Anatomy of a Small Compiler

COMS W4115

Prof. Stephen A. Edwards
Spring 2003
Columbia University
Department of Computer Science

CEC

CEC is the Columbia Esterel Compiler that my group is currently developing.

You can find the source code (well-documented C++) off the “software” link on my homepage.

Compiles the Esterel language into hardware and software.

A concurrent language: uses a concurrent control-flow graph as an intermediate representation.

Esterel Syntax

Standard free-form style:

```
module test_present2:
input A;
output B, C;

present A then
  emit B
else
  emit C
end present

end module
```

The Scanner

Options

```
class EsterelLexer extends Lexer;

options {
  // Lookahead to distinguish, e.g., : and :=
  k = 2;
  // Handle all 8-bit characters
  charVocabulary = '\3'..'\377';
  // Export these token types for tree walkers
  exportVocab = Esterel;
  // Disable checking every rule against keywords
  testLiterals = false;
}
```

Punctuation and Identifiers

```
PERIOD :   '.' ;
POUND :    '#' ;
PLUS :     '+' ;
DASH :     '-' ;
SLASH :    '/' ;
STAR :     '*' ;
PARALLEL : "||" ;
/* etc. */

ID options { testLiterals = true; }
  : ('a'..'z' | 'A'..'Z')
    ('a'..'z' | 'A'..'Z' | '_' | '0'..'9')*
  ;
```

C-style numeric constants

```
Number
  : ('0'..'9')+
      ( '.' ('0'..'9')* (Exponent)?
        ( ('f'|'F') { $setType(FloatConst); }
        | /* empty */
          { $setType(DoubleConst); }
        )
      | /* empty */ { $setType(Integer); }
      )
  ;
```

C-style numeric constants contd.

```
FractionalNumber
  : '.' ('0'..'9')+ (Exponent)?
        ( ('f'|'F') { $setType(FloatConst); }
        | /* empty */
          { $setType(DoubleConst); }
        )
  ;

protected
Exponent
  : ('e'|'E') ('+'|'-')? ('0'..'9')+
  ;
```

Strings, whitespace, newlines

```
StringConstant
  : '"'! ( ~('"' | '\n') | ('"'! '"'))* '"'!
  ;

Whitespace
  : (' ' | '\t' | '\f')+
    { $setType(antlr::Token::SKIP); }
  ;

Newline
  : ( '\n' | "\r\n" | '\r' )
    { $setType(antlr::Token::SKIP);
      newline(); }
  ;
```

The Parser

Options

```
class EsterelParser extends Parser;
options {
  // Lookahead
  k = 2;
  // Construct an AST during parsing
  buildAST = true;
  // Export these token types for the tree walker
  exportVocab = Esterel;
  // Create AST nodes with line numbers
  ASTLabelType = "RefLineAST";
  // Don't automatically catch every exception
  defaultErrorHandler = false;
}
```

Tokens

Extra token types; don't correspond to keywords. Used to build additional structure into the AST.

```
tokens {
 SIGS;
 VARS;
 TYPES;
 DECLS;
 TRAPS;
 SEQUENCE;
 ARGS;
 /* etc. */
}
```

File and module

```
file
  : (module)+ EOF!
  ;

module
  : "module"^ moduleIdentifier COLON!
      declarations
      statement
    (
      "end"! "module"!
    | PERIOD!  // Deprecated syntax
    )
  ;
```

Declarations

```
declarations
  : (interfaceDecls)*
    { #declarations = #([DECLS, "decls"],
                        #declarations); }
  ;
interfaceDecls :
    typeDecls
  | constantDecls
  | functionDecls
  | procedureDecls
  | taskDecls
  | interfacesignalDecls
  | sensorDecls
  | relationDecls
  ;
```

Various Declarations

```
typeDecls
  : "type"^ typeIdentifier
    (COMMA! typeIdentifier)* SEMICOLON!
  ;

constantDecls
  : "constant"^ constantDecl
    (COMMA! constantDecl )* SEMICOLON!
  ;
```

Expressions

```
expression
  : orexpr
  ;
orexpr
  : andexpr ("or"^ andexpr)*
  ;
andexpr
  : notexpr ("and"^ notexpr)*
  ;
notexpr
  : "not"^ cmpexpr
  | cmpexpr
  ;
```

Expressions

```
mulexpr
  : unaryexpr
    ( (STAR^ | SLASH^ | "mod"^) unaryexpr )*
  ;

unaryexpr
  : DASH^ unaryexpr
  | LPAREN! expression RPAREN!
  | QUESTION^ signalIdentifier
  | "pre"^
    LPAREN! QUESTION^ signalIdentifier RPAREN!
  | DQUESTION^ trapIdentifier
  | functionCall
  | constant
  ;
```

Statements in Parallel

```
statement
  : sequence (PARALLEL! sequence)*
    { if (#statement &&
          #statement->getNextSibling()) {
        #statement = #([PARALLEL, "||"],
                       #statement);
      }
    }
  ;
```

Statements in Sequence

```
sequence
  : atomicStatement
    (options {greedy=true;} :
      SEMICOLON! atomicStatement)*
    (SEMICOLON!)?
    { if (#sequence &&
          #sequence->getNextSibling()) {
        #sequence = #([SEQUENCE, ";"],
                      #sequence);
      }
    }
  ;
```

The Present (if) Statement

Two forms:

```
present S then         present
   nothing               case C do nothing
else                     case D
   nothing               else pause
end                    end present
```

```
present
  : "present"^
    (presentThenPart | (presentCase)+)
     (elsePart)? "end"! ("present"!)?
  ;
```

The Present (if) Statement

```
presentThenPart
  : presentEvent ("then"! statement)?
    { #presentThenPart = #([CASE,"case"],
                           presentThenPart); }
  ;
elsePart
  :  "else"^ statement
  ;
presentCase
  : "case"! presentEvent ("do"! statement)?
    { #presentCase = #([CASE,"case"],
                       presentCase); }
  ;
```

The AST Classes

My AST Classes

ANTLR, by default, builds its AST out of one type of object, an AST node with numeric type, a string, a first child, and a next sibling.

It has a facility for building heterogeneous ASTs (one class per token type), but I chose not to use it.

Instead, I created a new set of AST classes and translated the homogeneous AST into these classes during static semantics.

AST Classes

- Symbols (modules, signals, variables, functions)
  Name and usually a type
- Symbol table
  Holds symbols, points to a parent symbol table
- Expressions (literals, variables, operators)
  Each has a type
- Modules (like a function declaration)
  Has many symbol tables and a body
- Statement sequences and parallel groups
- Statements, one class per statement type

Example AST class

```
class Assignment : Statement {
  VariableSymbol *variable;
  Expression *value;
};
```

Example AST Classes

```
class CaseStatement : Statement {
  vector<PredicatedStatement *> cases;
  Statement *default;
};

class BodyStatement : Statement {
  Statement *body;
};

class PredicatedStatement : BodyStatement {
  Expression *predicate;
};
```

The Symbol Table Class

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

  SymbolTable() : parent(NULL) {}

  bool local_contains(const string) const;
  bool contains(const string) const;
  void enter(Symbol *);
  Symbol* get(const string);
};
```

SymbolTable contains tests

```
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;
}
```

SymbolTable::enter

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

SymbolTable::get

```
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);
}
```

Static Semantics

Static Semantics

Checks that every symbol is defined

Checks types (simple in Esterel)

Translates the ANTLR-generated AST into my own specialized version.

Written as a tree walker

The Tree Walker

```
class EsterelTreeParser extends TreeParser;

options {
  // Get the Esterel token types
  importVocab = Esterel;
  // Expect AST nodes with line numbers
  ASTLabelType = "RefLineAST";
}

file [Modules *ms, string filename]
    : { assert(ms); }
      ( module[ms] )+
    ;
```

The Module Rule

```
module [Modules* modules]
  : #( "module" moduleName:ID
{
  assert(modules);
  string name = moduleName->getText();
  if (modules->
      module_symbols.local_contains(name))
    throw LineError(moduleName,
                "Duplicate module " + name);
  ModuleSymbol *ms = new ModuleSymbol(name);
  Module *m = new Module(ms);
  ms->module = m;
  modules->add(m);
```

The notion of a Context

When you're translating, say, an expression, you need to know in which symbol table to look for symbols and other useful things.

I implemented a class called "Context" to hold this information.

Encountering a scope-generating statement creates a new context.

Translation routines pass the context to whatever they call.

Contexts are not part of the AST and are discarded after a scope closes.

Context

```
struct Context {
  Module *module;
  SymbolTable *variables;
  SymbolTable *traps;
  SymbolTable *signals;
  BuiltinTypeSymbol *boolean_type;
  BuiltinTypeSymbol *integer_type;
  BuiltinTypeSymbol *float_type;
  BuiltinTypeSymbol *double_type;
  BuiltinConstantSymbol *true_constant;
  BuiltinConstantSymbol *false_constant;
  Context(Module *m) :
    module(m), variables(m->constants),
    traps(0), signals(m->signals) {}
};
```

The Module Rule

```
Context c(m);

m->types->enter(c.boolean_type =
    new BuiltinTypeSymbol("boolean"));
m->constants->enter(c.false_constant =
  new BuiltinConstantSymbol("false", c.boolean_type, 0)
m->functions->enter(new BuiltinFunctionSymbol("and"));
/* ... */

VariableSymbol *vs =
   new VariableSymbol("tick", c.boolean_type, 0);
m->variables->enter(vs);
m->signals->enter(
  new BuiltinSignalSymbol("tick", 0,
                        "input", 0, vs, 0));
```

The Module Rule

```
   Statement *s; /* Local variable in module rule */
  }

 declarations[&c]
 s=statement[&c] { m->body = s; }
 ) /* matches #("module" ... */
 ;
```

Signal Declarations

```
input s1,
      s2 : boolean,
      s3 : combine integer with +,
      s8 := 3 : integer,
      s9 := 5 : combine integer with +;
```

Signal Declarations

```
signalDecl [Context *c, string direction,
            bool isGlobal]
  : #( SDECL signalName:ID
       {
         string name = signalName->getText();
         if (c->signals->local_contains(name))
           throw LineError(signalName,
                    "Redeclaration of " + name);
         Expression *e = 0;
       }
       ( #(COLEQUALS e=expr:expression[c]) )?
       { TypeSymbol *t = 0; FunctionSymbol *fs = 0; }
```

Signal Declarations

```
(t=typeToken:type[c]
  ( func:ID
    {
      string name = func->getText();
      if (!c->module->functions
            ->local_contains(name))
        throw LineError(func,
           "Undeclared function " + name);
      Symbol *sym = c->module->functions->get(name);
      fs = dynamic_cast<FunctionSymbol*>(sym);
      assert(fs);
    }
```

Signal Declarations

```
  | pcf:predefinedCombineFunction
    {
      string name = pcf->getText();
      assert(c->module->functions->contains(name));
      Symbol *sym = c->module->functions->get(name);
      fs = dynamic_cast<BuiltinFunctionSymbol*>(sym);
      assert(fs);
    }
  )?
)?
```

Signal Declarations

```
       {
          new_signal(c, name, t, direction, fs, e);
          if (e && (e->type != t))
            throw LineError(signalName,
              "initializer does not "
              "match type of signal");
       }
     )
  ;
```

Signal Expressions

```
sigExpression [Context *c] returns [Expression *e]
 : { Expression *e1, *e2; }
 ( #( "and" e1=sigExpression[c] e2=sigExpression[c] )
   { e = new BinaryOp(c->boolean_type,"and",e1,e2); }
 | sig:ID
   {
     string name = sig->getText();
     if (!c->signals->contains(name))
       throw LineError(sig,
                  "unrecognized signal " + name);
     SignalSymbol *ss = dynamic_cast<SignalSymbol*>(
                        c->signals->get(name));
     e = new LoadSignalExpression(ss);
   }
 );
```

Local Signal Statements

```
signal ls2,
       ls3 : boolean,
       ls4 := 3 + v1 : integer,
       ls5 := v3 or true :
          combine boolean with or in
   emit ls2;
   emit ls4(10);
   emit ls5(false)
end
```

Local Signal Statement

```
| #( "signal"
     {
       Signal *sig = new Signal();
       Context nc = *c;
       nc.signals = sig->symbols = new SymbolTable();
       sig->symbols->parent = c->signals;
     }
     #( SIGS ( signalDecl[&nc, "local", false] )+ )
     { Statement *s; }
     s=statement[&nc]
     {
       sig->body = s;
       st = sig;
     }
   )
```

Type checking expressions

```
expression [Context *c] returns [Expression *e]
  :
  {
    Expression *e1 = 0, *e2 = 0; // for safety
    e = 0; // for safety
  }
  (#( PLUS e1=expression[c] e2=expression[c] )
    { e = numeric_binop(#expression,
                        c, "+", e1, e2); }
  | #( STAR e1=expression[c] e2=expression[c] )
    { e = numeric_binop(#expression,
                        c, "*", e1, e2); }
```

Type checking expressions

```
static Expression*
numeric_binop(RefLineAST l, Context *c, string op,
              Expression *e1, Expression *e2)
{
  assert(c); assert(e1); assert(e2);

  if (e1->type != e2->type ||
      !(e1->type == c->integer_type ||
        e1->type == c->float_type ||
        e1->type == c->double_type ))
    throw LineError(l,
      "arguments of " + op + " must be numeric");
  return new BinaryOp(e1->type, op, e1, e2);
}
```

Dismantling

Dismantling

Many more complicated Esterel statement are equivalent to multiple simple statements, e.g.,

```
present                  if (p1) s1
  case p1 do s1          else if (p2) s2
  case p2 do s2          else s3
  else s3
end
```

Dismantling Case Statements

```
IfThenElse *dismantle_case(CaseStatement &c) {
  IfThenElse *result = 0; IfThenElse *lastif = 0;
  for (vector<PredicatedStatement*>::iterator i =
       c.cases.begin() ; i != c.cases.end() ; i++ ) {
    IfThenElse *thisif =
      new IfThenElse((*i)->predicate);
    thisif->then_part = transform((*i)->body);
    if (result) lastif->else_part = thisif;
    else result = thisif;
    lastif = thisif;
  }
  lastif->else_part = c.default_stmt;
  return transform(result); // Recurse
}
```

Some Statistics

File	Role	# lines
esterel.g	Parser/Scanner	850
staticsemantics.g	AST builder	1025
AST.nw	AST class source	1488
IR.nw	XML Serialization	827
Dismantle.nw	Dismantling	1571
AST.hpp*	AST classes	1828
AST.cpp*	AST classes	1525

* auto-generated