AARHUS
UNIVERSITY
SCIENCE AND TECHNOLOGY

A Case Study in Modular Programming: Using AspectJ and OCaml in an Undergraduate Compiler Project

Department of Computer Science

Aske Simon Christensen
Jan Midtgaard
Johnni Winther
Ian Zerny

Report

ISBN 978-87-7507-314-6

DOI 10.7146/aul.6.7

A Case Study in Modular Programming: Using AspectJ and OCaml in an Undergraduate Compiler Project

Aske Simon Christensen Jan Midtgaard Johnni Winther
Ian Zerny

Department of Computer Science, Aarhus University[*]

April 2015

Abstract

We report our experience in using two different languages to build the same software project. Specifically, we have converted an entire undergraduate compiler course from using AspectJ, an aspect-oriented language, to using OCaml, a functional language. The course has evolved over a period of eight years with, on average, 60 students completing it every year. In this article, we analyze our usage of the two programming languages and we compare and contrast the two software projects on a number of parameters, including how they enable students to write and test individual compiler phases in a modular way.

[*]The work described in this report was carried out while the authors were at Aarhus University. Current affiliations: Aske Simon Christensen, CLC Bio, Jan Midtgaard, Technical University of Denmark, Johnni Winther and Ian Zerny, Google.

Contents

List of Figures

1 Introduction

Programming languages come with claims of modularity, flexibility, and type safety. However, few implementation reports provide evidence to substantiate these claims. In this case study, we evaluate the implementation of a compiler for a subset of the Java programming language using two different programming languages and development environments: One, AspectJ, is aspect-oriented, and the other, OCaml, is functional. The compilers have been reference implementations for the undergraduate compiler course at Aarhus University. Each implementation is therefore under hard design constraints. As such, we can meaningfully compare and contrast the two implementations and the implementation languages of the same software project. The compiler project is substantial in that it features a non-trivial subset of the Java programming language (see Table 1, page 2).

This paper is structured as follows. After having introduced the project in more detail (Section 2) we describe the overall architecture common to the two compilers (Section 3) and then analyse the use of AspectJ and OCaml on a number parameters (Section 4). Finally we discuss related work (Section 5) and conclude (Section 6).

2 Domain of discourse

The two reference compilers have been implemented as part of an undergraduate compiler course. The use of the compilers as an educational tool has a significant influence on their designs. In this section, we briefly outline the structure of the compiler course. We then clarify which requirements the educational context places on the implementations. Finally, we specify what measures we consider for evaluating the two solutions.

2.1 The compiler project

We first establish the common ground of the compiler project: the feature set of the input (henceforth: source) language and the phases that compose the compiler. Both the source language features and the compiler phases are the same for both compiler implementations.

2.1.1 The source and target language

The source language of the compiler project is a subset of Java 1.3, named Joos.[1] The course makes use of three such subsets containing an increasing number of features:

$$\textit{Joos0} \subset \textit{Joos1} \subset \textit{Joos2} \subset \textit{Java 1.3}$$

[1] Joos is an acronym for *Java Object-Oriented Subset*

Feature	Joos0	Joos1	Joos2	Java1.3
classes	✓	✓	✓	✓
interfaces			✓	✓
packages		✓	✓	✓
subtyping		✓	✓	✓
non-static methods	✓	✓	✓	✓
static methods		✓	✓	✓
non-static fields	✓	✓	✓	✓
static fields			✓	✓
checked exceptions		✓	✓	✓
exception throw			✓	✓
exception handling				✓
class library access		✓	✓	✓
implicit `this`		✓	✓	✓
closest match overloading			✓	✓
boolean, int	✓	✓	✓	✓
char, byte, short		✓	✓	✓
array		✓	✓	✓
multi-dimensional array			✓	✓
long, float, double				✓

Table 1: Selected features in Joos0, Joos1, Joos2, and Java 1.3

Joos0 is the smallest of the Java subsets containing simple primitive types and strings, expressions, statements, and class definitions. Basic I/O can be performed through `System.in.read` and `System.out.print`. This subset does not include subtyping and exceptions,[2] yet it is big enough to implement, e.g., lists and trees, as classes with non-static fields. Joos1 extends the Joos0 subset with arrays, subtyping, packages, exception checking, etc. In addition, the resulting language can use a large fraction of the Java 1.3 standard library. As such, Joos1 represents a significant subset of Java 1.3 yet it is suitable for a compiler course project and possible to handle in a clean phase-separated compiler design. Joos2 extends the Joos1 subset even further toward the full Java 1.3 language with interfaces, static fields, closest match overloading, exception throwing, multi-dimensional arrays, etc. Overall, Joos2 represents the biggest Java 1.3 subset manageable within the same phase architecture as Joos1. The feature set of each of the languages has been designed to maintain the above subset relationship. Table 1 illustrates selected high-level features for each subset. To be more concrete, we include in Figure 1 the grammar of lvalues [1, pp. 64-65] and expressions in the Joos2 language.

The target language of the compiler project is the Java Virtual Machine (henceforth: JVM) in the form of the Jasmin assembly format [24, 25].

```
lvalue ::= id                                        (local)
         | expr . id                          (non-static field)
         | type . id                              (static field)
         | expr [ expr ]                           (array index)

  expr ::= null                                   (null literal)
         | int                                 (integer literal)
         | char                              (character literal)
         | string                               (string literal)
         | bool                                (boolean literal)
         | op expr                             (unary operation)
         | expr op expr                       (binary operation)
         | this                                 (this reference)
         | lvalue                                       (lvalue)
         | lvalue = expr                            (assignment)
         | type . id ( expr, ... )               (static invoke)
         | expr . id ( expr, ... )           (non-static invoke)
         | id ( expr, ... )                      (simple invoke)
         | new type ( expr, ... )            (object allocation)
         | new type [ expr ] ...              (array allocation)
         | ( type ) expr                                  (cast)
         | expr instanceof type                (instanceof query)
```

Figure 1: Grammar of lvalues and expressions in Joos2

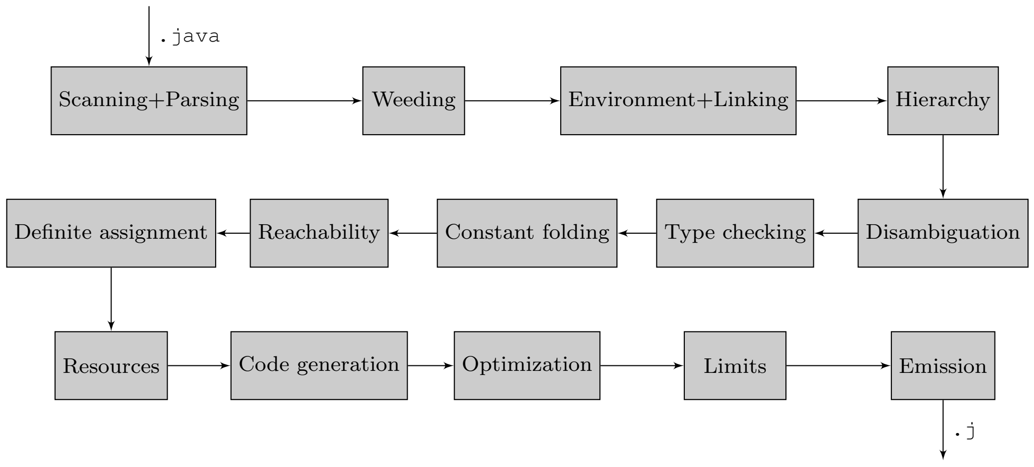

Figure 2: Basic compiler architecture

2.1.2 Basic compiler architecture

The pedagogical requirement for the implementation is modularity: students learn and program one compiler phase at the time. For inspiration, students are given the full source code for a small Joos0 compiler. In addition they are given a compiler skeleton on which they proceed to "fill in the blanks" of each phase towards a complete Joos1 compiler.[3] An online web-service allows students to upload a compiler phase, which is then compiled together with the remaining phases of the reference compiler, and finally the resulting compiler is run on a selection of more than 1200 test cases accumulated over a number of years. The students are thereby able to test each of their compiler phases in isolation. In order for such a service to be successful, the interfaces between phases have to be fixed. Figure 2 provides an overview of the involved phases which we describe next.

The characters of the input file are first grouped into tokens by the scanner (*Scanning*). The token stream is then parsed according to a context-free grammar and an abstract syntax tree (AST) is built (*Parsing*). Being context free, the parser accepts too many ASTs, some of which are subsequently rejected (*Weeding*). Names (local variables, type names, fields, methods, ...) are then mapped to their declaration (*Environment+Linking*). The *Hierarchy* phase is concerned with checking well-formedness of the class hierarchy and calculating sets of non-shadowed fields and methods for each type. Based on these sets the *Disambiguation* phase then resolves dot-separated identifier sequences. *Type checking* determines a type for each expression and lvalue, statically resolves overloaded method calls, and checks type consistency. The subsequent phase performs constant folding of expressions and of static final field declarations (*Constant folding*). Next come two static analyzes mandated by the Java Language Specification [12]: the *Reachability* analysis ensures that all statements of the program are reachable and the *Definite-assignment* analysis ensures that all local variables are properly initialized before being used. The *Resources* phase calculates signatures for types, methods, etc. and assigns each local variable to an integer offset in the stack frame. *Code generation* constructs (an internal representation of) a JVM byte code sequence for each method and constructor. The following *Optimization* phase performs peephole optimization of these sequences. The *Limits* phase ensures basic operand stack height consistency of the optimized byte code. In addition it calculates the maximal operand stack height needed for each method's stack frame. Finally the emitter serializes the internal byte-code representation into ASCII files (*Emission*), which are subsequently assembled into Java class files using the Jasmin assembler [25].

[2] A Joos0 class implicitly extends `Object` and all methods bypass exception checking by requiring the header `throws Exception`.

[3] For extra credit they can extend their Joos1 compiler with any number of Joos2 features.

2.2 Requirements

1. The implementation must be modular, where each module[4] represents a particular phase of the compiler in correspondence with the course structure. Each module should be reasonably self-contained with a clear interface defining its boundaries.
2. The implementation language and environment must be accessible to third-year computer-science students based on their previous education.

2.3 Evaluation criteria

Our evaluation does not take place against a predefined basis, but instead consists of a set of parameters that can be measured and evaluated within the project we outlined. We consider parameters that say something about general aspects of the implementation languages as well as specific solutions used within the implementations. We compare the implementations on each criteria in an attempt to clarify the relative strengths and weaknesses of the two implementations.

This evaluation *does not* establish any general properties of the languages. However, the present study does provide a single yet substantial point for which a relative comparison is meaningful. The specific aspects we consider and the evaluation criteria used are:

1. *The expressive power of the implementation languages.* Our abstract measure of expressiveness is lines of code. Following practice elsewhere [3], we measure both the lines of source code without comments as well as the number of bytes of compressed source code. The latter measurement attempts to compensate for the sometimes verbose and repetitive patterns and naming conventions common in idiomatic Java code.
2. *The availability of the language and tools on different platforms.* In particular, we will assess the ease with which these tools are installed and used on various operating systems and distributions.
3. *The availability of libraries.* To avoid starting from scratch, the compiler project depends on a number of libraries to facilitate the implementation.

We do not evaluate:

- *The development process.* The two compilers have been built at different times and by different people. The later reimplementation (in OCaml) benefited greatly from the existing implementation. Both in terms of design: the phases were well established; and in terms of the implementation: tricky details and corner cases were already dealt with and could simply be reimplemented.
- *The understandability of the implementations for the students.* We consider it a topic of teaching and hence outside the scope of this study.

[4]A module is some language- or implementation-specific notion of a separate component.

3 Design and implementation

This section describes the overall design of each of the two implementations.

3.1 Implementation languages

Java is a descendant of the Simula family of *object-oriented* programming languages, providing a class-based object model and static typing. AspectJ is an extension of the Java programming language which adds *aspect-oriented* features, allowing crosscutting modifications to Java structures, e.g., by specifying *join points* at which control flow can be rerouted elsewhere, and by *injecting* additional fields and methods into existing classes. For the implementation, we primarily use the latter feature.

OCaml is a descendant of the ML family of *functional* programming languages, providing static typing and full type inference. For the implementation, we use only traditional ML features: signatures, modules, algebraic data types, pattern matching, and to a limited degree, first-class functions and mutable cells. We avoid most of the advanced OCaml features including first-class and parametric modules, generalized algebraic data types (GADTs), polymorphic variants and the entire object-oriented layer.

In the following we detail the key features of abstraction, types, effects and the operational behavior of the two languages.

3.1.1 Abstraction

Interfaces and classes provide the specification/implementation facility in Java. Figure 3a illustrates how we might abstract two printing procedures using the Strategy Pattern [11]. On top of this, AspectJ adds a notion of aspects that can change the structure and behavior of interfaces and classes.

OCaml provides module types and (parameterized) module implementations as the specification/implementation facility. Figure 3b illustrates how we might abstract the equivalent printing procedures using modules. Alternatively, records and first-class lexically-scoped functions can provide similar lightweight abstractions as illustrated in Figure 3c.

3.1.2 Types and data

Both Java and OCaml are statically typed with a mostly sound type system. In Java, types are given by class definitions which allow subclassing. Interface and implementation (class) relationships are defined nominally, e.g., the class specifies the interface by name in the `implements` clause, as in Figure 3a.

In OCaml, types are structural and composed mostly from smaller types in literal expressions or by algebraic data-type definitions. For example, the `printer` type of Figure 3c is composed of two function types. Interface (signature) and implementation (module) relationships are defined structurally. For example, the `StdOutPrinter` module of Figure 3b is structurally an implementation of the `Printer` type without explicitly stating any relationship between

```
interface Printer {
  public void print(String str);
  public void println(String str);
}
class StdOutPrinter implements Printer {
  public void print(String str) { System.out.print(str); }
  public void println(String str) { print(str + "\n"); }
}
```

(a) A Java implementation using the Strategy Pattern

```
module type Printer = sig
  val print : string -> unit
  val println : string -> unit
end
module StdOutPrinter = struct
  let print str = print_string str
  let println str = print (str ^ "\n")
end
```

(b) An OCaml implementation using modules

```
type printer = {
  print : string -> unit;
  println : string -> unit
}
let stdout_printer =
  let print str = print_string str in
  let println str = print (str ^ "\n") in
    { print = print; println = println }
```

(c) An OCaml implementation using records and first-class functions

Figure 3: Abstracting printing procedures

the two. As a consequence, we can pass the `StdOutPrinter` module to any client expecting a `printer` type.

Figure 4a shows how to define the abstract syntax tree of lvalues as a class hierarchy in Java. Each subclass of `Lvalue` represents a kind of lvalue and corresponds to a production in the grammar of lvalues from Figure 1. A `toString` method is defined to construct the string representation of an lvalue. Virtual-method dispatch provides case discrimination on the distinct kinds of lvalues.

Figure 4b shows how to define the abstract syntax tree of lvalues as an algebraic data type in OCaml. Here, each case (constructor) of `lvalue` represents a kind of lvalue. A `string_from_lvalue` function is defined to construct the string representation of an lvalue. Here, pattern matching provides case discrimination on the distinct kinds of lvalues.

```
public abstract class Lvalue {
  public String toString();
}
public class Local extends Lvalue {
  public Identifier id;
  public String toString() {
    return id.toString();
  }
}
public class NonstaticField extends Lvalue {
  public Expression receiver;
  public Identifier name;
  public String toString() {
    return receiver.toString() + "." + name.toString();
  }
}
public class StaticField extends Lvalue {
  public NamedType type;
  public Identifier name;
  public String toString() {
    return type.toString() + "." + name.toString();
  }
}
public class ArrayIndex extends Lvalue {
  public Expression base;
  public Expression index;
  public String toString() {
    return base.toString() + "[" + index.toString() + "]";
  }
}
```

(a) A class hierarchy of lvalues in Java

```
type lvalue =
  | Local          of identifier
  | NonstaticField of expression * identifier
  | StaticField    of named_type * identifier
  | ArrayIndex     of expression * expression

let rec string_from_lvalue lvalue = match lvalue with
  | Local (id)
    -> string_from_identifier id
  | NonstaticField (receiver, name)
    -> string_from_expression receiver ^ "." ^ string_from_identifier name
  | StaticField (ntype, name)
    -> string_from_named_type ntype ^ "." ^ string_from_identifier name
  | ArrayIndex (base, index)
    -> string_from_expression base ^ "[" ^ string_from_expression index ^ "]"
```

(b) An algebraic data type of lvalues in OCaml

Figure 4: Abstract syntax tree representations of lvalues[†]

[†]These figures assume definitions of identifiers, types and expressions similar to the lvalue structure being defined.

```
interface StringMapInterface {
  // Throws an unchecked exception on unknown keys
  public class NotFoundError extends RuntimeException {}
  public String lookupImplicit(String key);

  // Throws a checked exception on unknown keys
  public class NotFoundException extends Exception {}
  public String lookupExplicit(String key) throws NotFoundException;
}
class StringMap implements StringMapInterface { /* ... */ }

String foo(StringMap map) {
  String str1 = map.lookupImplicit("key1");
  String str2;
  try {
    str2 = map.lookupExplicit("key2");
  } catch (NotFoundException e) {
    str2 = "key2 not found";
  }
  return str1 + " " + str2;
}
```

(a) Exceptions in Java

```
module type STRING_MAP = sig
  type t

  (* Throws an (unchecked) exception on unknown keys *)
  exception NotFound
  lookup_implicit : t * string -> string

  (* Returns 'NotFound' on unknown keys *)
  datatype result = Found of string | NotFound
  lookup_explicit : t * string -> result
end
module StringMap : STRING_MAP = struct (* ... *) end

let foo map =
  let str1 = StringMap.lookup_implicit map "key1" in
  let str2 = match StringMap.lookup_explicit map "key2" with
    | Found str -> str
    | NotFound  -> "key2 not found"
  in str1 ^ " " ^ str2
```

(b) Exceptions and types in OCaml

Figure 5: Exceptions and types

3.1.3 Effects

Java is an imperative language with mutation being the default. All variables and most data structures are mutable. OCaml is a largely pure functional language with immutability being the default (a notable exception is OCaml's mutable strings[5]). However, variables can be marked as mutable and explicit reference cells can be used to mutate data structures.

Both languages support a form of exceptions for error recovery and other control effects. Here Java has two kinds of exceptions: checked and unchecked. Unchecked exceptions are exceptions that can occur at any point during execution and do not affect the static semantics of the program. Checked exceptions must be declared as part of a method's type signature. When a method declaring checked exceptions is invoked, the caller must handle each declared exception or itself declare it as part of its own type signature. These constraints are statically enforced by the Java compiler. Figure 5a illustrates these two kinds of exceptions.

OCaml's exceptions are unchecked. An equivalent of checked exceptions can instead be represented using a data type in the existing static type system. Figure 5b illustrates the use of a result type to denote the statically enforced handling of the exceptional case.

3.1.4 Operational behavior

Both languages use an eager evaluation strategy where function application, or method invocation, evaluates arguments prior to entry. A peculiarity of the OCaml implementation is that it evaluates arguments from right to left, although the actual order is left unspecified by the language specification [23]. Both languages pass arguments by value. A specificity of Java programming language is that the value being passed for a heap-allocated object is a reference (pointer) to this object.

3.2 Implementation differences

With the language details in place, we now turn to how the implementations use the features of the languages. The major difference between the two compiler implementations is the way in which they accumulate information: one is imperative using shared and mutable data structures (AspectJ), another is functional using persistent immutable data structures (OCaml). This difference echoes a categorization from the abstract syntax tree compiler literature:

- *The "Christmas tree" compiler model* [1] in which a *mutable* abstract syntax tree structure accumulates information and (declaration) pointers as the tree is traversed and

- *The transformational compiler model* originating with Steele [34] (and since pursued in both untyped and typed settings [9, 18, 21]) in which different *immutable* tree structures capture the available information and in which additional information is accumulated through a tree-to-tree mapping.

[5]The first steps towards making OCaml's strings immutable have been taken with the recent 4.02 release.

The AspectJ implementation follows the Christmas tree model. Specifically the implementation makes use of the SableCC parser generator [10] to concisely and declaratively specify the initial abstract syntax tree. SableCC then uses this specification to automatically generate a Java class hierarchy which is similar in structure to the example in Figure 4a. Once generated, the compiler uses AspectJ to inject new fields on the auto-generated classes thereby decorating the tree with additional information for use during the compilation phases. This synergy between AspectJ and SableCC forms the basis of the AspectJ compiler design.

The OCaml implementation follows the transformational model. Each tree is defined by a data-type declaration similar to the example in Figure 4b. These trees are immutable, i.e., they make no use of reference cells or other mutable fields. Each phase is defined as a pure function mapping a tree of one type to a tree of another type. This functional mapping between algebraic data types forms the basis of the OCaml compiler design.

3.2.1 Modularity

By modularity of the compiler, we refer to how the compiler separates the various phases which an input program is subjected to during compilation. In the context of the compiler course, and listed as Requirement 1 in Section 2.2, it is important that each phase is independently specified and well-defined. The implementation must be modular for two reasons. First, from an educational point of view, a clear specification and implementation of each phase aids understanding. Second, from a technical point of view, and of particular importance to the course structure, is the ability to build a full compiler with some phases supplied by a student and with the rest of the phases copied from the reference implementation. This modularity allows the students to build and test a full compiler despite not having implemented all of its phases.

In the AspectJ implementation, we achieve modularity by implementing each compiler phase as an *aspect*. Each aspect, representing a compiler phase, is responsible for injecting fields and methods into the proper classes. For example, the typechecking phase is implemented as an aspect that will inject a type field on all lvalue and expression nodes. The injected information forms the interface (or *contract*) between the individual phases.

In the OCaml implementation, we achieve modularity by implementing each compiler phase as a program transformation from one algebraic data type to another. For example, the weeding phase is implemented as a statically-typed function:

```
Weeding.weed_program : Ast.program -> Weedingast.program
```

where `Ast` and `Weedingast` are distinct modules each declaring a `program` type. In this case the static type information forms the interface (or contract) between the individual phases.

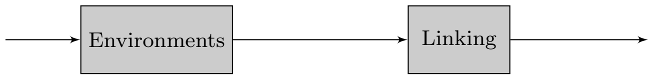

Figure 6: Environment and linking phases in the AspectJ implementation

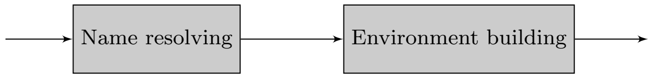

Figure 7: Environment and linking phases in the OCaml implementation

3.2.2 Environments and Linking implementation

The collective goal of the environment and linking phases is to provide means for looking up a member (a field, a constructor, or a method) on a particular reference type. The two implementations, however, achieve this goal in inherently different ways. The AspectJ version works by setting (injecting) a pointer from the use of a reference type to the corresponding declaration. The OCaml version, in contrast, builds a separate immutable data structure, `types`, which represents what a particular reference type means, e.g., the signature of its constructors, methods, and fields. Moreover the two phases are reordered: the AspectJ implementation first injects symbol tables into AST nodes (for method environments, field environments, and local environments), and subsequently injects a pointer from reference types to their declaration. In contrast, the OCaml implementation first builds a separate dictionary that maps (canonical) type names to the separate `types` structure and subsequently checks missing or duplicate declarations of fields, locals, etc. based on the dictionary. We illustrate the different order of the two phases in Figures 6 and 7.

3.2.3 Peephole optimization

The peephole optimization phase of the two implementations also differ significantly. Like GCC, the original AspectJ-version defines a domain-specific language (DSL), including a lexer, parser, pattern matching compiler, etc. to express patterns for peephole optimization. In contrast, the OCaml version relies on the builtin pattern matching functionality, expressing each peephole pattern as a pure, pattern matching procedural value. This design thereby follows the embedded DSL tradition [15].

3.2.4 Error reporting

The AspectJ implementation attempts to report a sequence of errors for programs containing several errors. In contrast, for simplicity, the OCaml implementation reports the first error and exits.

4 Evaluation

In this section we proceed to evaluate the two compiler implementations. We split the evaluation in two parts: the first part evaluates language-level parameters whereas the second part evaluates system-oriented parameters.

4.1 Language and design

4.1.1 Code conciseness

Table 2 summarizes the lines of code in each of the two implementations. Lines of code is a crude measure of code conciseness. It does not take into account programming style. As such idiomatic Java code with comments and lots of whitespace appear needlessly long. Nevertheless the measure gives some indication as to which things are easy to express in one language and harder to express in another. In the following subsections we will proceed to break down and analyse Table 2 in more detail.

From a first reading of (the two last lines of) Table 2, it is clear that the AspectJ implementation requires substantially more lines of code (8436+44624 LoC vs. 6790+1173+9029 LoC): a factor of three. What is perhaps more surprising, is that the two implementations use roughly the same number of handwritten lines (8436 LoC vs. 6790+1173 LoC). SableCC generates a surprisingly large amount of code, consisting of a lexer, a parser, `toString` methods, various forms of visitor patterns, etc. In contrast ocamllex and ocamlyacc generate only a barebones lexer and parser.

A number of parameters affect the brevity of the OCaml implementation.

- The tree traversals are structural over the data types, with one pattern-matching function per type. We have preferred explicit functions, e.g., structurally walking a list of statements by pattern matching, over relying on `List.map` and `List.fold_left` from the standard library. Since part of the reference compiler is handed out as a skeleton to students (novices to OCaml and functional programming) this preference should lower the bar at the cost of verbosity.

- We have chosen to decouple pattern matching of a formal parameter from the function headers themselves and instead use a separate `match-with` construct in function bodies. This eases code restructuring, e.g., inserting an additional `let`-binding before the pattern match, but at the cost of additional characters.

- Finally we use OCaml's `begin-end` constructs to emphasize sequential evaluation of expressions, rather than the less verbose semicolon-separated parenthesized list – again in order to avoid that students confuse the constructions with the syntactically similar lists (semi-colon separated in square brackets) or tuples (comma separated in parentheses).

	Source LoC				Compressed LoC (bytes)		
	AspectJ	OCaml		Ratio	AspectJ	OCaml	Ratio
Phase		Phase	AST				
lexer	38[a]	210	–	5.53	526	1869	3.55
lexer (auto)	1256	588	–	0.47	3167	3145	0.99
parser	737[a]	849	–	1.15	3174	4113	1.30
parser (auto)	13629	8441	–	0.62	19600	23557	1.20
AST	80[a]	0	185	2.31	567	1477	2.60
AST (auto)	19117	0	–	–	128750	0	–
visitor	0	0	–	–	0	0	–
visitor (auto)	10622	0	–	–	22209	0	–
weeding	479	495	136	1.32	3266	5487	1.68
environment	161	125	126[b]	1.56	1170	2438	2.08
type linking/name resolving	275	442	130	2.08	1970	5154	2.62
hierarchy	500	396	–	0.79	3113	3493	1.12
disambiguation	235	336	128	1.97	1726	4111	2.38
type checking	1262	952	161	0.88	6489	8557	1.32
constant folding	256	375	–	1.46	1333	2999	2.25
reachability	115	75	–	0.65	832	935	1.12
definite assignment	493	178	–	0.36	2318	1630	0.70
resources	123	254	139	3.20	761	3430	4.51
code generation	807	734	83	1.01	3117	5010	1.61
optimization[c]	21	87	–	4.14	281	890	3.17
limits	107	153	85	2.22	974	2439	2.50
code emission	128	121	–	0.95	1094	1274	1.16
util	415	18	–	0.04	2039	411	0.20
class environment	183	97	–	0.53	1338	990	0.74
class-file parser	243	200	–	0.82	1914	1804	0.94
errors	147	295	–	2.01	2293	2319	1.01
instruction	1284	296	–	0.23	12711	1791	0.14
main	347	102	–	0.29	2909	1318	0.45
total handwritten	8436	6790	1173	0.94	37306	49686	1.33
total auto generated	44624	9029	–	0.20	72292	26530	0.37

[a] Number of lines of code in the relevant SableCC file section.
[b] Number of lines of code in the `types` module.
[c] Excluding the peephole pattern collection. The AspectJ numbers do not include code implementing the peephole optimization DSL, nor the AspectJ peephole driver.

Table 2: Number of lines of code (LoC) in each implementation. Each source LoC was obtained by first removing comments and blank lines and counting with `wc -l`. Each compressed LoC was obtained by further removing line breaks and redundant whitespace before compressing with `gzip`. Entries marked with *(auto)* denote auto-generated code.

All in all, the two implementations are very close according to the lines-of-code measure, with OCaml being a bit more concise on raw lines-of-code and AspectJ a bit more concise when considering compressed lines-of-code. This matches our expectation that the compressed lines-of-code measure will benefit the AspectJ implementation given the sometimes verbose and repetitive style of idiomatic Java. OCaml is sometimes cited as a *domain-specific language for compilers* thus it was surprising to us that the OCaml implementation did not turn out to be smaller than the AspectJ implementation. In our view, the fact that the AspectJ implementation is comparable to the OCaml implementation, regarding lines of handwritten code, indicates that AspectJ and SableCC are indeed very expressive tools for writing a compiler in Java.

4.1.2 Modularity

Both systems are designed with modularity in mind and satisfy Requirement 1 to a sufficient degree. In our compiler design, a module is associated with a compiler phase. Each phase is specified separately, as an aspect in AspectJ, or as a function in OCaml. When using the testing suite, the concrete implementation of a phase can then be supplied by either the student or by the reference implementation.

Even though both implementations satisfy the modularity requirement, the abstractions used to this effect are very different. A compiler phase is more flexible in the AspectJ implementation. For example, the implementation might inject administrative information as fields on the tree nodes and use them locally to the phase or as input to a later phase. This injection is not possible in the OCaml implementation: The data type of trees defines the interface between phases and changing it would require explicitly changing the phases that interact with it. The flip-side of flexibility is stability. The OCaml specification is very robust. All interactions between phases are determined by the static type of the phase.[6] Failure to comply with the interface is thus caught immediately at compile time. In AspectJ, the interactions between phases are determined by the concrete use that a particular phase makes of the information contained within shared data structures, e.g., the abstract syntax tree. The interactions of phases is therefore defined largely by the implementation and not by the static specification. Thus, some discipline is needed when implementing phases in AspectJ.

4.1.3 Tree types

The ASTs of both compilers are specified in a BNF-like notation: in OCaml the AST is specified as an algebraic data type in a separate module (summarized in the OCaml AST column of Table 2), whereas in SableCC the AST is specified

[6]For completeness, an OCaml implementation could potentially circumvent the fixed transformation interfaces dictated by static types by introducing, e.g., shared mutable state and using it to communicate with other phases. However, such use of state would need to be done deliberate and is not the intended design in the OCaml implementation. In contrast, injecting fields in classes is the intended design in the AspectJ implementation.

as a grammar in a dedicated section of the SableCC specification (summarized in the AST row of Table 2). From this specification SableCC auto-generates a class hierarchy that represents ASTs (summarized in the AST (auto) row of Table 2).

The concrete data types for representing ASTs are a bit more involved than the lvalue example from Figure 4, as they also represent position information. Position information is common to many node types in the AST, as they need it to generate user-friendly error messages. The auto-generated, object-oriented AST underlying the AspectJ implementation models the sharing of position information by inheritance: All AST nodes share a common super type, `Node`, which contains a position field along with getter and setter methods for manipulating positions. This object-oriented sharing in a common super type is reminiscent of the AST representation in the OCaml implementation: Rather than distributing the position information out into each variant (which would require a needless case dispatch to obtain the position), it is instead kept in a common record type along with the variant type. Any recursive occurrence of the type refers to the (outermost) record type, thereby retaining both the position and the pattern matching ability. Our example of lvalues is thus represented as follows in the OCaml representation:

```
type lvalue = { lvalue_pos: Lexing.position; lvalue: lvalue_desc }
and lvalue_desc =
  | Local          of identifier
  | NonstaticField of exp * identifier
  | StaticField    of namedtype * identifier
  | ArrayIndex     of exp * exp
```

The representation of lvalues after the type-checking phase extends this design by using an extended record to remember the type of an lvalue. Surprisingly, this representation did not arise in an attempt to mimic the AspectJ representation: Instead it was inspired by OCaml's own AST representation [23], which should be representative of best practice within the language.

The resulting tree types of the two implementations differ in that one is based on mutation: in AspectJ a sub-tree is replaced by destructing the original tree, whereas the other is based on duplication: in OCaml a sub-tree is replaced by creating a new AST with a new sub-tree. Furthermore the tree representations differ in that the auto-generated tree types from SableCC are doubly linked, with a parent and a child node each containing a reference to each other, whereas the OCaml tree types are structural, with a parent node containing references to (structurally smaller) child nodes (but not vice versa).

The singly-linked, structural representation has the advantage of enabling sharing: multiple identical sub-trees need only be allocated once with each position just containing a pointer to it. On the other hand, when a sub-tree changes, all nodes on the spine from the root of the AST to the corresponding sub-tree, need to be copied to preserve the singly-linked representation. In the OCaml implementation, the phases map between multiple data types. Each data type encodes which AST nodes that can occur at a specific point in the compilation process. To avoid needless copying from one such data type to the next, tree modules share data types that do not change between them. For example, the binary operations are invariant and hence shared across the first six phases, until occurrences of +, ==, and != have been correctly resolved by the typechecker, e.g., into string concatenation, address equality, and address inequality, respectively. In contrast, the AspectJ implementation shares the same auto-generated (sub)class hierarchy of binary operations across all compiler phases.

4.1.4 Tree traversal

The auto-generated visitor patterns in the AspectJ implementation have the advantage of being reusable: the same traversal code is used throughout almost all of the phases. In contrast, the OCaml implementation is based on hand-written traversal code. This code has the advantage of being customizable for the particular phase at hand, e.g., if a phase does not need to traverse certain sub-trees of the AST it simply doesn't do so. This tradeoff between reusability and customizability shows up in Table 2 for many transformation phases where the size of the OCaml code dominates the size of the AspectJ code: explicit traversal code simply takes up more space.

One phase stands out as incompatible with the auto-generated visitor code in AspectJ: the hierarchy phase, which resolves class hierarchy relationships and the inheritance, hiding, and overriding of fields and methods according to the Java Language Specification [12, chap.8,9], needs to traverse the class hierarchy in a depth-first (or topologically-ordered) manner. Concretely this is implemented in AspectJ by overriding the visitor methods for a type declaration to first check any declared super types. The resulting code is not as concise as the corresponding OCaml code when comparing the lines of code, however it may be needlessly punished by Java's tradition for repetitive code and long, descriptive method names, as the compressed AspectJ phase actually takes up less space than the OCaml phase (see Table 2's hierarchy row).

Alternatively, one could have chosen to write reusable fold functions for the AST data types that are shared across multiple phases in the OCaml implementation. For example, the constant folding, reachability, definite assignment, and resources phases all traverse the same typed AST representation of the program and would therefore potentially have benefited from such fold functions. We chose not to pursue this direction further, out of a desire to keep the skeleton code as simple as possible for newcomers to the language — which is in line with the requirements for the project.

4.1.5 Static guaranties

The two implementations differ in how many invariants of the project are expressed in (and thereby statically ensured by) the type system. Three invariants of this sort are guaranteed by OCaml's type system:

(a) *"there are no nodes of this sort after phase X"*. For example, names and method invocations are syntactically ambiguous and are therefore represented as `AmbiguousName` and `AmbiguousInvoke` constructors in the

AST data types up to the disambiguation phase. Once these syntactic forms are disambiguated based on name resolving and type checking, the constructors do not occur in the subsequent AST data types.

(b) *"there are no nodes of this sort before phase X"*. For example, implementation-wise arrays are special, with special-purpose instructions on the JVM. As a consequence the array-length primitive and the array-clone methods must be treated separately. However as arbitrary fields may be named `length` and arbitrary methods may be named `clone()`, we require type analysis to distinguish the array-cases from the rest. As a consequence, the `ArrayLength` and `ArrayClone` constructors do not occur in the AST data types before type checking.

(c) *"all nodes of this sort carry this information from phase X"*. For example, the resource-phase calculates an offset for each local variable as well as the signature of each method. As a consequence the AST data type contains an integer component for local variable declaration and use nodes, and a signature string for method declaration and call nodes.

In contrast, the AspectJ implementation uses a single class hierarchy to model the node types (as would a pure Java-based implementation), thereby not statically guaranteeing **(a)**, **(b)**, or **(c)**. The above static guarantees come at the price of code duplication in the OCaml implementation (compare the handwritten 80 LoC in AspectJ's AST row with the handwritten 1173 LoC that are the sum of OCaml's AST column in Table 2).

4.1.6 Emerging patterns

Throughout both implementations, design patterns arose to address similar issues in each their own way.

In the AspectJ implementation, the code makes repeated use of an auto-generated `getAncestor`-method, which accepts a class as argument and walks up the spine of the AST until it finds an instance of that class argument. For example, this method is used to type check occurrences of the subexpression `this`, in which the type checker needs access to the (type of the) outer class. A visit to a `this`-expression node will invoke the following excerpt which walks up the spine in the search for a type declaration node (`PTypeDecl`) and subsequently extracts and saves its types:

```
@Override
public void outAThisExp(AThisExp exp) {
    ...
    PTypeDecl typed = exp.getAncestor(PTypeDecl.class);
    exp.type = typed.type;
}
```

In the OCaml implementation a different pattern arose: as the AST traversals proceed from parent to child nodes without any immediate way to retrieve the parent of a given node, any such contextual information must be passed along as additional arguments. In the above example, to type check occurrences of the subexpression `this`, the type checker would therefore pass along (information about) the outer class declaration. The type checker however needs to pass multiple pieces of such contextual information, e.g., the return type of a method to type check `return` statements. Rather than having to repeatedly patch the function headers and their multiple call sites to each pass and accept additional arguments, we developed a pattern of *info records* which contain all such contextual information. For example, the following excerpt declares an info record type containing the current return type as well as the canonical name of the enclosing class declaration and the type environment in which to look it up:

```
type info =
    { tenv                : Types.type_env;
      type_cname          : Types.canonical_name;
      ...
      current_return_type : Types.typeexp;
      ... }
```

The traversal code all just pass *one* additional argument which is an instance of the above record type. If at one point the compiler implementer needs an additional piece of information to be passed, the info record type can easily be expanded without having to change the traversal code. This pattern of info records is reminiscent of attribute grammars [19]. In particular we typically pass *inherited attributes* as info records, which is a form of information passed downwards in the AST traversal.

One can imagine a number of alternative implementation choices for this particular issue: For example, in AspectJ we could have saved such contextual information in (phase-)global variables for constant time access. Similarly in OCaml we could have saved such contextual information in (phase-)global reference cells and thereby avoided the allocation and garbage collection of the info records.

4.1.7 Multiple return values

Figure 8 contains an excerpt from the definite assignment analysis of the two implementations. The analysis ensures that all local variables are assigned before they are used. Section 16 of the Java Language Specification [12] mandates that Java implementations use a specific static analysis to ensure this property. The analysis operates over four sets of locally declared variables:

- one set represents the set of variables that are *definitely* assigned *before* evaluating an expression,
- one set represents the set of variables that are *definitely* assigned *after* evaluating an expression, and
- two sets represent the set of variables that are *definitely* assigned *after* evaluating an expression to *true* or *false*, respectively.

```
public aspect DefiniteAssignment extends DepthFirstAdapter {

    public Set<ALocalDecl> PExp.defasnBefore = null;
    public Set<ALocalDecl> PExp.defasnAfter = null;
    public Set<ALocalDecl> PExp.defasnAfterTrue = null;
    public Set<ALocalDecl> PExp.defasnAfterFalse = null;

    public Set<ALocalDecl> PLvalue.defasnBefore = null;
    public Set<ALocalDecl> PLvalue.defasnAfter = null;

    ...

    private Set<ALocalDecl> infiniteSet = new HashSet<ALocalDecl>();

    ...

    @Override
    public void caseAAssignmentExp(AAssignmentExp exp) {
        PLvalue lvalue = exp.getLvalue();
        PExp rightExp = exp.getExp();
        lvalue.defasnBefore = exp.defasnBefore;
        lvalue.apply(this);
        rightExp.defasnBefore = lvalue.defasnAfter;
        rightExp.apply(this);
        if (lvalue instanceof ALocalLvalue) {
            exp.defasnAfter = new HashSet<ALocalDecl>();
            exp.defasnAfter.addAll(rightExp.defasnAfter);
            exp.defasnAfter.add(((ALocalLvalue)lvalue).local_decl);
        } else {
            exp.defasnAfter = rightExp.defasnAfter;
        }
        exp.defasnAfterTrue = exp.defasnAfter;
        exp.defasnAfterFalse = exp.defasnAfter;
    }

  ...

  }
```

(a) The AspectJ implementation of definite assignment analysis

```
and defass_exp exp scopeset b cld = match exp.TAst.exp with
    ...
  | TAst.Assignment (lvalue,exp) ->
    let a = defass_lvalue lvalue true scopeset b cld in
    let a,_,_ = defass_exp exp scopeset a cld in
    (match lvalue.TAst.lvalue with
      | TAst.Local x -> let a’ = Varset.add x a in
                        a’,a’,a’
      | _ -> a,a,a)
    ...
```

(b) The OCaml implementation of definite assignment analysis

Figure 8: Definite assignment analysis

Lvalues require only two such sets though: before and after.

The OCaml code for the function `defass_exp` models this by passing around sets (`b` in Figure 8b represents the *before set*) and returning a triple consisting of the three *after sets*. Similarly `defass_lvalue` accepts a 'before' set and returns an 'after' set. In contrast the AspectJ implementations injects four fields into expression nodes (`PExp`), and two fields into lvalue nodes (`PLvalue`). It remains only to connect together the sets appropriately. In OCaml this happens by passing an after set (`a`) obtained from the lvalue as an argument to the recursive `defass_exp` call. In AspectJ this happens by assigning to the injected `rightExp.defasnBefore` field before calling the visiting method.

If the value being assigned is a local variable it needs to be added to the set of definitely assigned ones. In (immutable) OCaml this is straightforward, whereas the mutable AspectJ implementation requires duplicating an existing set.

The OCaml code passes around a few additional inherited attributes: `defass_exp` passes `scopeset` which is the set of all local variables in scope and `cld` which is an option describing the possibly enclosing local declaration. Similarly `defass_lvalue` accepts a boolean to indicate whether the lvalue is being *assigned* (written) or read. Since these attributes differ between the different syntactic categories (statements, expressions, lvalues, ...) there is too little overlap in this case to warrant a common info record type as described in Section 4.1.6. The AspectJ implementation contains corresponding idioms representing the same information (not shown in the listing).

4.1.8 Pretty printing

SableCC comes with handy autogenerated `toString` routines. In OCaml these routines have to be written by hand. For pretty-printing the AST there is a design choice: one can use OCaml's standard pretty-printing library, Format, or write one by hand. Whereas the first choice would be preferable from a software engineering angle, we fear its syntax would set too high a bar for beginners in OCaml.

4.1.9 Subtleties

For both languages we encountered a number of subtleties and surprises.

In OCaml the evaluation order is unspecified, however in practice both implementations try to adhere to a (somewhat unusual) right-to-left evaluation order following Caml's Zinc virtual machine design [22]. This may lead to surprises, e.g., if one traverses a list of methods, `m::ms` without explicitly dictating evaluation order using let bindings:

```
(visit_method_decl m)::(visit_method_decls ms)
```

If both `m` and `ms` contain errors, the resulting traversal rejects the program with a reference to the last error (in `ms`), which is usually not what was intended.

In AspectJ the combination of auto-generated visitor code and replacing sub-trees by mutation can lead to surprising results as well: if the visitor pattern code keeps a reference to substituted sub-trees, e.g., in an iterator, the visitor pattern will suddenly traverse the old, already replaced sub-tree, thereby causing unexpected behavior.

Unlike OCaml, the tree representation auto-generated by SableCC is doubly-linked with parent and child nodes containing references to each other. This representation however forbids sharing of sub-trees: as a node can only reference a single parent, if one sub-tree is inserted in several positions the parent reference will only be valid for one of the positions. To preserve this tree invariant one has to explicitly `clone()` the sub-tree and insert the copy instead. Since this implicit tree invariant is not statically guaranteed, it is up to tests to catch a missing `clone()` invocation.

In many languages it is a common source of errors to confuse reference (address) equality with structural equality. In OCaml this shows up when comparing data, e.g., variable or type names, using address comparison operators == and != instead of the structural comparison operators = and <>. In AspectJ and Java the same issue shows up in comparisons that use reference comparison == instead of a suitably overridden `equals` method. In the presence of sub-typing, `equals` is itself prone to comparing incomparable types as all `equals` methods override the `Object` comparison from `java.lang.Object`, which can lead to tests that are always false. Prior work by the third author [38] suggests a refinement of Java's type system to address this issue.

4.2 System and tools

We now turn to an evaluation of the system-oriented parameters.

4.2.1 Libraries

For a non-trivial software project such as the present one, it is essential to be able to build upon existing libraries for common tasks rather than having to start from scratch. In the present case, both implementations rely on a library for parsing Java's `.jar` and class files. This enables the implementations to compile up against the Java standard library. Since the `.jar` format is essentially a zip-archive, available libraries for parsing those are required too.

The AspectJ implementation uses *BCEL* (the Byte Code Engineering Library, now part of the Apache Project) [5] for parsing class files from the Java standard library. The OCaml implementation uses Javalib [4]. We found both libraries to be mature and well-documented and hence up for the task. Alternatives to the above libraries exist, e.g., the more recent Barista library for OCaml [7]. However we have found no need to pursue such alternatives.

In both cases, these library dependencies have been made easily available through established package systems on the different platforms, e.g., Cygwin,

```
%{ %}
%token A
%start x
%type <unit> x
%%
x   :           { }
    |  A x      { }
    |  A A x    { };
```

(a) The example in ocamlyacc

```
Tokens
  a = 'a';

Productions
  x = {empty}
    | {one} a x
    | {two} [first]:a [second]:a x;
```

(b) The example in SableCC

Figure 9: Example of an ambiguous grammar

MacPorts and APT/RPM, or shipped as part of the project skeleton, e.g., as jar files for the various Java libraries. Furthermore, for the OCaml project, the Makefile for building the skeleton contains a special "libs" target that will download and install all remaining libraries that were not installed by the host's package management system. For subsequent editions of the course we would consider the recent OPAM package management system for OCaml [28].

4.2.2 Lexing and parsing

For the OCaml implementation we used ocamllex (a lexer generator) and ocamlyacc (a parser generator) both included in the OCaml distribution, whereas for the AspectJ implementation we used (a locally branched version [37] of) SableCC [10] — a combined lexer and parser generator which also auto-generates classes representing the syntax tree along with methods implementing generic tree traversals via visitor patterns.

The error messages from ocamlyacc (shift-reduce, reduce-reduce conflicts) leaves a bit to be desired compared to SableCC. For example, consider the ambiguous grammars in Figure 9a and Figure 9b corresponding to the following context-free grammar (writing Λ for the empty string):

$$x \ ::= \ \Lambda \ | \ \mathtt{A}\ x \ | \ \mathtt{A}\ \mathtt{A}\ x$$

When fed to ocamlyacc the error message included in Figure 10 indicates the problem in terms of the automaton to which the grammar has been compiled without indicating the context which leads to said problem. The error messages of SableCC have been improved to be more informative: as the example error in Figure 11 illustrates, the error message now includes information about the parser's stack, which reflects a derivation in the input grammar. The error messages of ocamlyacc are not as descriptive and hence more on par with the original SableCC conflict error messages. For the second iteration of the OCaml version of the course, we therefore switched parser generator from ocamlyacc to Menhir [29]. To a large extent, the switch was seamless, as the input format of the two tools is largely compatible. Overall, Menhir provides better error messages on par with the patched SableCC for the benefit of the students debugging

```
   0  $accept : %entry% $end

   1  x :
   2    | A x
   3    | A A x

   4  %entry% : '\001' x

...

7: reduce/reduce conflict (reduce 2, reduce 3) on $end
state 7
	x : A x .  (2)
	x : A A x .  (3)

	.  reduce 2


Rules never reduced:
	x : A A x  (3)


State 7 contains 1 reduce/reduce conflict.


4 terminals, 3 nonterminals
5 grammar rules, 8 states
```

Figure 10: Output error from ocamlyacc (excerpt from `parser.output`)

```
SableCC version 3.2/daimi-201008261629

...

class java.lang.RuntimeException:

reduce/reduce conflict in state [stack: TA TA PX *] on EOF in {
	[ PX = TA PX * ] followed by EOF (reduce),
	[ PX = TA TA PX * ] followed by EOF (reduce)
}
class java.lang.RuntimeException:

reduce/reduce conflict in state [stack: TA TA PX *] on EOF in {
	[ PX = TA PX * ] followed by EOF (reduce),
	[ PX = TA TA PX * ] followed by EOF (reduce)
}
```

Figure 11: Output error from (patched) SableCC

shift-reduce and reduce-reduce conflicts. For example, on the input from Figure 9, Menhir produces the error in Figure 12, which reflects the ambiguity in terms of the input grammar.

```
** Conflict (reduce/reduce) in state 3.
** Token involved: #
** This state is reached from x after reading:

A A x

** The derivations that appear below have the following common factor:
** (The question mark symbol (?) represents the spot where the
** derivations begin to differ.)

x // lookahead token is inherited
(?)

** In state 3, looking ahead at #, reducing production
** x -> A x
** is permitted because of the following sub-derivation:

A x // lookahead token is inherited
  A x .

** In state 3, looking ahead at #, reducing production
** x -> A A x
** is permitted because of the following sub-derivation:

A A x .
```

Figure 12: Output error from Menhir (`parser.conflicts`)

A range of alternative lexer and parser tools exist, e.g., JLex and ANTLR for Java/AspectJ, and uLex for OCaml. No such alternatives have been pursued at this point.

While developing the Joos lexer with ocamllex we encountered the following error message:

```
ocamllex: transition table overflow, automaton is too big
```

The problem is that with too many keywords in the language, the size of the generated lexer automaton explodes. This is however such a common error that the OCaml manual [23] suggests a work-around: to keep a separate hashtable of keywords and scan all identifiers with a single regular expression, e.g.,

```
['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_']*
```

and subsequently look up the identifier in the hashtable to determine whether the present token is a keyword or an identifier.

In the initial porting of the grammar from SableCC to ocamlyacc we accidentally used the identifier `type` as a non-terminal in the grammar fed to ocamlyacc. This was unfortunately not caught by ocamlyacc even though `type` is a keyword in OCaml. Instead we received a subsequent error from the OCaml compiler as the auto-generated parser did not itself parse. Historically ocamlyacc is derived from the Berkeley Yacc implementation, with the original C backend replaced by an OCaml backend. As such it does not check for such conflicts.

	AspectJ		OCaml	
Time (sec)	Total	SableCC*	native	bytecode
Best	62.67	25.21 + 6.56	25.51	15.47
Average of 3 runs	64.51	25.51 + 6.63	25.71	15.50

* Column reports times to invoke SableCC twice — once for the Joos2 parser (first number) and once for the Peephole DSL parser (second number)

Table 3: Compile times for a full build of the two implementations

The Menhir parser generator, which was developed more recently, performs such checks.

Space-wise, the handwritten OCaml code for generating a lexer and a parser dominates the corresponding SableCC specification. First SableCC uses a single file format for both the lexer and the parser and thereby enjoys the sharing of a common token definition. Second by studying the numbers in Table 2, the difference is greatest in the description of the lexers: the lexer specification in ocamllex takes 210 source lines, whereas the lexer section of the entire SableCC specification takes only 38 source lines. This difference is mostly due to a triple duplication of the OCaml token definition: one occurrence declares the token data type, one token-to-string function dispatches on the token data type, and finally the lexer itself is defined by a regular expression for each token in the data type. In contrast, SableCC generates code for the former two automatically, thus saving a factor of three. Third, the slight size difference in the parser specifications is due to SableCC's support for *Extended BNF* (EBNF) grammars, which includes common regular expression shorthands for options (`A?`), non-empty lists (`A+`), and possibly empty lists (`A*`). Again ocamlyacc does not support these for historic reasons, however Menhir does.

4.2.3 Compiling and building

OCaml is well-known for its fast compiler, which results in a very short edit-compile-run development cycle. Table 3 reports build times to build each implementation from scratch. The reported byte code build times for OCaml to some extent confirms the above understanding.

The compile times for the AspectJ implementation is clearly dominated by the time to run the SableCC parser generator (reported in a separate column). The times to run ocamllex and ocamlyacc are negligible (0.10 sec for ocamllex, and 0.08 sec for ocamlyacc, on average). When subtracting the lexer and parser generation times of SableCC, the build times are more comparable (about 30 seconds for AspectJ and about 25 seconds for native OCaml).

The AspectJ implementation can be built with Eclipse's built-in compilation system. As an alternative, we have provided students with a vanilla `Makefile` as part of the skeleton. The OCaml implementation uses ocamlbuild (a dedicated build tool included with the OCaml distribution) which we invoke with suitable arguments from a plain `Makefile`. Initially we used ocamldep (a dedicated dependency analyser included with the OCaml distribution) to manually generate suitable Makefiles for the compilation process, however ocamlbuild deals with dependency analysis automatically and reliably.

Both of the above tools have their advantages: for AspectJ, integrated building in Eclipse is a plus for development. Similarly for OCaml, Makefiles integrate well with other tools, e.g., the Emacs editor.

4.2.4 Execution

The OCaml implementation primarily makes use of immutable data structures. These structures can in some cases slow things down, e.g., by a logarithmic factor for search trees. On the other hand, a common idiom of the AspectJ implementation, described in Section 4.1.6, is to walk parent pointers up to the AST root. This walk is avoided in the OCaml implementation by instead threading (passing around) the local environment. Another difference in execution comes from the dynamic optimization techniques used by the JVM, as well as its ability to cache static structures for fast initialization time on subsequent runs.

We have measured the compilation time using the AspectJ implementation on four different configurations of the underlying Java runtime:

1. the client mode is the default mode of the runtime system;
2. the server mode, invoked with `-server`, performs additional compilation during startup;
3. the interpreter mode, invoked with `-Xint`, disables dynamic compilation; and
4. the client mode without class caching, invoked with `-Xshare:off`, disables the caching of class structures to disk.[7]

In almost all cases, the compilation times of the server mode, the client mode, and the client mode without caching show no statistically significant difference in performance, measured at 95% confidence. For completeness, we have included the full data table in A, Table 5. The Java runtime in interpreter mode, i.e., without dynamic compilation, is consistently slower than the other Java modes, but not by much. The AspectJ implementation performs comparably to Sun's (now Oracle's) Java compiler, but for some of the larger programs it is up to a factor of two slower, e.g., for the czero and grammar benchmarks (see A, Table 5).

We have measured the compilation time using the OCaml implementation for both the compiler produced by the OCaml byte-code compiler and the compiler produced by the OCaml native code compiler. Our measurements confirm the folklore that OCaml performs quite predictably (cf. the standard deviation column of OCaml native in A, Table 5). Overall the byte-code compiled version is

[7] *Class data sharing* [35] is a JVM technique that saves internal class representations to disk and uses memory mapping to quickly recreate them on subsequent runs

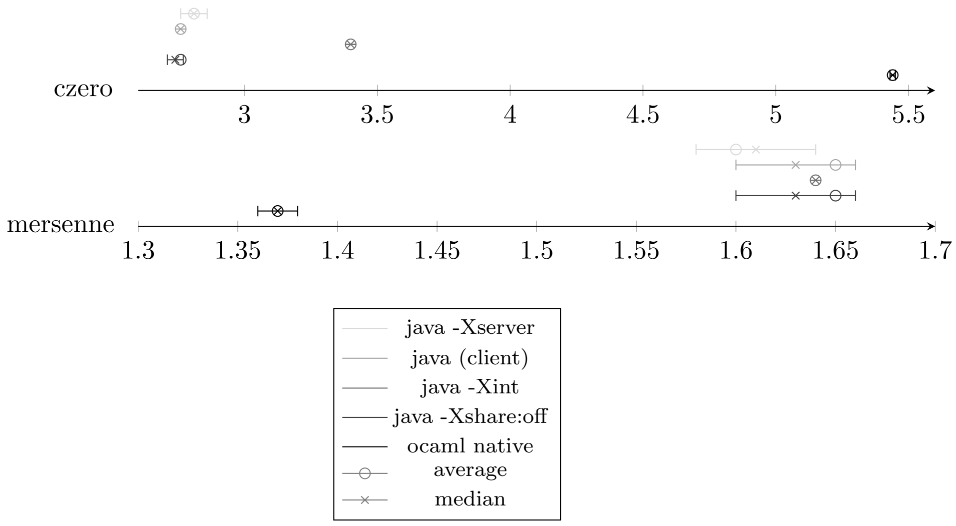

Figure 13: Compile times of Joos compilers on the czero and mersenne benchmarks

a factor of two behind the native-code compiled version, which in turn is within a factor two of the AspectJ implementation. For small input programs, the overhead of starting the OCaml native code is faster than the AspectJ startup, e.g., for the helloworld, helloworld2, int2str, mandelbrot, mersenne, and ping-pong benchmarks (see A, Table 5). However, once the JVM is up and running the statically optimized OCaml cannot keep up. Another explanation is the maturity of the AspectJ implementation which has been refined over the years. We expect this gap to narrow as the OCaml implementation also matures.

Figure 13 displays two selected benchmarks in more detail. The czero benchmarks shows how, for a larger benchmark, the Java and AspectJ implementations outperform the OCaml implementation in terms of compilation speed. The three Java configurations with dynamic compilation perform comparably, while the interpreted mode is a bit slower and the OCaml native code is a factor of two slower. The mersenne benchmarks shows how, for a small benchmark, the startup overhead of the Java runtime becomes noticeable. Here OCaml is the fastest and the three Java configurations with dynamic compilation show some unpredictability as seen by the relatively large standard deviations.

4.2.5 Debugging

Java ships with JDB, the Java Debugger, and its integration with, e.g., Eclipse, provides a reasonable debugging environment.

OCaml ships with `ocamldebug` which can be used to debug a byte-code compiled OCaml program. It provides all the standard features for stepping and inspecting the code. It also allows hooking in OCaml printing routines to allow improved pretty-printing of values. Initially the debugger could not be used on Windows due to a bug in the Cygwin Flexlib package which OCaml depends upon. This error has been fixed in later version of Flexlib.

4.2.6 Documentation

For documenting the two implementations, we have used the de-facto tools for the platform: Javadoc for Java/AspectJ and ocamldoc for OCaml. Both tools work reasonably, but we found neither were perfect for the job. Javadoc does not support the AspectJ extensions to Java. As a consequence the documentation has been generated by temporarily rewriting AspectJ constructs into Java ones, e.g., `aspect` into `class`, and running Javadoc on the temporary files.

The ocamldoc tool makes many assumptions about how and where code is commented. We would prefer to place all comments with the implementation code even when we also specify an interface, since the implementation code is what the students will be editing. The documentation build uses the OCaml compiler to obtain symbols and type information. This is useful but also complicates generation of documentation for "non-existing" code, e.g., the lexer and parser specifications. Also we found the generation time to be very long. We have not pursued alternatives at this point, e.g., the enhanced ocamldoc tool, Argot [6]. For a project such as this compiler project it might also be worth considering an alternative type of documentation tool all together, e.g., literate programming [20, 30] or out-of-source documentation.

4.2.7 Development environments

There are a multitude of editors and development environments. For the AspectJ project, a standard Java setup works and we do not cover their respective merits here.

For the OCaml project, being a bit more esoteric, we have considered two setups specifically: Emacs using Tuareg Mode and Eclipse using the OCaml plugin OcaIDE. Both Emacs and Eclipse are widely used and the Tuareg and OcaIDE extensions appeared to be the most complete OCaml extensions for the respective editors.

Emacs and Tuareg Mode Installation is straightforward: one has to fetch and unpack the tuareg archive and paste a few setup lines into one's `.emacs` configuration file. Development of OCaml within Emacs follows mostly the same workflow as with other major Emacs modes for programming. When editing in tuareg-mode, a keyboard shortcut will start an interactive top-level; and another keyboard shortcut will invoke a compilation command, here the default `make -k` will result in building the compiler with default arguments. Furthermore compilation errors are formatted in accordance to Emacs style which makes it possible to easily navigate between them.

Eclipse and OcaIDE Installation is straightforward using Eclipse's existing plugin installation infrastructure. After installing, Eclipse might need to be told where the OCaml and Make executables and library files are located. Finally, Eclipse users have two choices for configuring the build setup: (1) they can use the Makefile-managed project which will simply use our existing build infrastructure; or (2) they can use the ocamlbuild-managed project. Option (1) does not provide full development environment support, e.g., for the debugger and top-level. However, option (2) requires manually configuring actual library locations for the compiler and linker. At the time of writing, this setup cannot be done in a system-agnostic and reusable way.

A frustration we have with the Eclipse system is that some of its configuration is located outside of the project directory and prohibits us from simply shipping this setup to the students as part of the source archive.

4.3 Summary

Both implementations have satisfied the project objectives. Both programming languages have sufficient support across the main platforms. The languages have the necessary libraries and provide the necessary tools to work effectively. By design, the main programming concepts are within reach of the course students—witness the eight successive instances of our compiler course. Based on the total number of lines of the two implementations and experience running the course, we also perceive both projects as equally challenging development-wise. The main differences to draw from our evaluation are:

Efficiency of the build setup. The build times of our implementations confirm the reputation of the OCaml native-code compiler and byte-code compiler as being efficient. Both provide a faster turnaround time compared to compiling the AspectJ implementation.

Efficiency of the compilers. Our evaluation of the execution speed of the compilers, i.e., the time it takes to compile a Joos program, shows an overall win for the AspectJ implementation by a factor of two. There are two main suspects that might cause this deficiency in the OCaml implementation:

- **Garbage-collection time.** The OCaml implementation is based on the *transformational compiler model*, where each phase constructs a new tree leaving the previous as garbage. This can lead to increased garbage collection time, e.g., if the compiler code retains a pointer to an old tree thus forcing OCaml's tracing garbage collector to preserve it, and any additional data structures it (transitively) refers to.
- **General lack of optimization.** The AspectJ implementation has been refined over six years which is not yet the case for the more recent OCaml implementation. Indeed, the current implementation still uses a few inefficient data structures, such as linked lists to represent sets.

Investigating and optimizing the compilers is an ongoing effort and we expect to narrow the gap between the two as the OCaml implementation matures.

Reliability of the compilers. Our evaluation shows some benefit in the expressiveness of specifications in the OCaml implementation. This expressiveness is due both to the static type system of OCaml, which allows a strict specification of each phase, as well as the transformational compiler model, which allows characterizing the input and output at a finer level of granularity. The static specification guides and in some places forces a particular implementation.

5 Related work

We split our discussion of related work in three: first we compare to previous work concerned with language comparison, second we compare to previous work related to the so-called *expression problem* and extensible compiler frameworks, and third we compare to previous work on general compiler writing.

5.1 Language and language feature comparisons

Hudak and Jones [16] report on an experiment of prototyping a "Geometric Region Server" in a number of different languages, including Haskell, Ada, C++, and Awk. Their comparison confirms the impression of functional programs as being compact: solutions range from the authors' Haskell solution of only 85 LOC to 1105 LOC for the reported C++ solution. The present case study, which is orders-of-magnitude larger, does not confirm this impression. This could be due to several factors: our mostly first-order OCaml implementation, the expressiveness of AspectJ combined with SableCC, or maybe some inherent property of our project and its modularity requirements.

Hartel et al. [14] compare an impressive range of functional programming language implementations on the same program, *Pseudoknot*, a non-trivial application of approximately 3000 LOC taken from molecular biology. The comparison focuses primarily on compile time and execution time of this floating-point sensitive program. As in the present case, the Pseudoknot benchmark program was translated from one language into the next (starting with a version in Scheme). The comparison comes with a number of caveats: for example, some implementations use single precision floating point numbers whereas others use double precision floating point numbers — a difference which can affect the comparison of run times. Nevertheless, with suitable optimization, optimizing implementations of functional languages could approach the performance of C for such numeric applications more than 15 years ago.

The *Computer Language Benchmarks Game* [3] (formerly known as the *Great Computer Language Shootout*) is a community effort to benchmark different programming language implementations against each other. Starting from one

class \ method	weeding	disambiguation	type checking	definite assignment
local	·	·	·	·
non-static field	·	·	·	·
array index	·	·	·	·
static field	·	·	·	·

Table 4: A dispatch matrix on lvalues

man's desire to compare scripting languages, the benchmarks game has grown increasingly popular and spurred programming language communities to improve their benchmark performance in order to make their language of choice more attractive. At the time of writing, the benchmarks game compares execution time, memory usage, and CPU load of 14 different small, but non-trivial applications written in 27 different languages (and running on even more implementations) across four different architectures. Like us, the benchmarks-game implementers notice a small, but significant difference in Java's execution time when comparing the first and subsequent runs of the Java VM.

Minsky and Weeks [26] describe the Jane Street Capital company's usage of OCaml for high-frequency trading on Wall Street. They highlight the terseness of the language, the strong guarantees they obtain from the language's type system, as well as the predictable behavior of the compiler. On the other hand they criticize the language for lacking generic operations, e.g., for printing arbitrary OCaml values, and for lacking tools for programming in the large. Admittedly of course, our 10 KLOC comparison is far from the 1-2 MLOC in production at Jane Street Capital.

5.2 The expression problem and extensible compilers

Reynolds [31] originally pointed out the tension between programs that are easy to extend with additional data vs. programs that are easy to extend with additional operations. The tension was later popularized by Wadler as '*the expression problem*'. Harper classifies this problem as the representation choice of a dynamic dispatch table and the inherent duality of the problem [13]. Consider again the grammar of *lvalues* from Figure 1. Here we have four classes representing the four productions of lvalues. The compiler implementation consists of several passes over lvalues, e.g., weeding, disambiguation, type checking, and definite assignment. The product of these two sets populates a dispatch matrix as illustrated in Table 4. We might represent this table by rows or by columns. In Java we represent a row as a Java class and each method as a Java method on that class. Extending the table with a new row is easily done by locally defining a new class of lvalues. Extending the table with a column is difficult since we must separately add a new method to each existing class of lvalues. In OCaml we represent a column as an OCaml function that case dispatches on the classes of lvalues. Extending the table with a new column is easily done by locally defining a new function over lvalues. Extending a row is difficult since we must separately add a new case to each existing function over lvalues.

In our particular context the ability to add new columns, e.g., new compiler passes, is needed most. For the same reason the AspectJ implementation uses double dispatch (through the Visitor Pattern [11]) to locally define a compiler pass and uses aspects to add new information to the AST classes. The OCaml implementation simply defines a new AST data type whenever information is added or removed from the AST. In both cases, this somewhat static definition of the AST is reasonable and even desirable because it makes many aspects of the compiler transparent. The OCaml data types are closed and fixed and we don't use AspectJ to extend the types of AST nodes beyond those auto-generated by SableCC. However, AspectJ is capable of defining new classes within the existing class hierarchy as well as adding new methods and fields to existing classes, which makes it a viable candidate for use in an *extensible* compiler.

Polyglot [27] is an extensible compiler framework for the Java programming language, which can be extended with both new AST node types and with new compiler passes. It has furthermore been designed with *extension scalability* in mind, meaning that the implementation effort of extensions should vary with the size of the language extension. The Polyglot framework includes a dataflow analysis engine, which comes with implementations of the reachability and definite assignment analyses similar to the present compilers. Like the OCaml implementation (and in contrast to the AspectJ implementation), the Polyglot framework is functional: it does not perform destructive updates to the abstract syntax tree.

Ekman and Hedin [8] present another extensible Java compiler, *JastAddJ*. JastAddJ is itself written in *JastAdd*, a declarative, aspect-oriented, domain-specific programming language based on attribute grammars. The JastAddJ compiler is structured as a Java 1.4 compiler which is extended to a Java 1.5 compiler using *attribute grammar extensions*, including the substantial addition of generic types and wildcards to the type system. The compiler is implemented with the AST as the only data structure. As such, its type analysis is represented by a *reference attribute*, similar to our AspectJ implementation.

JastAdd shares some of the aspect-oriented advantages with our AspectJ implementation in the form of keeping related code grouped together. It furthermore shares some of the declarative advantages with our OCaml implementation by keeping reference attributes to a minimum. Ekman and Hedin [8] report source-lines-of-code (SLOC) for their implementation which are comparable with those of the present paper — albeit for a full Java 1.4 compiler. These should be taken with a grain of salt though, as (a) their domain-specific language leaves the order of attribute evaluation unspecified, and (b) the JastAdd system caches many attributes, both of which may make code with explicit evaluation order and caching come out worse. They do not report on the lines-of-code generated by the JastAdd system.

5.3 Compiler work

As previously mentioned, the OCaml data types were inspired by the OCaml compiler's own AST representation [23]. This data type representation is in itself close to that of the *Standard ML of New Jersey* compiler [2], and as such reasonably traditional within the functional programming community. The encoding of additional invariants in the types themselves bears similarities to the typed intermediate languages of the type-directed TIL compiler [36].

A number of other papers discuss design issues in a compiler course. In the nano-pass framework of Sarkar, Waddell, and Dybvig [32], the authors argue that several minimalistic passes over a syntax tree is preferable to fewer "monolithic" passes. Even though Scheme is dynamically typed, the framework allows to define input and output languages for each pass (specified as BNF grammars), and (1) have boilerplate traversal code auto generated, (2) have the framework ensure that each pass conforms to the specification and, (3) ensure that the pipeline input and output languages match up. The auto-generated tree traversal code of the nano-pass framework is comparable to the auto-generated visitor code by the SableCC parser generator. On the other hand, the nano-pass framework's static checks are comparable to the static guarantees provided by the tree types of the OCaml implementation. Keep and Dybvig [17] later make the case for the nano-pass framework in an industrial setting of the commercial Chez Scheme compiler.

Schwartzbach [33] describes the overall design decisions behind the compiler course based on the original AspectJ implementation by the first author. Schwartzbach furthermore argues that students can appreciate formal notation such as inference rules, as they distill the essence of a complex, prose-based language specification. In contrast, we focus on comparing and contrasting the two reference implementations.

6 Conclusion and perspectives

In this case study, we have compared and contrasted two implementations of the same software project in an attempt to give a non-trivial apples-to-apples comparison. The resulting analysis is finer than a micro-benchmark shootout yet big enough to reflect real-world practice.

Our entire code base shows that functional code (OCaml) is more concise than the corresponding aspect-oriented code (AspectJ). However, a large fraction of the aspect-oriented code is auto-generated, and the parts we wrote by hand have about the same size.

The aspect-oriented implementation follows the "Christmas tree" model, where the abstract-syntax tree is mutable and updated sequentially, and the functional implementation follows the transformational model, where the abstract-syntax tree is immutable and successive versions are created sequentially. The functional implementation provides fast response times in the best case and a slowdown by a factor of two in the worst case. Its performance is predictive. The overhead of the additional tree copying is to some extent outweighed by tree sharing at design time and by OCaml's well-tuned garbage collector at run time.

Each of AspectJ and OCaml has its advantages and disadvantages. Each phase of the OCaml implementation has a statically guaranteed interface, but this static guarantee induces redundant data-type declarations and corresponding traversal code. The visitor patterns of Java and AspectJ prevent much of this redundancy, but the interface of each phase of the AspectJ implementation must be checked dynamically. Also, the visitor pattern may interact poorly with mutable abstract-syntax trees. Naturally, one could mirror each approach in the other, e.g., by using only a single data type throughout the phases of the OCaml implementation, along with a uniform fold function for traversing the corresponding abstract-syntax tree. We expect such an approach to reduce the amount of hand-written OCaml code below that of the AspectJ implementation.

In conclusion, from a language perspective, each of the two implementations represents a tradeoff between flexibility and static guarantees. The flexibility of the AspectJ implementation was helpful in creating the first version of the project in an exploratory manner. Once the design had stabilized, the static guarantees of the OCaml version were helpful in the teaching context to guide students and to isolate the effects of each phase, thereby ensuring stability of the outcome. From a systems perspective, both platforms come with their quirks and surprises which hinder a completely smooth user experience. As such, there is no clear winner. The two implementations demonstrate that non-trivial and modular software can be obtained by combining a reasonable amount of code with auto-generated (boiler-plate) code and available libraries.

A Compile time table

Benchmark	LoC	Files	Java			AspectJ												OCaml					
			javac			server			client			interp			noshare			native			byte		
			mid	avg	dev	mid	avg	dev	mid	avg	dev	mid	avg	dev	mid	avg	dev	mid	avg	dev	mid	avg	dev
anagram	100	1	1.28	1.24	0.09	1.79	1.78	0.03	1.75	1.83	0.15	1.79	1.79	0.00	1.76	1.77	0.03	5.75	5.75	0.01	12.27	12.26	0.04
brainfuck	314	7	1.35	1.33	0.05	2.00	2.02	0.08	2.00	2.02	0.07	2.14	2.14	0.00	2.10	2.08	0.03	4.48	4.49	0.02	9.93	9.93	0.01
container	32	1	1.33	1.32	0.03	1.54	1.54	0.02	1.59	1.58	0.03	1.60	1.61	0.03	1.55	1.55	0.01	1.93	1.93	0.01	4.59	4.59	0.01
cpr	188	4	1.39	1.34	0.09	1.90	1.88	0.03	1.85	1.85	0.04	1.89	1.89	0.00	1.85	1.86	0.03	2.63	2.63	0.01	5.53	5.53	0.01
crypto	275	6	1.23	1.26	0.05	1.95	1.97	0.03	1.95	1.96	0.03	1.99	1.99	0.00	1.95	1.98	0.06	4.43	4.43	0.00	9.49	9.48	0.02
czero	791	1	1.42	1.42	0.02	2.81	2.81	0.05	2.76	2.76	0.00	3.40	3.40	0.00	2.76	2.74	0.03	5.44	5.44	0.01	11.68	11.68	0.03
fib	99	3	1.23	1.20	0.08	1.65	1.65	0.00	1.64	1.65	0.05	1.69	1.68	0.03	1.70	1.66	0.06	2.73	2.74	0.01	5.83	5.84	0.03
grammar	570	18	1.54	1.49	0.12	2.76	2.77	0.03	2.80	2.78	0.03	3.30	3.30	0.00	2.80	2.82	0.03	5.86	5.86	0.00	12.88	12.85	0.05
helloworld	19	1	1.13	1.12	0.11	1.39	1.41	0.03	1.40	1.40	0.00	1.44	1.44	0.00	1.40	1.39	0.00	1.03	1.03	0.00	2.24	2.24	0.01
helloworld2	25	1	1.08	1.08	0.03	1.39	1.41	0.03	1.39	1.40	0.00	1.49	1.50	0.01	1.45	1.44	0.00	1.09	1.09	0.00	2.37	2.37	0.01
int2str	31	1	1.10	1.09	0.08	1.44	1.43	0.03	1.40	1.41	0.03	1.49	1.49	0.00	1.44	1.43	0.03	1.11	1.11	0.00	2.35	2.35	0.01
joos1	531	17	1.30	1.29	0.02	2.20	2.17	0.06	2.20	2.18	0.07	2.19	2.19	0.00	2.10	2.11	0.03	2.39	2.40	0.01	5.23	5.35	0.22
joos2html	301	1	1.16	1.20	0.07	2.36	2.24	0.20	2.15	2.13	0.08	2.19	2.20	0.00	2.05	2.07	0.08	3.65	3.64	0.02	7.65	7.65	0.03
mandelbrot	38	1	1.12	1.12	0.01	1.44	1.45	0.00	1.45	1.45	0.00	1.49	1.49	0.00	1.44	1.43	0.03	1.02	1.02	0.00	2.22	2.22	0.00
matrix	320	5	1.20	1.24	0.09	1.95	1.96	0.08	2.00	2.01	0.08	2.05	2.03	0.03	2.00	2.00	0.05	2.33	2.33	0.01	5.53	5.53	0.03
mersenne	104	4	2.13	2.14	0.02	1.60	1.61	0.03	1.65	1.63	0.03	1.64	1.64	0.00	1.65	1.63	0.03	1.37	1.37	0.01	2.95	2.95	0.01
pingpong	15	1	1.89	1.93	0.10	1.39	1.39	0.00	1.39	1.38	0.03	1.44	1.44	0.00	1.39	1.38	0.03	0.80	0.80	0.00	1.83	1.82	0.00
salary	226	5	1.95	1.92	0.07	1.70	1.71	0.03	1.70	1.70	0.00	1.74	1.74	0.00	1.70	1.70	0.00	2.25	2.25	0.01	4.86	4.86	0.02
scheme	975	30	2.77	2.78	0.01	2.60	2.63	0.06	2.60	2.64	0.11	3.35	3.35	0.00	2.60	2.58	0.03	3.70	3.70	0.02	7.86	7.87	0.07
search	124	4	2.13	2.14	0.04	1.70	1.71	0.03	1.75	1.73	0.03	1.75	1.76	0.03	1.70	1.71	0.03	2.18	2.19	0.02	4.59	4.59	0.00
strategy	205	4	2.10	2.11	0.07	1.75	1.75	0.00	1.75	1.75	0.04	1.79	1.80	0.00	1.74	1.75	0.05	2.47	2.48	0.02	5.16	5.15	0.02
sudoku	164	1	2.06	2.07	0.12	1.80	1.83	0.06	1.85	1.83	0.02	1.89	1.89	0.00	1.76	1.77	0.03	5.01	4.99	0.02	10.57	10.57	0.05
temp	68	4	2.10	2.10	0.01	1.55	1.55	0.00	1.55	1.55	0.00	1.59	1.58	0.03	1.54	1.55	0.05	2.21	2.21	0.01	4.58	4.63	0.10
turing	151	1	2.19	2.20	0.09	1.84	1.87	0.08	1.80	1.87	0.17	1.85	1.85	0.00	1.85	1.83	0.03	5.34	5.34	0.02	11.79	11.80	0.01
geo. avg.			1.53			1.82			1.81			1.90			1.81			2.51			5.44		

Table 5: Compile times of resulting Joos compilers

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. World Student Series. Addison-Wesley, Reading, Massachusetts, 1986. 2, 10

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, New York, 1992. 34

[3] Doug Bagley, Brent Flugham, and Isaac Gouy. The computer language benchmarks game. `http://benchmarksgame.alioth.debian.org/` [10 November 2014]. 5, 31

[4] Nicolas Barré, Nicolas Cannasse, Laurent Hubert, Vincent Monfort, and Tiphaine Turpin. Javalib, 2010. `http://sawja.inria.fr/` [10 November 2014]. 22

[5] Dave Brosius, Torsten Curdt, Markus Dahm, and Jason van Zyl. The byte code engineering library BCEL, 2011. `http://commons.apache.org/proper/commons-bcel/` [10 November 2014]. 22

[6] Xavier Clerc. Argot, 2012. `http://argot.x9c.fr/` [10 November 2014]. 29

[7] Xavier Clerc. Barista, 2012. `http://barista.x9c.fr/` [10 November 2014]. 22

[8] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22nd ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '07, pages 1–18, 2007. doi: 10.1145/1297027.1297029. 33

[9] Pascal Fradet and Daniel Le Métayer. Compilation of functional languages by program transformation. *ACM Transactions on Programming Languages and Systems*, 13:21–51, 1991. doi: 10.1145/114005.102805. 10

[10] Étienne Gagnon. SableCC, 1998. `http://sablecc.org/` [10 November 2014]. 11, 23

[11] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. 6, 33

[12] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley, Boston, MA, USA, 2nd edition, 2000. 4, 17, 19

[13] Robert Harper. *Practical foundations for programming languages*. Cambridge University Press, 2013. 32

[14] Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. Van Groningen, Kevin Hammond, Bogumil Hausman, Melody Y. Ivory, Richard E. Jones, Jasper Kamperman, Peter Lee, Xavier Leroy, Rafael D. Lins, Sandra Loosemore, Niklas Rjemo, Manuel Serrano, Jean-Pierre Talpin, Jon Thackray, Stephen Thomas, Pum Walters, Pierre Weis, and Peter Wentworth. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *Journal of Functional Programming*, 6(04):621–655, 1996. doi: 10.1017/S0956796800001891. 31

[15] Paul Hudak. Modular domain specific languages and tools. In *Proceedings of the Fifth International Conference on Software Reuse*, pages 134–142, June 1998. 12

[16] Paul Hudak and Mark P. Jones. Haskell vs. Ada vs. C++ vs. Awk vs. ... An experiment in software prototyping productivity. Research Report YALEU/DCS/RR-1049, Department of Computer Science, Yale University, New Haven, CT, Oct 1994. 31

[17] Andy Keep and R. Kent Dybvig. A nanopass framework for commercial compiler development. In Greg Morrisett and Tarmo Uustalu, editors, *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming (ICFP'13)*, pages 343–350, Boston, MA, Sep 2013. doi: 10.1145/2500365.2500618. 34

[18] Richard A. Kelsey and Paul Hudak. Realistic compilation by program transformation. In Michael J. O'Donnell and Stuart Feldman, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, January 1989. doi: 10.1145/75277.75302. 10

[19] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968. doi: 10.1007/BF01692511. 19

[20] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2): 97–111, 1984. doi: 10.1093/comjnl/27.2.97. 29

[21] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. Orbit: An optimizing compiler for Scheme. In Stuart I. Feldman, editor, *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. doi: 10.1145/12276.13333. 10

[22] Xavier Leroy. The Zinc experiment: an economical implementation of the ML language. Rapport Technique 117, INRIA Rocquencourt, Le Chesnay, France, February 1990. 21

[23] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, documentation and user's manual – release 3.12.1.* INRIA, July 2011. http://caml.inria.fr/pub/docs/manual-ocaml-312/ [10 November 2014]. 10, 16, 25, 33

[24] Jon Meyer and Troy Downing. *Java virtual machine.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1997. ISBN 1-56592-194-1. 2

[25] Jonathan Meyer. Jasmin, 1996. http://jasmin.sourceforge.net/ [10 November 2014]. 2, 4

[26] Yaron Minsky and Stephen Weeks. Caml trading – experiences with functional programming on Wall Street. *Journal of Functional Programming*, 18(4):553–564, 2008. doi: 10.1017/S095679680800676X. 32

[27] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In Görel Hedin, editor, *CC'03: Proceedings of the 12th International Conference on Compiler Construction*, volume 2622 of *Lecture Notes in Computer Science*, pages 138–152, Warsaw, Poland, April 2003. Springer-Verlag. doi: 10.1007/3-540-36579-6_11. 33

[28] OCamlPro, 2013. OCaml package manager OPAM, 2013. http://opam.ocamlpro.com/ [10 November 2014]. 23

[29] Franois Pottier and Yann Régis-Gianas. Menhir, 2012. http://gallium.inria.fr/~fpottier/menhir/ [10 November 2014]. 23

[30] Norman Ramsey. Literate programming simplified. *Software, IEEE*, 11(5): 97–105, 1994. doi: 10.1109/52.311070. 29

[31] John C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In Stephen A. Schuman, editor, *New Directions in Algorithmic Languages 1975*, pages 157–168, Rocquencourt, France, 1975. IFIP Working Group 2.1 on Algol, INRIA. 32

[32] Dipanwita Sarkar, Oscar Waddell, and R. Kent Dybvig. A nanopass framework for compiler education. *Journal of Functional Programming*, 15(5), September 2005. doi: 10.1017/S0956796805005605. 34

[33] Michael I. Schwartzbach. Design choices in a compiler course or how to make undergraduates love formal notation. In Laurie J. Hendren, editor, *CC'08: Proceedings of the 17th International Conference on Compiler Construction*, volume 4959 of *Lecture Notes in Computer Science*, pages 1–15, Budapest, Hungary, April 2008. Springer-Verlag. doi: 10.1007/978-3-540-78791-4_1. 34

[34] Guy L. Steele Jr. Rabbit: A compiler for Scheme. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474. 10

[35] Sun Microsystems, 2004. Java 2 platform standard edition 5.0 development kit (JDK 5.0) documentation, 2004. `http://docs.oracle.com/javase/1.5.0/docs/` [10 November 2014]. 27

[36] David Tarditi, Greg Morrisett, Perry Cheng, and Chris Stone. TIL: a type-directed optimizing compiler for ML. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Languages Design and Implementation*, pages 181–192. ACM Press, June 1996. doi: 10.1145/231379.231414. 34

[37] Johnni Winther. The Esau LALR(1) parser generator, 2012. `http://cs.au.dk/~jwbrics/Esau/` [10 November 2014]. 23

[38] Johnni Winther and Michael I. Schwartzbach. Related types. In Mira Mezini, editor, *Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP 2011)*, volume 6813 of *Lecture Notes in Computer Science*, pages 434–458, Lancaster, UK, July 2011. Springer. doi: 10.1007/978-3-642-22655-7_21. 22