

# Comp 511 Notes

Inscribed by Anthony Castanares, Seth Fogarty,  
and Kristin Y. Rozier

March 2, 2005

## 1 Jumbo

We begin today's discussion with Jumbo. Jumbo is a compiler for a two-level version of Java. In their paper, *Jumbo: Run-time Code Generation for Java and Its Applications*, Kamin, Clausen, and Jarvis describe how Jumbo provides constructs for run-time code generation such as brackets and back-quotes. Brackets, represented as `$<>$`, signal the beginning of a value of type `Code`. The single back-quote character, `'`, is used to splice-in existing code fragments into larger ones. Jumbo has no explicit lift operator, but the semantics of lift can be achieved by using certain syntactic categories when splicing-in code. For example, the authors list various categories that reify Java values so that they may be used generated programs. These categories include `String`, `Bool`, `Float`, `Int`, `Char`.

In Jumbo, classes can be declared inside brackets. Classes declared in this manner are converted into class files at compile time when the `create` method is called with this code fragment as an argument. The `create` method invokes another method, `generate`, which converts the argument of `create` to an intermediate form, Java Virtual Machine code (JVM code), and finally returns a class file representing this code fragment.

It is important to note that code declared within brackets is not statically typed in Jumbo. Code is only checked before it is run at runtime. Although MetaOCaml performs type checking statically, it is very difficult to do so. As a side note, it would be a good experiment to develop a version of MetaOCaml that did not perform any static checking of generated code. Users would then be able to see what kind of interesting things could happen when code is

stitched together like strings without any static checking. Type checking at runtime is difficult. The authors of this paper indicate that runtime checking is performed on generated code, but it is difficult to image how this is done since the compiler usually discards the AST representation of a program before runtime. Finally, we as a class will pose the following question to Sam Kamin: Is type checking on generated code performed on byte code or source code?

The Jumbo compiler is a compositional compiler. A compositional compiler defines the meaning of a statement in terms of its subterms. For example, upon evaluation of an if-statement, code is generated for it in a compositional manner whereby code is generated for the branches of the conditional statement but filled in at some later time with values.

On to the topic of syntactic categories: when splicing-in code into an existing code fragment, the back-quote character must be followed by a special syntactic category. A syntactic category is similar to a tag that wraps an expression. During discussion, the class was trying to figure out why these categories must be explicitly specified. One suggestion was that perhaps special functions are associated with different kinds of expressions, for example, one for statements, one for identifiers, etc, and these functions can only be invoked when a particular syntactic category is encountered. But upon inspection we found that the authors gave only one explanation for the use of syntactic categories: they are there to allow for parsing of the surrounding code.

An interesting note on syntactic categories: you can do things in Jumbo with categories that can not be done in MetaOCaml. Namely, you can splice in identifiers. This would make static type checking very hard. No other language we have seen in class thus far provides such an ability. One rational reason for providing this ability is because you can write an entire program that is entirely full of quotes. Finally, in contrast to MetaOCaml, a programmer can quote types in Jumbo.

## 1.1 Piecing Code Together

In jumbo you are allowed to create case statements out of branches using `MonoLists` (pg 5). `MonoList` is a collection class interface included in the Jumbo API. At runtime, a program can throw a bunch of pieces of code together and create a case statement out of them of arbitrary length. This seems like an odd feature; perhaps its purpose is just to avoid the explicit use

of labels. Using `MonoLists`, one could quote a bunch of switch statements and piece them together. `MonoLists` allow a programmer to take pairs of conditions and statements and put them in a sequence such that the program will look for the first true condition and then execute it's expression. However, they do not provide a really clean way of piecing them together in a statically-typed way. Although, if one is dealing with arbitrary conditions it would be difficult statically typing them in any language so there is no point in making an attempt! (This property is independent of the number of possible staging levels of the language.)

The end of the paper was less satisfying than the others we have covered; their examples are not clearly useful experiments. This should be a lesson in writing research papers: when you write a paper make sure that you are very clear about what your contributions are, state them up front, and demonstrate them clearly in the text.

## 1.2 Syntax Categories

Do we really need all of the syntax categories provided by Jumbo? All of them seem to be instances of `lift`. It seems that the last seven syntax categories in Table 2, starting from `Char`, are dispensable. They could be replaced with a more simple `lift`-equivalent, which we know how to deal with.

In principle, Jumbo expression statements can be merged in the same way that they are merged in MetaOCaml without causing a problem. Essentially, what we need to do to get rid of the `'syntax-category()` construct is to join the expressions. However, that would require changing the base language, which we might not be able to do in this case. There is no example of the `Name` construct in the paper so its necessity is not obvious. The `Type` category may really be needed in the language but could probably be replaced with a construct more similar to the brackets and escapes of MetaOCaml. It is unclear what exactly the role of splicing and constructing types is in an object-oriented language. Ultimately, with a sufficiently expressive type system, what they call `Types` could be merged into expressions.

Now we are left with `Fields` and `Methods`. These two constructs can be composed without having to know which one of these is being dealt with. `Method` is a declaration like you would include in a body of a class.

## 2 Tempo

The difference between partial evaluation and 2-level languages is that partial evaluation is accomplished by the following steps:

1. The binding-time analysis process takes as inputs a program and binding time annotations. It produces a 2-level program in a language like ‘C or Cyclone or MetaOCaml which is an annotated version of the basic program that it took as input.
2. The specialization process takes as input the 2-level program and the static inputs. It outputs the specialized program.

These two major blocks are the major steps of partial evaluation. The formal link between hand-staging and what partial evaluators do in binding time analysis is a little bit unclear because of the hand-waving over CPS and other functions that partial evaluation programmers add during the binding-time analysis phase.

### 2.1 Partial Evaluation In Practice

We use MSP because it gives us better control. The principle with partial evaluation is that the programmer deals with the specialized program as a black box. In practice, a programmer has to tweak a source program many many times to arrive at just the right specialized program so ultimately, the programmer has to know exactly what’s going on anyway. The authors don’t mention the amount of work involved in partial evaluation in practice. MetaOCaml allows a programmer to do his or her own binding-time analysis for a program. If there are any binding-time improvements that make it easier to annotate to yield the right result, such as CPS, monads, etc. the programmer can make those changes whereas a partial evaluator will not do both binding-time analysis and binding-time code improvements. It is also not clear that it will do the right binding-time improvements automatically.

A programmer has to know what the resulting code should look like, formalize this vision, write a 2-level program, and then throw away the annotations in order to give the program to Tempo to put them back in. This seems backward. Especially when abstract interpretation is involved, it is very hard to image automatic optimizers being adept at binding-time improvement analysis. The way a program is written is so closely tied to how

it is staged and annotated that the programmer might as well do that part too.

It would be really nice if MetaOCaml did automatic annotations as an initial offer to the programmer. This feature should be added to the language.

## 2.2 Polyvariant Specialization

Polyvariant specialization is when you want to binding-time annotate your program with different assumptions about what is late and what is early. This is not a pressing problem because the nature of the problem usually dictates a typical scenario of what inputs will be late or early. For example, typically we only stage interpreters in one way. When it is necessary to deal with polyvariant specialization, abstract interpretation helps (as in the familiar FFT example).

The overhead of polyvariant specialization is not too bad and this feature is definitely advantageous when the program contains a recursive function or one that needs a fixed-point computation. Then it would be helpful to have two interpretations: one with the recursion inside brackets and one with the brackets on the outside. This is because the nature of the recursion depends on whether a certain input is received early or late. Here termination analysis becomes necessary, which is the Achilles heal of binding-time analysis. How well you do binding-time analysis is a variation of how well you do termination analysis.

## 3 Return to Tempo

One point of confusion was in the use of the terminology “Compile time specialization” versus “run time specialization”. When they say “compile time specialization”, they mean specialization in which you generate the source code and then compile the completed source code. The implication here is that the first and second compile times are very close together, and that the specialization depends on a small set of completely static data. The second compile time occurs “at compile time”, not “at run time”.

What they call “runtime specialization: specialization” is where you generate machine code at compile time and then assemble it in a certain fashion at run time. The problem with this is that we can generate source code at

run time and compile it, and this falls under neither category. You should leave the decision of what to generate at run time up to the programmer

## 4 Evolution

We have 'C fiddling with the compiler to do run time code generation. Then Tempo comes and say “We can grab gcc and manipulate the binaries to be able to compile the stuff into binary template, and stitch them together at run time.”

But Tempo has no guarantee of correctness, it's a hack around the compilers. Cyclone is an attempt to formalize this trick of compiling early on incomplete programs. The main issue in Tempo is keeping optimizations from operating across template barriers. Cyclone fixes this, but then attempts to bring back safe optimizations across templates. The problem here is compiling early on incomplete programs: it is not obvious that you can do any optimizations when you don't know how the templates will be assembled. Cyclone seeks to justify these optimizations as correct by justifying flow analysis on incomplete programs.

Jumbo takes a slightly different path. They write own compiler from scratch, like 'C, but the compiler work on the full language and is compositional. This is their main point: “If you want to be able to compile programs with holes in them, make your compiler compositional.” The reason Tempo fails in some cases is that gcc is not a compositional compiler, it generates code based on ad-hoc combinations of nodes.

One note on Jumbo: it is somewhere between run-time code generation and source-code generation because jvm is so close to Java.

And a brief note on writing papers: examples and diagrams are very useful. Remember this when it comes time to write your own.

## 5 Homework

Download Tempo and stage the power function.