# Compiler Implementation in Standard ML

Daniel Phelps
daniel.phelps@uky.edu
University of Kentucky
Supervised by Dr. Raphael Finkel

June 15, 2015

## 1 Introduction

I designed and built a compiler in Standard ML of New Jersey (SML/NJ) for the CSX programming language. This language is defined and presented in Crafting a Compiler by Fischer et al, and given as a course project in CS541 Programming Languages [1, 2]. In CS541, we implemented a CSX compiler using Java, an imperative, object-oriented language. My personal interest in functional programming and languages led me to propose rewriting the compiler in Standard ML (SML), a functional language popular among compiler writers [3, 4]. The basic code for this project was derived from a small calculator program included with Andrew Appel's book "Modern Compiler Implementation in Standard ML" [5].

I was familiar with functional programming concepts, but I had never written a non-trivial program in a functional language. I believed SML to be a good choice for my project because it would require me to learn SML, a statically typed functional language with Hindley-Milner type inference, parametric polymorphism, and pattern matching.

We briefly discuss the SML environment and the compiler generation tools that come with it. Next, we discuss the implementation of each part of the compiler. Then, we cover the approach for testing the implementation.

### 1.1 SML/NJ Environment

SML/NJ is available for Unix, MacOS, Windows, and Cygwin. Packages are available for Ubuntu Linux via aptitude. The SML/NJ Language Processing Tools (LPT) are installed separately under the names `ml-yacc`, `ml-ulex`,

and `ml-lpt`. An interactive session is started with the `sml` command:

```
$ sudo apt-get install smlnj ml-lpt ml-ulex ml-yacc > /dev/null
$ sml
Standard ML of New Jersey v110.76 [built:  Mon Jul 7 23:25:08 2014]
- 1 + 1;
val it = 2 :  int
-
```

## 1.2 SML/NJ Compilation Manager

SML/NJ provides its own build system called the Compilation Manager (CM) [6]. The CM is invoked by running the `ml-build` command, and takes as arguments the heap image name, the entry point, and a build configuration file containing the root group. The root group is a list of external dependencies and all source files in the project to be built. The CM compares time stamps of files in the root group to the SML heap time stamp, and rebuilds the heap image when the code changes. The CM does not require the developer to explicitly define dependencies within the project. It decides when to invoke `ml-yacc` and `ml-ulex` according to the input filename extensions in the root group. The CM appends the architecture and operating system name to the heap image file for later reference. Building the heap image with `ml-build` and loading it with `sml` is straightforward:

```
$ ml-build sources.cm mystructure.main myprogram-img > /dev/null
$ sml @SMLload myprogram-img.x86-linux [arguments for myprogram]
```

## 1.3 SML Terminology

SML is equipped with unique constructs. We use the following terms and definitions throughout this paper:

- **structure** — a module containing definitions and declarations of functions and values
- **datatype** — a keyword for defining an abstract record type
- **reference** — a stateful value of a certain datatype
- **exception** — a value that can be raised and handled in an error state
- **function** — a mapping from values of some type to values of some type

- **type constructor** — a function returning a value of a certain datatype

SML and other functional languages encourage the developer to eschew side effects whenever possible. A change of state in a functional program is accomplished by the use of tail recursion or continuations; rather than modify a data structure with assignment, the developer creates a new structure and either recurs or continues in another context. SML provides a mechanism allowing for assignment, but it requires use of the `ref` datatype along with a special syntax for doing so. The `ref` is useful in situations where adding another argument to a set of mutually recursive functions would be time consuming or redundant. The `ref` is also helpful for tasks naturally implemented with assignment, such as memoization.

# 2 The CSX Language

CSX is an imperative, block-structured language. Its definition calls for `bool`, `char`, and `int` datatypes, and arrays of each. It supports recursion, and looping is available with a `while-do` construct. Input and output is available with `read()` and `print()`. A CSX program with recursion and looping might appear as follows. This program takes a positive integer as input and prints integers from that number down to zero:

```
class PrintToZero {
   void print_from(int i) {
      print(i, "\\n");
      if(i <= 0) {
          return;
      }
      print_from(i - 1);
   }
   void main() {
      int max = 0;
      while(max <= 0) {
         print("Enter a number greater than 0: ");
         read(max);
      }
      print_from(max);
   }
}
```

# 3 The CSX Lexical Analyzer

The lexer of a compiler segments the input string into tokens later used by a parser. It recognizes the finite set of terminal symbols found in the language definition such as delimiters, keywords, and identifiers. Writing a lexer by hand is possible but tedious. Instead, lexers are automatically generated based on a specification. A lexer specification for `ml-ulex` consists of three sections: user definitions, regular expression pattern definitions, and another containing the semantic actions for each token type [7].

The first section allows the developer to define arbitrary functions that can be used elsewhere in the specification. We define values allowing the lexer to track its current position within the input string. The position values accompany tokens in the lexer output stream.

## 3.1 Integer Literals

CSX places restrictions on the use of integer literals, and those rules are enforced by the lexical analyzer. Integers in CSX are 32-bit and signed. Any integer literal in a program not within the interval $[-2^{31} - 1, 2^{31}]$ results in a warning, and the input value becomes the maximum or minimum possible `int` value, whichever is closest to the literal input value.

The next section of the lexical specification names regular expressions that appear in the third section.

The third section contains the rules for tokenizing the input string. We include a regular expression for each token type and associate a semantic action with it. The semantic action updates the current position within the string based on the length of the token and returns a value containing the text and position within the file. The CSX language defines the following terminal symbols: identifiers, integer literals, character literals, string literals, conventional arithmetic symbols, logical operation symbols, delimiters such as brackets and parenthesis, and keywords. Our lexer is not case sensitive; "WhiLE" and "while" are equivalent tokens.

# 4 The CSX Abstract Syntax Tree (AST)

We use SML type constructors to represent syntactical elements of the CSX language. The recursive nature of the elements is elegantly expressed using the datatype construct in SML. SML's `and` keyword allows the developer to define mutually recursive datatypes. We show only a portion of the definition for the sake of brevity:

```
datatype csx_program = Program of (id * csx_member_decls)
     and csx_member_decls = MemberDecls of ( csx_decl list
                                           * csx_decl list)
     ...
```

The most complex part of the AST defines statements and expressions. We define a type constructor for each syntactical element given in the CSX definition. We show part of the statement datatype definition below. Each statement type contains its constituent elements and a pair of integers. The integers are used to indicate the line and column where the construct appears in the source program:

```
datatype
stmt = NoneStmt
     | Block of (csx_decl list * stmt list * int * int)
     | IfThen of (expr * stmt * int * int)
     | IfThenElse of (expr * stmt * stmt * int * int)
     | While of (expr * stmt * int * int)
     | LabeledWhile of (label * expr * stmt * int * int)
     | Asg of (expr * expr * int * int)
     | Read of (expr * int * int)
     ...
```

# 5 CSX Symbol Table

A compiler writer implements a symbol table for a block-structured language using a stack of hash tables. In addition to a hash table, we store the name of the current method under analysis and label stack. CSX supports the use of labels on `while` loops, so the table must store that information as well.

We provide ancillary functions for searching the symbol table based on the semantics of block-structured languages. The symbol table in this implementation also contains functions for retrieving information about items in the structure. Our symbol table supports several operations, all of which are written in a functional style:

```
insert_symb : string * symb_info * symb_table -> symb_table
make_symbol_table : unit -> symb_table
symb_exists : string * symb_table -> bool
get_symb : string * symb_table -> symb_info
```

A `symb_info` is a tuple containing the name of a symbol, its type, the declaration, and metadata to help determine the origin of the symbol:

```
datatype
symb_info = CSXMethod
          | CSXField
          | CSXLocalDecl
          | CSXFormalArg
          | CSXLabel
          | CSXClass of (string)
          | NoneSymbInfo
          | Info of {name   : string,
                     tipe   : csx_type,
                     decl   : csx_decl,
                     whatisit : symb_info}
          | CodegenInfo of {info : symb_info,
                            addr : jvm_address}
```

The `name`, `tipe`, and `whatisit` fields of the `Info` tuple can be derived from the `decl`. We duplicate this information to assist with pattern matching elsewhere in the compiler. The `CodegenInfo` tuple contains a `jvm_address`, which allows the compiler to quickly determine whether a symbol refers to a field or local variable.

# 6 CSX Parser

The parser generator `ml-yacc` emits an LALR parser based on the input specification. The specification contains three sections: one for user definitions, another for `ml-yacc` directives, and the last for parsing rules from which `ml-yacc` derives the parse table [8].

In the first section, we import three structures that are used within the parser: String, Char and DataTypes. String and Char are basic utilities for string and character manipulation and come as part of the SML environment. The DataTypes structure contains the AST datatypes discussed earlier.

In the second section, we define the set of terminals and non-terminals. Each named non-terminal is associated with an appropriate abstract data type. For example, the `METHODDECL` non-terminal is of type `csx_decl`:

```
%nonterm PROGRAM of csx_program
     |  MEMBERDECLS of csx_member_decls
     |  FIELDDECLS of csx_decl list
     |  METHODDECLS of csx_decl list
     |  METHODDECL of csx_decl
     ...
```

This section also contains directives for the parser to disambiguate the grammar by specifying token precedence and to identify the starting symbol. Other options assist with debugging the generated parser. For example, the `%verbose` directive produces a ".desc" file that summarizes any errors in the specification, and provides a list of all possible states of the parser[8].

In the last section, we encode the context-free grammar, expressed in extended Backus-Naur Form (EBNF). For instance, a `void` method declaration with no arguments has the following EBNF production:

```
METHODDECL : RW_VOID ID LPAREN RPAREN LBRACE
             FIELDDECLS STMTS RBRACE OPTIONALSEMI (
              (Method(Void, id(ID), nil,
                     FIELDDECLS, STMTS,
                     RW_VOID1left, RBRACE1right))
            )
```

All constituent pieces of the method declaration are available in an SML block, and we return a value containing all information about the method declaration, including the return type, the argument list, and so on. The parser generator also provides position information for each symbol on the right-hand side of the production. For example, `RW_VOID` begins at `RW_VOID1left` and ends at `RW_VOID1right`. In general, the positions are given by name$_{n+1}$left and name$_{n+1}$right where $n$ is the number of occurrences of the symbol to the left of the symbol in the production [8].

## 6.1 If/Else Shift-Reduce Conflict

Ambiguities in the grammar result in a shift-reduce conflict, in which the parser does not know whether to follow a reduction rule and execute a semantic action or continue reading from the token stream.

Consider the following short program:

```
if(a)
    if(b)
        print("a and b are true");
else
    print("a is not true");
```

When $a$ is false, nothing should be printed at all in CSX. By default, a parser generated by `ml-yacc` assumes a shift operation when this ambiguity is present, and thus emits the equivalent of this program:

```
if(a) {
    if(b)
```

```
        print("a and b are true");
}
else {
    print("a is not true");
}
```

We direct `ml-yacc` to associate `RW_ELSE` to the left, so that the parser performs a reduction instead of a shift, yielding a parse tree equivalent to this program:

```
if(a) {
    if(b) {
        print("a and b are true");
    } else {
        print("a is not true");
    }
}
```

# 7 Semantic Analysis

The type-checker traverses the AST and builds a new AST containing type information it discovers during semantic analysis. Pattern matching is a key component of the type-checker implementation. For example, we define a type constructor for each possible type in CSX:

```
datatype csx_type = Int | Bool | Char | Void | Error
                  | UnknownConst | NoneType
                  | StringLit of int
                  | Const of csx_type
                  | Param of csx_type
                  | Array of (csx_type * Int32.int)
                  | ArrayParam of csx_type
```

We pattern match on these values in several ways in the type checker. In the case of arithmetic, the CSX program might contain the expression $'a' + 1000$. The compiler takes the safest route possible by assuming the expression could overflow. Thus, it determines the type of the expression should be the "widest" of the two operands. In CSX, variables of type `char` and `int` are 8 bits and 32 bits wide, respectively. We utilize SML's pattern matcher to implement `widest()`. It does not make sense to find the widest of two types where at least one of them is not integral, and we return the `Error` type in that case. Other parts of the compiler test for the presence of the `Error` value and report the situation to the end user:

```
fun widest(Char, Int) = Int
  | widest(Int, Char) = Int
  | widest(Char, Char) = Char
  | widest(Int, Int) = Int
  | widest(Error, _) = Error
  | widest(_, Error) = Error
  | widest(Const(t0), Const(t1)) = widest(t0, t1)
  | widest(Const(t0), t1) = widest(t0, t1)
  | widest(t0, Const(t1)) = widest(t0, t1)
  | widest(_, _) = Error
```

The type-checker rebuilds the abstract syntax tree given by the parser and embeds extra type information within it. Expressions in CSX can be

- a literal value such as a character, integer, string, or boolean
- an identifier
- a function call
- an array dereference
- an arithmetic operation such as add, subtract, multiply, divide
- the logical operators &&, ||, and !
- the relational operations ==, !=, <=, >=, >, <

Some expressions are not composed of any others, and determining their type is trivial. For composite expressions, we must determine the type of the operands and apply a rule given in the CSX specification. For example, $1 + x$ is of type `int` so long as $x$ is a type that can be added to an `int`; any other type of $x$ would produce an error.

# 8 Code Generation

The code generator uses the augmented AST to generate appropriately typed bytecode for the Java Virtual Machine (JVM). With the JVM as our platform, we must map CSX features to the JVM facilities. A CSX class becomes a Java class. CSX fields become public static Java fields. CSX methods become public static Java methods. Variables on the runtime stack are stored in registers on the JVM. The registers on the JVM are virtually unlimited and are identified by an integer.

### 8.1 Jasmin

The code generator emits Jasmin code, an assembler notation for JVM bytecode. The last step of compilation from assembler to bytecode is handled by the Jasmin open source project [9]. Jasmin reads the assembler text and emits a Java class file.

## 9 The Glue

The glue provides the main entry point of the compiler, and it integrates each of the parts we previously discussed. It implements the rules to determine whether the compiler should proceed from one phase to the next. Fatal errors occurring during compilation propagate back to the glue as an exception. The glue is straightforward; without any defensive tests to determine whether to proceed from one phase to the next, it would say:

```
fun main(prog_name, args) = (
  if length(args) = 2 then (
       codeGen(typeCheck(parse(lex(hd(tl(args))))))
       handle e => (print("Compilation failed: "^
                   exnToString(e)^"\n");
                   1)
  )
  else (
    usage();
    1
  )
)
```

The final integration point for all parts in the compiler is a shell script that invokes the CSX compiler, Jasmin assembler, and JVM for execution of the program. The script's return code is that of the CSX program under execution. If the compiler or the assembler fails, it reports an error and exits.

## 10 CSX Runtime Support

Some utilities must be present for a language to have any useful features. Fischer et al. provide a runtime library that implements the underlying functionality of `read()` and `print()`[2]. It also provides other semantic details of the language, such as checking array lengths on assignment. The CSX language specification states that an exception should be thrown when the

lengths do not match. This code, and the exception itself, is defined in the CSX runtime library.

We refer to this library in the generated code as needed. This library is placed on the Java classpath when the CSX shell script invokes the JVM with the compiled class file.

# 11 Testing

It is imperative to test a compiler during development. Even the lexing and parse phases must be tested, which can be difficult because these phases use automatically generated code. The best approach is to build the lexer, parser, and a special "unparse" phase that prints the AST created by the parser. Only when all terminals and non-terminals in the language definition are handled appropriately in the lex and parse stages can the unparse be successfully completed. Once this phase is debugged, it is safe to proceed to semantic analysis.

A good testing strategy for semantic correctness is to first write a set of positive tests. A positive test is a program that the CSX compiler accepts. A valid program without semantic error succeeds in execution and returns 0. We have 85 such programs for the SML implementation. Approximately 60 of the 85 originated in the test suite for the Java implementation, a smaller portion of which are adapted from the CS541 course materials [1, 2]. The remaining programs test for bugs appearing during the development of our SML implementation. Each program performs an action and tests the result. If the result is not what is naturally expected, it prints a message stating the problem and exits with a non-zero exit code. For example, the CSX specification states that `false` $<$ `true` evaluates to `true`. The following program tests for this property and returns a non-zero exit code when the test fails:

```
class false_lt_true {
       void main() {
              if(false < true) {
                     print("ok\n");
              } else {
                     print("false < true should be true.\n");
                     exit(1);
              }
       }
}
```

When the compiler accepts a sufficient set of CSX programs exercising each part of the type checker and code generator, the developer should likewise construct a set of negative tests. A negative test for the CSX compiler is a program that either fails compilation or exits abnormally during execution. We have 85 such programs adapted from the set of positive tests, and 6 more that test for bugs appearing during development of the SML implementation.

For example, the CSX specification does not allow the addition of a `bool` and `int`. An attempt to compile this program should result in a non-zero error code and a meaningful message indicating the source of the problem:

```
class add_bool_to_int {
       void main() {
          bool x = true;
          print(x + 1);
       }
}
```

Each test program should be small and print exactly what was expected. We expect negative tests to compile and run but then exit with an error code. We test the return code in a shell script to detect a failure for a particular program. The entire test suite can be executed after each change, which aids the developer in finding bugs within the compiler. This approach is called "regression testing".

# 12 Debugging

SML does not come with a debugger like `gdb`. The easiest approach is to debug with `print()` statements. Every function has a return type, so we have to group statements together in a block to achieve this effect. The last expression in a block of expressions is its return value:

```
fun myfun(a, b, c) = (
    debug(a); (* a side effect *)
    [a, b, c] (* the return value *)
)
```

Debugging the code generator involves much of the same process, but with the addition of debugging the code emitted by the compiler. In some situations, it can be difficult to determine which bytecode to generate. It can be informative to view the bytecode of a compiled Java program that achieves the desired effect. The Java `javap` command decompiles a Java class file:

```
$ javap -c HelloWorld.class
Compiled from "HelloWorld.java"
public class HelloWorld
public HelloWorld();
Code:
0:  aload_0
1:  invokespecial #1 // Method java/lang/Object."<init>":()V
4:  return

public static void main(java.lang.String[]);
Code:
0:  getstatic #2 // Field java/lang/System.out:Ljava/io/PrintStream;
3:  ldc #3 // String Hello world.
5:  invokevirtual #4 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
8:  return
```

# 13 The Java Virtual Machine (JVM)

The JVM provides several advantages as a target platform. First, the JVM is available and supported on several architectures. Second, the JVM provides architecture specific runtime optimization; Java bytecodes are adaptively compiled into native code as the JVM detects performance bottlenecks [10]. This behavior is especially beneficial with numerical computation. Next, the JVM provides generational garbage collection that is configurable upon startup, unlimited virtual registers, and immediate access to a large library. Libraries that are developed in other languages can be used in another language hosted via the JVM. Last, there is an enormous community around the JVM, so it is easy to find documentation and support for a given problem. These features make the JVM an attractive option for new languages.

# 14 Conclusion

SML differs greatly from the typical object-oriented and imperative approach to programming. The most radical difference is found in its type system. Types are inferred; the user is not required to declare types in every situation, which makes some programs shorter. However, programs in SML are not necessarily easier to understand than, say, the Java counterpart. Having written a CSX compiler in both Java and SML, I feel that the CSX

implementation was more difficult to achieve, but it is more stable.

The Java implementation I wrote contains 5796 lines of code, and the SML implementation contains 2969 lines, including the lexer specification, parser specification, blank lines and comments. Blank lines and comments increase readability within the sources, so I feel it appropriate to include them. Counting lines of code does not account for the program's complexity, but it offers an indication to the relative effort involved between two programs [11].

## 14.1 Side Effects

The primary difference between the Java and SML implementations lies in the treatment of the AST during semantic analysis. We modify the AST in the Java implementation, but rebuild the AST in the SML implementation.

The CSX AST in the Java implementation provides fields that are unused until the semantic analysis phase begins. These fields are initialized as type information is discovered. The CSX AST in the SML implementation uses type constructors. Unlike Java classes, all fields in an SML type constructor must be initialized when the value is created, and the field values cannot change. Because the AST uses immutable data types only, the type checker creates a new AST as its result. The type checker determines the type of an expression, decorates any composite parts, and decorates the expression with the type it determines during analysis:

```
fun check_expr(expr, table) =
      let val tipe = typeof_expr(expr, table) in
        Checked(tipe, rebuild_expr(expr, table))
      end
```

The symbol table is based on a `HASH_TABLE` from the Basis library, which uses a `ref` internally [12]. We use a `ref` in the SML implementation only for the following:

- line and column position in the lexical analyzer
- presence of a semantic error in the type-checker
- label generation for `goto` support in the code generator

## 14.2 The SML Community

The SML community is small, and it is difficult to find relevant discussions on SML programming. For example, there are only 3,144 questions on Stack-Overflow relating to SML [13]. However, there are several other emerging

and popular functional languages from which to choose for future projects. For example, Microsoft developed F#, a functional language for the .NET environment offering a feature set similar to SML [14]. Haskell is a statically typed functional language with type inferrence, concurrency support, and a vibrant community [15]. OCaml continues to gain popularity for financial applications, and it is still under active development [16].

# References


[1] *Syllabus: CS541, Compiler Design.* `http://www.cs.uky.edu/~raphael/courses/CS541/backgr.html`. Accessed 22 May 2015.

[2] *Crafting a Compiler.* `http://www.cs.wustl.edu/~cytron/cacweb/`. Accessed 22 May 2015.

[3] *Standard ML of New Jersey.* `http://www.smlnj.org/`. Accessed 22 May 2015.

[4] *Why ML/OCaml Are Good for Writing Compilers.* `http://flint.cs.yale.edu/cs421/case-for-ml.html`. Accessed 22 May 2015.

[5] Andrew Appel. *Modern Compiler Implementation in ML.* Cambridge: Cambridge University Press, 1998.

[6] *The SML/NJ Compilation Manager (CM).* `http://www.smlnj.org/doc/CM/`. Accessed 22 May 2015.

[7] *ML-Lex: A Lexical Analyzer Generator for Standard ML.* `http://www.smlnj.org/doc/ML-Lex/manual.html`. Accessed 22 May 2015.

[8] *ML-Yacc User's Manual version 2.4.* `http://www.smlnj.org/doc/ML-Yacc/index.html`. Accessed 22 May 2015.

[9] *Jasmin Home Page.* `http://jasmin.sourceforge.net/`. Accessed 22 May 2015.

[10] *Java SE HotSpot at a Glance.* `http://www.oracle.com/technetwork/articles/javase/index-jsp-136373.html`. Accessed 15 June 2015.

[11] Robert Zeidman. *The Software IP Detective's Handbook Measurement, Comparison, and Infringement Detection.* Upper Saddle River, NJ: Prentice Hall, 2011.

[12] *The Standard ML Basis Library*. http://sml-family.org/Basis. Accessed 22 May 2015.

[13] *Stack Overflow*. http://stackoverflow.com/search?tab=newest&q=sml. Accessed 15 June 2015.

[14] *Visual F#*. https://msdn.microsoft.com/en-us/library/dd233154.aspx. Accessed 22 May 2015.

[15] *Haskell Language*. http://www.haskell.org/. Accessed 1 June 2015.

[16] *(* Musings from Jane Street's OCaml Developers *)*. https://blogs.janestreet.com/category/ocaml/. Accessed 22 May 2015.