From Macrogeneration to Syntactic Abstraction

R. Kent Dybvig (dyb@cs.indiana.edu)
Indiana University Computer Science Department
Lindley Hall 215, Bloomington, IN 47408, USA

October 1999

Abstract. In his 1967 lecture notes, Christopher Strachey states that macrogenerators are useful as the only alternative to rewriting the compiler when language extensions are needed. He also states, however, that they deal inappropriately with programs as strings of symbols, ignoring their semantic content, and that they lead to inconvenient syntax and often less transparent code. He concludes that a goal of language designers should be to eliminate the need for macrogeneration. This article attempts to reconcile the contemporary view of syntactic abstraction, which deals with programs at a higher level, with Strachey's views on macrogeneration.

Syntactic abstraction has evolved to address the deficiencies of macrogeneration and has, to a large extent, eliminated them. Syntactic abstractions are conveniently expressed, conveniently used, and usually lead to more rather than less transparent code. While a worthwhile goal for language designers is to *reduce* the need for macrogeneration through the inclusion of an appropriate set of built-in syntactic forms, this article concludes that syntactic abstraction is a valuable tool for programmers to define language extensions that are not anticipated by the language designer or are domain-specific and therefore not of sufficiently general use to be included in the language core.

Keywords: Macrogeneration, Syntactic abstraction

1. Introduction

The views on language foundations that Christopher Strachey expresses in his 1967 lecture notes, published elsewhere in this volume [7] remain impressively relevant today, over thirty years later. An overriding theme of the notes is that foundations are deep semantic issues, not superficial syntactic ones. Strachey levies rather harsh criticism on language theoreticians of the day who emphasized syntax over semantics, claiming that some had developed "an intense concern for the way in which things are written" and "a preoccupation with the problems of syntax." He claims that "the urgent task in programming languages is to explore the field of semantic possibilities," and he does precisely this throughout most of the notes.

Strachey later returns to the subject of syntax while introducing the topic of macrogeneration. He advances the view that programming languages should deal with "abstract objects (such as numbers or functions)," while pointing out that the conventional view is that

Higher-Order and Symbolic Computation 13, 57–63, 2000

"a program is a symbol string (with the strong implication that it is nothing more)." He points out that the conventional view leads directly to macrogenerators that manipulate programs as symbol strings "without any regard to their semantic content." He notes the correspondence between macrogeneration and functional abstraction, anticipating the contemporary view of macrogeneration as a mechanism for *syntactic abstraction* [1].

In Strachey's view, macrogeneration is useful for extending the power of the base language, "although generally at the expense of syntactic convenience and often transparency." He goes on to make the following classic statement regarding overuse of either abstraction mechanism.

> ... it is possible by ingenuity and at the expense of clarity to do by a macrogenerator almost everything that can be done by a function and *vice versa*. However the fact that it is possible to push a pea up a mountain with your nose does not mean that this is a sensible way of getting it there. Each of these techniques of language extension should be used in its proper place.

With this statement, he seems to be advocating support for both macrogeneration (syntactic abstraction) and functional abstraction in programming languages. Indeed, he goes on to say that macrogeneration is useful as the only alternative to rewriting the compiler when semantic extensions of a language are required. Yet he concludes his discussion of macrogeneration with the statement that "a proper aim for programming language designers [is] to try to make the use of macrogenerators wholly unnecessary." This conclusion was not drawn by someone who lacked understanding of or appreciation for macrogeneration; quite the contrary, Strachey had only two years earlier published an article introducing an early macrogeneration language, GPM [6].

This article attempts to reconcile the contemporary view of syntactic abstraction with Strachey's views on macrogeneration. Section 2 introduces syntactic abstraction and places syntactic abstraction and macrogeneration in context. Section 3 discusses the role of syntactic abstraction, including whether eliminating the need for syntactic abstraction is a worthwhile goal. Section 4 summarizes the article and discusses what must be done to make syntactic abstraction more widely supported.

2. Syntactic Abstraction

Syntactic abstraction is the introduction of new syntactic forms that abstract over common source-code patterns, usually with the aim of making programs more readable. Syntactic abstraction is not synonymous with macrogeneration, which is the *means* by which syntactic abstractions are typically defined. One can conceive of other mechanisms for defining syntactic abstractions, and macrogenerators are useful as a means for more than just syntactic abstraction.

Traditional macrogenerators deal with programs as strings of symbols (often tokens such as identifiers and numbers) without regard to the syntactic structure of the program. It is certainly true that this makes them inconvenient to use and often leads to less transparent code. These problems are largely solved, however, by contemporary syntactic abstraction facilities. In fact, these problems with traditional macrogenerators motivated the development of contemporary syntactic abstraction.

Syntactic abstraction deals with program structures, such as expressions. More recent syntactic abstraction mechanisms respect lexical scoping [1, 3] and support modular programming [8]. They also provide convenient pattern-matching facilities that simplify the definition of syntactic abstractions and make their definitions more readable and robust.

In a Scheme program, syntactic abstractions can appear anywhere an expression or definition can appear. (Scheme does not distinguish statements from expressions.) A syntactic abstraction usually takes the form (keyword subform **...**), where keyword is the identifier that names the syntactic abstraction[1]. The syntax of each subform varies from one syntactic abstraction to another, just as for core syntactic forms.

New syntactic abstractions are implemented via transformation procedures, or *transformers*. Syntactic abstractions are expanded into core forms at the start of evaluation (before compilation or interpretation) by a syntax *expander*. The expander runs once for each top-level form in a program. If the expander encounters a syntactic abstraction, it invokes the associated transformer to expand the syntactic abstraction, then repeats the expansion process for the form returned by the transformer. If the expander encounters a core syntactic form, it recursively processes the subforms, if any, and reconstructs the form from the expanded subforms. The expander maintains a record of visible identifier bindings during expansion in order to enforce lexical scoping. It augments this record when it encounters a binding form, such as a lambda

[1] An "identifier" is a syntactic entity that may denote a program variable, keyword, or constant symbol.

expression or an instance of the syntactic abstraction defining form define-syntax. The expander consults this record when it encounters an identifier reference.

Transformers receive a representation of the input form and return a similar representation of the output form. The representation of forms is implementation-dependent but must include sufficient information to determine the role of each free identifier contained within a form, i.e., an encoding of the information about identifier bindings maintained by the expander. This information must generally be associated with each identifier occurrence, since over the course of a series of expansion steps, multiple like-named identifiers with different bindings may wind up in the same form. Whether an identifier is constant (quoted), free, or bound (and if bound, to what it is bound), can be determined only after the forms surrounding the identifier have been reduced to core forms.

This entire mechanism is usually completely hidden from the programmer by the pattern-matching language, although "low-level" facilities for defining transformers may expose some or all of the mechanism. Consider Scheme's let form, defined below using the now standard "high-level" mechanism for defining syntactic abstractions in Scheme [2].

```
(define-syntax let
  (syntax-rules ()
    [(let ([x e] ...) body)
     ((lambda (x ...) body) e ...)]))
```

The definition succinctly expresses the transformation of let into a direct lambda application, following Landin's correspondence principle [4]. It is easy to see from the input and output patterns both the syntax of let and the code into which let expressions expand. Scoping relationships are clear, with the bound variables x ... visible within body but not e ... and the free reference to lambda scoped where the let definition occurs. The importance of preserving scoping relationships is underscored by the definition of or.

```
(define-syntax or
  (syntax-rules ()
    [(or e1 e2)
     (let ([t e1])
       (if t t e2))]))
```

Here there are three introduced identifiers: let, t, and if. The binding for t is visible only within the code introduced by the syntactic abstraction;

in particular, it is not visible within the code represented by e2, where it would otherwise capture free references to t within e2. Similarly, the free references to the keywords let and if cannot be captured by local bindings in the context of an or expression. Thus, the expression

```
(let ((if #f))
  (let ([t 'okay])
    (or if t)))
```

properly evaluates to okay, in spite of the local bindings for and references to if[2] and t. The expander preserves scoping relationships in the output by renaming bound variables, at least in effect. For example, the expression above expands into the equivalent of

```
((lambda (if1)
   ((lambda (t1)
      ((lambda (t2)
         (if t2 t2 t1))
       if1))
    'okay))
 #f)
```

in which bound variables have been consistently renamed as indicated by the subscripts.

3. The Role of Syntactic Abstraction

Should a goal of language design be to eliminate the need for syntactic abstraction? Certainly, a goal should be to *reduce* the need for syntactic abstraction through the inclusion of an appropriate set of built-in syntactic forms. Programmers should not be required to define syntactic abstractions for common program constructs such as let and or. Said another way, the language designer should not use the inclusion of syntactic abstraction facilities in a language as an excuse for leaving out syntactic forms that are commonly used and deserve status as primitive constructs. Of course, the implementor should feel free to code such forms as syntactic abstractions to be included in a standard library, much as is done for many standard functions.

Experience has shown, however, that syntactic abstraction remains useful for extending the language in ways not anticipated by the language designer. Perhaps equally important, syntactic abstraction is

[2] The reader may be surprised that the identifier if can be bound in this manner. Scheme has no reserved words; instead, the role of each identifier is determined by its current visible binding.

useful for extending the language in ways anticipated but *rejected* by the language designer. This is because many useful syntactic abstractions are domain-specific, i.e., useful only for a particular application or kind of application, and therefore not sufficiently common as to merit inclusion in the language.

For example, *Chez Scheme*'s lexical analyzer is expressed as a state machine implemented by a set of mutually recursive procedures. Each state is coded using the state-case syntactic form, a variant of case tailored to use in the domain of lexical analysis. While the definition of state-case is too involved to present here, the following code fragment demonstrates how state-case might be used to define the state that reads Scheme comments, once the initial semicolon has been read.

```
(define (rd-token-comment port)
  (let ([c (read-char port)])
    (state-case c
      [eof (rd-token port)]
      [(#\newline) (rd-token port)]
      [else (rd-token-comment port)])))
```

The state-case syntax requires the inclusion of eof (end of file) and else cases, resulting in more reliable code. Although not shown in the example, it also supports character ranges, such as (#\a – #\z), representing all of the characters from "a" to "z." The use of state-case in the lexical analyzer makes the code readable and reliable. In fact, it makes coding the reader in terms of mutually recursive procedures viable, which in turn results in the best possible performance[3] and more flexibility than the alternative strategy of using a lexer generator.

The state-case form is clearly useful, yet building it into the language would not be sensible. A language incorporating all useful syntactic forms would be unwieldy to the point of worthlessness.

4. Conclusion

In his lecture notes, Strachey elaborated several deficiencies of macro-generation as a mechanism for language extension. Syntactic abstraction has evolved to address such deficiencies and has, to a large extent, eliminated them. Syntactic abstractions are conveniently expressed, conveniently used, and usually lead to more rather than less transparent source code. At the same time, a wide variety of uses for syntactic

[3] Because the recursive calls are all tail calls and the states share a set of common arguments, the recursive calls reduce to unconditional jumps.

abstraction have appeared that go beyond what one would reasonably want to account for in a language definition. Languages should include a proven set of built-in syntactic forms along with syntactic abstraction facilities permitting the programmer to abstract over domain-specific syntactic patterns. This will lead to more readable and more reliable code without unnecessary language bloat and will free the language designer to concentrate less on the syntax of a language and more on its semantics.

Syntactic abstraction has become very popular in the Scheme community, but is much less so outside of that community. Mechanisms that provide limited forms of syntactic abstraction exist in other languages, e.g., the templates of C++ and the generics of Ada. Yet general-purpose syntactic abstraction has not caught on more widely. In part, this is because it has been unclear how to extend it cleanly to the algebraic syntax used by most other languages. The Dylan language design addresses this issue [5] with a rewrite system that permits incorporation of new productions into the Dylan grammar. Dylan's syntax was designed to support this mechanism, however, so extending the mechanism to other algebraic languages may not be straightforward.

Integrating syntactic abstraction with static typing is also problematic. In the presence of arbitrary user-defined transformers, it is impossible to perform type inference (or type checking) prior to expansion of syntactic abstractions. An obvious alternative is to perform type inference after syntactic expansion. This is straightforward but further complicates the already difficult problem of identifying the source of type errors in a program. Another possibility is to require or allow type declarations to be affixed to the definitions of syntactic abstractions and to perform type inference twice: once before syntactic expansion and once after. Failure to type before expansion would be an indication of an error in the body of the program, while failure to type after expansion would be an indication of an error in one of the syntactic abstraction definitions. This approach is complicated by syntactic abstractions that themselves expand into definitions of other syntactic abstractions. To support them, it may be necessary to perform multiple expansion-inference steps. The best solution may be to tightly integrate syntactic expansion with type inference, although the complexity of both may make this a difficult task.

Strachey felt programming languages should deal with abstract objects rather than program text. While syntactic abstraction deals with programs at a higher level than macrogeneration, it by nature still deals with program structure rather than abstract objects. In spite of this, one can only hope that Strachey would view syntactic abstraction as a

step in the right direction and as a more appropriate tool for language extension than string-based macrogeneration.

Acknowledgements

Discussions with Benjamin Pierce and Oscar Waddell led to a better understanding of the issues involved in combining syntactic abstraction and static typing. Comments from Olivier Danvy, Andrzej Filinski, Bernd Grobauer, Oscar Waddell, David Wise, and Zhe Yang helped improve the presentation.

References

1. R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation 5*, 4, pages 295–326, 1993.
2. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation 11*, 1, pages 7–105, 1998. Also appears in ACM SIGPLAN Notices 33(9), September 1998.
3. Eugene Kohlbecker, Daniel P. Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM Conference on Lisp and Functional Programming*, pages 151–161, 1986.
4. Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal 6*, pages 308–320, 1964.
5. Andrew Shalit. *The Dylan Reference Manual*. Addison Wesley Longman, 1996.
6. Christopher Strachey. A general purpose macrogenerator. *Computer Journal 8*, 3, pages 225–241, October 1965.
7. Christopher Strachey. Fundamental concepts in programming languages. *Higher Order and Symbolic Computing 13*, 1, pages ?–?, 1999.
8. Oscar Waddell and R. Kent Dybvig. Extending the scope of syntactic abstraction. In *Conference Record of the Twenty Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 203–213, January 1999.