Expressiveness

Lecture Notes for cs173, Fall 2001

December 7, 2001

Morgan McGuire

morgan@cs.brown.edu

Are all programming languages created equal?

Perl seems more powerful than straight shell scripting in Bash, and Java must have some advantage over C; why else would programmers use it? This notion of language power seems intuitive—if we can accomplish a task easily in a language because of the features we find in it, then that language must be powerful.

However, the Church-Turing thesis tells us that all programming languages with some basic features are equally powerful. Alan Turing showed that his theoretical Turing machine can execute any computation that can be performed in any programming language (read the Diamond Age or Godel-Escher-Bach for a good description). Alonzo Church did the same for Lambda Calculus. So if Lambda Calculus and Turing machines, which are each pretty simple, can run any computation, how is any language more powerful than another? In a less mathematical context, we know that all programming languages are actually translated into machine code before executing—there aren’t any Intel C++ processors out there. So, if Java and C both turn into the same set of opcodes, how is Java the more powerful one?

Performance isn’t the issue—Java is several times slower than C. It is nice for programs to be short, but program length isn’t the issue either. A “hello world” program is about the same length in Java or C, and some programs (like those that parse binary image data formats) are shorter in C than Java. So the power must be something entirely different than computability, performance, or program length.
Expressiveness

For this course, the most interesting power difference between languages is their expressivity. Church-Turing tells us that a given program could be written in either Lambda Calculus or Scheme, but this does not mean that the two implementations would look anything like each other. Because Scheme is strictly more expressive than Lambda Calculus, it is possible to write Scheme programs that have no direct correspondence to Lambda Calculus programs, but not vice versa. This means that there will be some programs that are well suited to Scheme implementations, but not Lambda Calculus implementations. It also means that there are no Lambda Calculus programs that don’t have a Scheme implementation that is at least as good.

Let’s look at two examples. In the first example, we’ll look at a program that adds two numbers (using function application, to make it interesting). This is a program where there is a direct analog between the two languages. In the second example, we’ll look at a program that is well suited to Scheme, but not to Lambda Calculus.

((lambda (x) (+ 1 x) (read))
Example 1a. A Scheme program that adds two numbers

([\ x . 1 + x] (read))
Example 1b. A Lambda Calculus program that adds two numbers.

Examples 1a and 1b show the same program in two different languages. Assume that both Scheme and Lambda Calculus have a procedure READ of no arguments that will read a number from the standard input stream. Also, in this Lambda Calculus notation, we’ll use parentheses for explicit function application, like in Scheme, and square brackets for grouping.

Examples 1a and 1b are obviously the same implementation with different syntax. We could use another syntax and still not change anything:

#include <stdio.h>
int addOne(int x) {
    return 1 + x;
}

int main(int argc, char *argv[]) {
    return addOne(read());
}

Example 1c. A C program that adds two numbers.
Well, actually, there was a change. The C language doesn’t have closures, so we had to move the function definition to the top. Maybe C doesn’t have the same READ function in its standard library; that’s not a big deal, since we could extend the library with:

```c
int read() {
    int a;
    scanf("%d", &a);
    return a;
}
```

But the fact that we had to move ADDONE is a bigger change—rather than just changing a Greek letter to a word or moving some parentheses, we actually had to move a piece of code.

Aside from closures (which we didn’t really take much advantage of), note that we didn’t use any language constructs that weren’t in each language: all of them have functions, addition, and integers. What if we use a Scheme construct that isn’t in Lambda Calculus?

```scheme
(define (inc!) (set! a (+ a 1)))
(define a (read))
(inc!)
(inc!)

Example 2a. Introducing mutation
```

Example 2a is a Scheme program that uses mutation (a.k.a. assignment). Since Lambda Calculus doesn’t have mutation, the corresponding Lambda Calculus program won’t be able to look like the Scheme program any more.

```lambdcal
([\lambda \ a \ inc] \ . \ (inc \ (inc \ a))) \ (read), \ [\lambda \ x \ . \ x + 1])
```

Example 2b. The mutating program in Lambda Calculus

Example 2b is one possible translation of Example 2a into Lambda Calculus. It looks a little like the Scheme program because we kept the same variable and function names, but it no longer uses mutation or imperative programming. Now the definition of INC is at the end of the program and INC returns a value instead of mutating a. In fact, all of the code has been moved around and touched in some way. This isn’t because we wanted the Lambda Calculus implementation to look like this—although there are many Lambda Calculus translations that will produce the same result, we can’t make a Lambda Calculus program that looks like the Scheme one. That is, we can’t make a Lambda Calculus program that expresses our program in the same way we could in Scheme. Because of this, we
say Lambda Calculus is less expressive that Scheme because it lacks mutation (among other things).

Note that mutation is not the expressiveness difference between Scheme and C, however. Because C has mutation, we can implement our program in a similar fashion:

```c
#include <stdio.h>
int a;

void inc() {
    a = a + 1;
}

int main(int argc, char *argv[]) {
    a = read();
    inc();
    inc();
    return a;
}
```

**Example 2c. A C program that adds two numbers.**

Note that once again, there is a little bit of a change because Scheme and C define their top-level environments differently. In C, we have to define `a` before INC, where in Scheme top-level definitions are mutually recursive. Nonetheless, the presence of mutation in Scheme and C let us keep the implementations very similar, except for syntax. Moreover, Lambda Calculus required a complete rewrite of the program. With regard to mutation, Scheme and C are equally expressive because they can be rewritten into each other via syntax changes, but Lambda Calculus is less expressive because mutations must be removed by completely rewriting the program.

**The expressiveness of language constructs**

We say that a language construct makes a language *more expressive* than the language was without the construct if it changes the way we can implement programs in that language. Software engineers traditionally call “a way we can implement a program” a *design pattern*. The programming language theory community has recently begun calling these *aspects*, as well.
Below are some of the language constructs present in (PLT) Scheme.

Listing 1. Some Scheme language constructs

<table>
<thead>
<tr>
<th>Construct</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET!</td>
</tr>
<tr>
<td>CALL/CC</td>
</tr>
<tr>
<td>LETREC</td>
</tr>
<tr>
<td>Procedure application</td>
</tr>
<tr>
<td>COND</td>
</tr>
<tr>
<td>IF</td>
</tr>
<tr>
<td>BEGIN</td>
</tr>
<tr>
<td>LET</td>
</tr>
</tbody>
</table>

We can determine if each construct adds expressiveness to the language by seeing if it enables a design pattern that can’t be produced without it. Recall that these constructs don’t change the set of functions (programs) we can compute, only the ways in which we can implement those programs.

In the previous section we saw that mutation adds expressiveness. SET! is Scheme’s mutation construct, so it must be adding expressiveness. We’ve also seen previously in this course that in order to use the continuation design pattern without continuations we have to manually CPS a program. We did this by adding an explicit continuation argument to every procedure and changing every return value into a continuation call. Changing “every” instance of anything means rewriting an entire program, so CALL/CC must be adding expressiveness as well. If we continued this evaluation, we’d find that most (about half of the full list) Scheme language constructs add expressiveness. This is not by accident—Scheme was designed to be a minimal language, with new language constructs added only if they added expressiveness.

Let’s skip down the list to LET. We know that LET can be rewritten using procedure application:

```
(LET ([a b]) expr) => ((LAMBDA (a) expr) b)
```

Listing 2. Rewrite rule for LET

Although this isn’t exactly a syntactic change like moving around parentheses or adding semicolons, it is not a very dramatic change. Is LET adding any expressiveness to Scheme, or is it just there to make code easier to read?

---

1 By “can’t be produced” I mean we can’t introduce the design pattern into a program without rewriting the entire program due to the lack of a certain language construct. For practical purposes, if a design pattern (like continuations) is too onerous to use without a language construct, that construct was probably adding the expressiveness needed for the pattern.
Macro-expressiveness

We call constructs like LET *macro-expressive* because they can be implemented with something historically called “macros.” *Micro*-expressive might be a better word: they are a little expressive and require only little changes to add or remove. Unlike the case with SET!, if we remove LET from Scheme, we don’t have to rewrite entire programs. We just have to rewrite local pieces around where the LETs used to be. But some rewriting still occurs, so it is still a little expressive.

Another example of a macro-expressive construct is the FOR loop in C/C++. Although it is syntactically convenient to make FOR loops, we could change all of the FOR loops to WHILE loops without changing the macro structure of the program. The rewrite rule is given below.

```
for (init; test; inc) body;  =>  init; while (test) { body; inc; }
```

Listing 3. FOR rewrite rule

We could even go further, changing all WHILE expressions into GOTOs. There we have to stop, however. Unlike Scheme, C lacks proper tail call semantics so it is not possible to eliminate FOR, WHILE, and GOTO without losing expressiveness. Proper tail call semantics is the language construct that allows Scheme to avoid needing any other iteration constructs.

Macros

Macro-expressiveness is named after *macros*. Macros are simply a way of expressing rewrite rules inside of a programming language. The use of “expressive” in the previous sentence is no accident. A sufficiently powerful macro system (like Scheme’s) frees a language from needing any constructs that are only macro-expressive. Given a macro system, LET and WHILE don’t have to be built into a language’s interpreter or compiler, they can simply be expressed in the language itself.

Many programmers are familiar with C++’s #define macros and templates. These offer a small amount of macro functionality in C++, but don’t give enough power to express rewrite rules like LET and FOR.

Scheme has a much more powerful macro facility that lets you write the kinds of rewrite rules we’ve been using throughout the course, including LET and FOR. There are actually two macro facilities—older versions of PLT Scheme provide DEFINE-MACRO and new versions have SYNTAX-CASE.
DEFINE-MACRO creates a special kind of procedure. This procedure takes a tree as input and produces a tree as output. It is special for two reasons. First, it is run at elaboration/compile time… before the rest of the program. In fact, it acts as if it were an extension to the interpreter/compiler itself. Second, the input and output trees are source code! This is a natural way to express rewrite rules: a procedure that rewrites the source code. These special procedures are the motivation behind Scheme’s somewhat bizarre syntax. The nested parentheses of Scheme give a program the form of nested lists (i.e. a tree) of code. A compiler writer would call this a parse-tree. Because the code we edit and the parse-tree the macro operates on have the same form, it is easy to write macros. If this wasn’t the case, there would be a special parse tree syntax and a separate programmer syntax, and rewrite rules wouldn’t look much like the programmer syntax.

<table>
<thead>
<tr>
<th>Macro mini-FAQ</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Q. Aren’t Macros those viruses I get in e-mail?</strong></td>
</tr>
<tr>
<td><strong>A.</strong> The name “macro” is applied to any kind of meta-scripting. In a programming language, this is code that writes code. In an application, it might be code that sends mail or modifies a document. Using Scheme macros won’t cause your mail program to send your friends love letters… unless you program that.</td>
</tr>
</tbody>
</table>

| **Q. Don’t C programmers say macros are bad?** |
| **A.** Any language construct gives you a little more rope to hang yourself, if you misuse it. Creating new syntax can make a program easier to read (e.g. LET vs procedure application). Badly designed syntax can be much harder to read than the original code, however. Some C programmers get carried away with #define and make their code hard to read. You also have to be careful with scooping, termination, and side effects when writing macros… macros let you make new kinds of bugs. Don’t be afraid of macros-- used well, they are a powerful tool. |

SYNTAX-CASE offers the same power as DEFINE-MACRO but simplifies the syntax of macro definitions. It allows us to create macro definitions that look like the pseudo-code rewrite rules we use in class, with boxes for inserting expressions, rather than procedures that walk trees.

All languages are not created equal

Scheme is powerful for two reasons. It has a small set of language constructs that seem to be exactly the ones we need for all design patterns. As far as we know, we can structure Scheme programs however we want. This is not true in other languages (like C and Lambda Calculus), where a missing language construct might force us to rewrite an entire program.

The macro system and macro-friendly syntax let us introduce new language syntax. This new syntax can add a little expressive power, but it also lets us write programs concisely. This macro system is so powerful that we can even change the syntax substantially… for example, adding infix math or iteration constructs.
Most Scheme programmers don’t choose Scheme because of the Spartan syntax, Greek letters, and fun of debugging runtime errors; they choose it because it offers a macro system that lets them extend the language as needed and guarantees they won’t have to rewrite entire programs to add a new local design patterns.

**Summary**

We now have a basis for formally comparing the power of different languages: expressiveness. This is also a basis for understanding which language constructs are essential to the power of a language, and which are simply for syntactic convenience. It is even possible to make logic proofs showing that one language is more expressive than another (see Felleisen’s paper).

We’ve also learned why Scheme has all of the parentheses. They are there so Scheme code can be easily turned into parse-trees to which we can apply rewrite rules. Scheme is not the perfect language for solving every programming problem. However, it is a really good language for building the language you can use to solve a given programming problem. Because of this, many programmers have found that it was the second to last programming language they ever needed.

**Download**

You can get the experimental v200 version of PLT Scheme, with SYNTAX CASE macros, from [http://download.plt-scheme.org/](http://download.plt-scheme.org/)

A sample DEFINE-MACRO macro that adds infix (1 + 2 * 3 instead of (* (+ 1 2) 3)) math to Scheme is at [http://www.cs.brown.edu/people/morgan/infix.scm](http://www.cs.brown.edu/people/morgan/infix.scm)

**References**

Dybvig. Writing Hygenic Macros in Scheme with Syntax-Case. Computer Science Department, Indiana University, June 1992, 29pgs  