute Reverse Dominators

This uses the iterative dominator computation algorithm from Keith Cooper, Timothy Harvey, and Ken Kennedy, *A Simple, Fast Dominance Algorithm*, submitted to Software—Practice and Experience.

```

30  <compute reverse dominators 30>≡
    ridom.clear();

    // Compute immediate dominators on the reverse graph

    ridom[exit_node] = exit_node;
    bool changed;
    do {
        changed = false;
        for ( vector<GRCNode*>::reverse_iterator b = nodes.rbegin() + 1;
              b != nodes.rend() ; b++ ) {
            GRCNode *new_idom = NULL;
            for ( vector<GRCNode*>::iterator p = (*b)->successors.begin() ;
                  p != (*b)->successors.end() ; p++ ) {
                if ( ridom.find(*p) != ridom.end() ) {
                    if ( new_idom == NULL )
                        new_idom = *p;
                    else {
                        // Intersect
                        GRCNode *b1 = *p;
                        GRCNode *b2 = new_idom;
                        while (b1 != b2) {
                            while (nodeNumber[b1] < nodeNumber[b2]) b1 = ridom[b1];
                            while (nodeNumber[b2] < nodeNumber[b1]) b2 = ridom[b2];
                        }
                        new_idom = b1;
                    }
                }
            }
            if ( ridom[*b] != new_idom ) {
                ridom[*b] = new_idom;
                // std::cerr << "idom of " << cfgmap[*b] << " is " << cfgmap[new_idom] << '\n';
                changed = true;
            }
        }
    } while (changed);

```

### 11.3 C Statements

```

31  <c statement classes 31>≡
    class Printer;

    struct CStatement {
        static Printer *printer;

        GRNode *node;
        CStatement *next;
        string label;

        CStatement(GRNode *node) : node(node), next(0) {}
        virtual ~CStatement() { delete next; }
        virtual void print(unsigned int = 0);
        void indent(unsigned int);
        void begin(unsigned int);
    };

    struct CIfElse : CStatement {
        CStatement *thenSt;
        CStatement *elseSt;
        CIfElse(GRNode *node, CStatement *thenSt, CStatement *elseSt)
            : CStatement(node), thenSt(thenSt), elseSt(elseSt) {}
        virtual ~CIfElse() { delete thenSt; delete elseSt; }
        void print(unsigned int = 0);
    };

    struct CGoto : CStatement {
        string label;
        CGoto(string label) : CStatement(NULL), label(label) {}
        void print(unsigned int = 0);
    };

    struct CBreak : CStatement {
        CBreak() : CStatement(NULL) {}
        void print(unsigned int = 0);
    };

    struct CSwitch : CStatement {
        CStatement *body;
        CSwitch(GRNode *node, CStatement *body) : CStatement(node), body(body) {}
        virtual ~CSwitch() { delete body; }
        void print(unsigned int = 0);
    };

    struct CCase : CStatement {
        int label;
        CStatement *body;
        CCase(int label, CStatement *body) : CStatement(NULL), label(label), body(body) {}

```

```

        void print(unsigned int = 0);
    };

32a  <definitions 3a>+≡
        Printer *CStatement::printer = 0;

32b  <definitions 3a>+≡
        void CStatement::indent(unsigned int n)
        {
            for (unsigned int i = 0 ; i < n ; i++) printer->o << " ";
        }

32c  <definitions 3a>+≡
        void CStatement::begin(unsigned int i)
        {
            if (!label.empty()) {
                indent(i > 0 ? i - 1 : i);
                printer->o << label << "\n";
            }
            indent(i);
        }

32d  <definitions 3a>+≡
        void CStatement::print(unsigned int i)
        {
            begin(i);
            printer->printExpr(node);
            printer->o << "\n";
        }

32e  <definitions 3a>+≡
        void CIfElse::print(unsigned int i)
        {
            begin(i);
            printer->o << "if (";
            printer->printExpr(node);
            printer->o << ") {\n";
            for ( CStatement *st = thenSt ; st ; st = st->next ) st->print(i+1);
            indent(i);
            printer->o << "}";
            if ( elseSt ) {
                printer->o << " else {\n";
                for ( CStatement *st = elseSt ; st ; st = st->next ) st->print(i+1);
                indent(i);
                printer->o << "}\n";
            } else {
                printer->o << "\n";
            }
        }
    }

```



- 33a  $\langle$ definitions 3a $\rangle$ + $\equiv$   

```
void CGoto::print(unsigned int i)
{
    begin(i);
    printer->o << "goto " << label << ";\n";
}
```
- 33b  $\langle$ definitions 3a $\rangle$ + $\equiv$   

```
void CBreak::print(unsigned int i)
{
    begin(i);
    printer->o << "break;\n";
}
```
- 33c  $\langle$ definitions 3a $\rangle$ + $\equiv$   

```
void CSwitch::print(unsigned int i)
{
    begin(i);
    printer->o << "switch (";
    printer->printExpr(node);
    printer->o << ") {\n";
    for ( CStatement *st = body ; st ; st = st->next ) st->print(i+1);
    indent(i);
    printer->o << "default: break;\n";
    indent(i);
    printer->o << "}\n";
}
```
- 33d  $\langle$ definitions 3a $\rangle$ + $\equiv$   

```
void CCase::print(unsigned int i)
{
    indent(i > 0 ? i - 1 : 0);
    printer->o << "case " << label << ":\n";
    assert(body);
    for ( CStatement *st = body ; st ; st = st->next ) st->print(i);
}
```

## 11.4 Statement synthesis

- 33e  $\langle$ declarations 4a $\rangle$ + $\equiv$   

```
CStatement *synthesize(GRCNode*, GRNode*, bool);
```

```

34  <definitions 3a>+≡
    CStatement *Printer::synthesize(GRCNode *node, GRCNode *final, bool needBreak)
    {
        assert(node);
        assert(final);

        //std::cerr << "/* initial synthesize(" << cfgmap[node] << ", " << cfgmap[final] << ", " <<

    #if 0
        std::cerr << "successors: ";
        for ( vector<GRCNode*>::const_iterator i = node->successors.begin() ;
              i != node->successors.end() ; i++ ) {
            if (*i) std::cerr << cfgmap[*i] << ' ';
            else std::cerr << "NULL ";
        }
        std::cerr << std::endl;
    #endif

        if ( node == final )
            return needBreak ? new CBreak() : 0;

        if ( statementFor.find(node) != statementFor.end() ) {
            CStatement *target = statementFor[node];
            if (target->label.empty()) {
                char buf[20];
                // sprintf(buf, "L%d", nextLabel++);
                assert(cfgmap.find(node) != cfgmap.end());
                sprintf(buf, "N%d", cfgmap[node]);
                target->label = buf;
            }
            return new CGoto(target->label);
        }

        assert(ridom.find(node) != ridom.end());

        GRCNode *next =
            (node->successors.size() > 1) ? ridom[node] : node->successors.front();

        CStatement *nextStatement = next ? synthesize(next, final, needBreak) : 0;

        // std::cerr << "/* continue synthesize(" << cfgmap[node] << ", " << cfgmap[final] << ", " <<

        CStatement *result = NULL;

        switch (node->successors.size()) {
        case 0:
        case 1:
            // std::cerr << "simple statement\n";
            result = new CStatement(node);
            break;

```

```

case 2:
{
    if (node->successors.front() && node->successors.back()) {
        //std::cerr << "if-then-else statement\n";
        CStatement *elsePart =
            synthesize(node->successors.front(), next, false);
        CStatement *thenPart =
            synthesize(node->successors.back(), next, false);
        result = new CIfElse(node, thenPart, elsePart);
    } else {
        // std::cerr << "Identified a node with two successors that became a simple statement" << std::endl;
        result = new CStatement(new Nop());
    }
}
break;
default:
// Three or more successors: a switch statement
{
    // std::cerr << "switch statement\n";
    unsigned int nonzero_successors = 0;
    for ( vector<GRCCNode*>::reverse_iterator i = node->successors.rbegin() ;
          i != node->successors.rend() ; i++ )
        if (*i) ++nonzero_successors;
    if (nonzero_successors > 1) {
        CStatement *body = NULL;
        bool useSyncNumbering = dynamic_cast<Sync*>(node) != NULL;
        for ( vector<GRCCNode*>::reverse_iterator i =
              node->successors.rbegin() ;
              i != node->successors.rend() ; i++ )
            if (*i) {
                CStatement *caseBody = synthesize(*i, next, true);
                int caseLabel = node->successors.rend() - i - 1;
                if (useSyncNumbering) caseLabel = (1 << caseLabel) - 1;
                CStatement *thisCase = new CCase(caseLabel, caseBody);
                thisCase->next = body;
                body = thisCase;
            }
        result = new CSwitch(node, body);
    } else {
        // std::cerr << "Identified a node with multiple successors that became a simple statement" << std::endl;
        result = new CStatement(new Nop());
    }
}
break;
}

assert(result);
assert(result->next == NULL);
result->next = nextStatement;

```

```
// std::cerr << "done with " << cfgmap[node] << "\n";
assert(result);
statementFor[node] = result;
if (labelFor.find(node) != labelFor.end())
    result->label = labelFor[node];
return result;
}
```

## 12 Utilities

```
36 <utilities 36>≡
    template <class T> bool contains(set<T> &s, T o) {
        return s.find(o) != s.end();
    }

    template <class T, class U> bool contains(map<T, U> &m, T o) {
        return m.find(o) != m.end();
    }
```

## 13 Top-Level Files

```
37a  <CPrinter.hpp 37a>≡
      #ifndef _CPRINTER_HPP
      # define _CPRINTER_HPP

      # define USE_STRUCTS_FOR_SIGNALS
      /* # define USE_STRUCTS_FOR_STATES */

      # include "AST.hpp"
      # include <iostream>
      # include <cassert>
      # include <set>
      # include <vector>
      # include <map>

      namespace CPrinter {
        using namespace AST;
        using std::set;
        using std::vector;
        using std::map;

        <utilities 36>

        <c statement classes 31>

        <printer class 2>
      }

      #endif

37b  <CPrinter.cpp 37b>≡
      #include "CPrinter.hpp"
      #include <stdio.h>

      namespace CPrinter {
        <definitions 3a>
      }
```