Comput. Lang. Vol. 16, No. 1, pp. 97–107, 1991
Printed in Great Britain. All rights reserved

0096-0551/91 $3.00 + 0.00

TXL: A RAPID PROTOTYPING SYSTEM FOR PROGRAMMING LANGUAGE DIALECTS

JAMES R. CORDY[1], CHARLES D. HALPERN-HAMU[2] and ERIC PROMISLOW[1]
[1]Department of Computing and Information Science, Queen's University at Kingston, Kingston, Canada K7L 3N6
[2]Department of Computer Science, University of Toronto, Toronto, Canada M5S 1A4

(*Received* 22 *January* 1990)

Abstract—This paper describes a rapid prototyping system for extensions to an existing programming language. Such extensions might include new language features or might introduce notation specific to a particular problem domain. The system consists of a dialect description language used to specify the syntax and semantics of extensions, and a context sensitive syntactic transducer that automatically implements the extensions by transforming source programs written using them to equivalent programs in the original unextended language. Because the transformer is context sensitive, it is more powerful than traditional context free preprocessors and extensible languages can be used to prototype language extensions involving significantly new programming paradigms such as object oriented programming.

Language dialects   Prototypes   Source transformation   Preprocessors

1. INTRODUCTION

As the diversity of programming paradigms continues to grow and the importance of problem domain specific notation in programming languages is increasingly accepted [1], it becomes more and more important to be able to try out new language features and new notation. Ideally, we should be able to rapidly prototype the new language features in order to benefit from user experience before full scale production implementation and avoid expensive modifications to the new language implementation later.

Because the expense of producing complete new language processors is prohibitive, the usual way of conducting such prototyping experiments involves implementing the new language features on top of an existing base language, creating a new *dialect* of the original base language. Traditionally, this has been done using either a regular or context-free syntactic preprocessor such as a macro processor, or by using an extensible programming language as the base language. These traditional solutions have several drawbacks.

Syntactic preprocessors such as the PL/1 preprocessor [2], M4 [3] and the C preprocessor [4] generally limit the range of possible dialects to regular or context-free translations to the syntax of the original base language [5]. While this is a reasonably large set, it is by no means clear that all of the dialects we might wish to prototype fall in this class. In particular, dialects involving significantly new programming paradigms, such as object oriented and generic programming, cannot be prototyped in this way.

While the more powerful macro preprocessors and extensible languages such as ICON [6], CLEF [7] and Lithe [8] often allow a larger range of dialects than simply the context free set, they tend to place limits on the syntactic form of the dialect notation and remove that necessary degree of freedom in the prototyping capability. For example, macro preprocessors often limit the syntax of new constructs to simple variants of functional notation while extensible languages usually limit extensions to syntactic forms which are simple variants of the syntax of existing language features such as functional notation and binary operators.

The *mkmac* extension tool for the language Scheme [9], while still somewhat limited in its syntactic capabilities, provides a very convenient method for specifying new language features. In mkmac, dialect features are specified by example. Each mkmac macro gives an example of the desired syntax along with a transformation of the example to Scheme code to specify the semantics of the feature. By taking advantage of the inherent self-reference capabilities afforded by its interpretive nature (Scheme is a variant of Lisp), significantly new language features can be very conveniently added.

This paper describes TXL, a system explicitly designed to allow easy description and automatic prototype implementation of significantly new programming language features and programming paradigms. The goal of TXL is to provide an extension tool which allows some measure of the power and flexibility of the *mkmac* feature-by-example technique for traditional Pascal-like compiled languages. TXL uses a context sensitive transformation algorithm that is not limited by the constraints typical of most other preprocessors and extensible languages, and is driven by a concise, readable dialect specification language that conveniently expresses the syntax and semantics of new language features.

2. TXL, THE TURING EXTENDER LANGUAGE

Using the Turing programming language (or any other operational language) as a base, TXL provides the ability to describe the syntactic forms and run-time model of new language dialects at a very high level, and automatically provides an implementation of the new dialect. Dialects are described using a specially designed dialect description language (TXL).

Each dialect is described in two parts, the context-free syntactic forms of the dialect (described in terms of the syntactic forms of the base language using a BNF-like notation), and the run time model of the dialect (described in terms of a set of transformations to the base language). The TXL Processor uses these descriptions to transform source programs written in the described dialect to programs in the base language, which can then be compiled or interpreted by the normal base language processor (Fig. 1).

The syntactic forms of the base language itself are described to TXL using the same BNF-like notation used to describe syntactic forms of the dialect. The base language syntactic description forms a data base of syntactic forms used to describe the syntactic structures of the dialect. For example, the syntactic forms of the Turing base include the forms *declarationsAndStatements*, *variableReference*, *assignmentStatement*, and so on.

The semantics of the dialect are described as a set of recursive context transformations from the syntactic structures of the dialect to generated base language structures.

3. A TRIVIAL EXAMPLE

As a simple example of the description of a dialect, consider the addition of coalesced assignment short forms (i.e. the "+ =", "− =" etc. of C) to the Turing language. The desired syntactic forms can be described in terms of the Turing base forms as a replacement of the *statement* syntactic form to include the original Turing statement forms plus a new form we call *coalescedAssignment* (Fig. 2).

The new definition for the *statement* syntactic form replaces the original Turing form in the effective grammar of the dialect, so that the dialect accepted will include all of original Turing plus

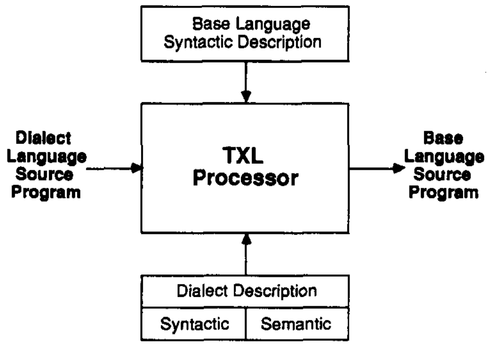
Base Language Syntactic Description
Dialect Language Source Program
TXL Processor
Base Language Source Program
Dialect Description
Syntactic
Semantic

Fig. 1. Dialect descriptions for the TXL processor.

```
% Trivial coalesced assignment dialect;
%       allows a += b etc.

% Syntactic forms

define statement                % replaces Turing base syntactic form of the same name
        choose
                [coalescedAssignment]  % new dialect statement form
                [assignment]           % original Turing
                [assert]               %       statement forms
                ...
                [get]
end define

define coalescedAssignment
        order
                [variableReference] [coalescedOperator]= [expression]
end define

define coalescedOperator
        choose + - * /
end define
```

Fig. 2. TXL description of the syntactic forms of the coalesced assignment dialect. Syntactic forms are described using a BNF-like notation in which the keyword **order** indicates sequence and the keyword **choose** indicates alternation. The dialect syntactic forms are integrated into the base language grammar by replacing an existing base language syntactic form with a new form. In the above example, the new form of statement replaces the original Turing syntactic form of the same name in the dialect grammar.

the new coalesced assignment statement. The form of the coalesced assignments themselves is described using the new syntactic form *coalescedAssignment* and its sub-form *coalescedOperator*.

The meaning of the new syntactic form is described as a transformation to equivalent Turing base language code. In this case, for example, the transformation changes the coalesced assignment $a + = b$ to the semantically equivalent Turing statement $a := a + b$ (Fig. 3).

4. IMPLEMENTATION OF TXL

The TXL processor consists of three parts, the Parser, the Transformer and the Deparser (Fig. 4). The TXL parser merges the base language syntactic description and user-supplied dialect syntactic description to form an integrated dialect language syntactic description. The merge is done by simply replacing each syntactic form specification (i.e. production) of the base language grammar by the dialect syntactic form specification of the same name (if any). In this way, the syntax of new dialect features is smoothly integrated into the features of the original base language. Using

```
% Trivial coalesced assignment dialect (continued)

% Semantic transformations

rule replaceCoalescedOperators
        replace [statement]
                V [variableReference] Op [coalescedOperator]= E [expression]
        by
                V := V Op ( E )
end rule
```

Fig. 3. TXL description of the semantic transforms of the coalesced assignment dialect. The semantics of the dialect are described using a set of rules that transform the syntactic forms of the dialect to semantically equivalent base language structures. In this case every occurrence of a statement containing the dialect syntactic form *coalescedOperator* is transformed to an assignment statement using the corresponding Turing operator.

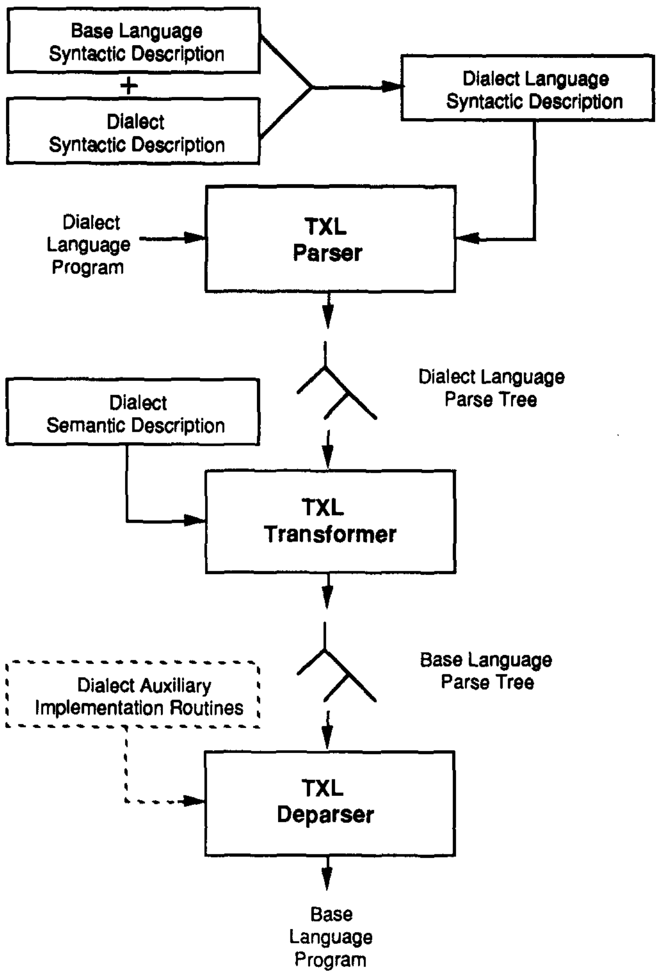
Base Language Syntactic Description
+
Dialect Syntactic Description
Dialect Language Syntactic Description
Dialect Language Program
TXL Parser
Dialect Language Parse Tree
Dialect Semantic Description
TXL Transformer
Dialect Auxiliary Implementation Routines
Base Language Parse Tree
TXL Deparser
Base Language Program

Fig. 4. Implementation of the TXL processor.

this integrated grammar of the dialect language, the parser reads in dialect language source programs and transforms then into dialect language parse trees that can be manipulated by the transformer.

As an example, the syntax of the coalesced assignment dialect of the Turing base was specified by simply copying the existing Turing base language *statement* syntactic form into the dialect description and adding an alternative for the new syntactic form *coalescedAssignment*. The dialect's new *statement* syntactic form then replaced the original Turing form in the integrated dialect grammar, effectively adding coalesced assignments to the dialect language syntax.

The TXL transformer uses the dialect language semantic transform rules to recursively transform the dialect program parse tree to a parse tree for a base language program with equivalent semantics. The transform is done using a general purpose tree pattern matching algorithm. Beginning with the main (first) transformation rule in the dialect semantic description, the algorithm searches the parse tree for instances of the rule's anchor node which match the rule's pattern and replaces the subtree of each matched instance with a new subtree for the replacement. Other transformation rules may then be applied to the replacement subtree in a similar fashion, and so on, recursively applying replacements down the tree.

As an example, the main transformation rule of the coalesced assignment dialect (Fig. 3) specifies that in each subtree below a *statement* node, any subtree matching the pattern *variableReference coalescedOperator =expression* should be replaced by another *statement* subtree containing the assignment statement *V:=V Op(E)* where *V*, *Op* and *E* are the original subtrees for the *variableReference*, *coalescedOperator* and *expression* matched by the pattern.

In order to maintain the structural integrity of the parse tree throughout the process of transformation and allow recursively applied transforms the replacement subtree is re-parsed using the anchor node production of the dialect grammar before being linked in.

Finally, the TXL deparser generates the final base language source program by walking the base language parse tree resulting from the transformations using a leftmost depth-first traversal of the tree. Some dialects, such as those introducing concurrency primitives, may involve inclusion of auxiliary implementation routines from a library in the generated result as well.

5. A MORE CHALLENGING EXAMPLE

One common modern programming technique not present in the Turing language is the ability to declare generic (i.e. type parameterized) procedures and functions. An obvious Turing dialect then is one which has this feature. However, with TXL it is just as easy to describe a dialect that allows not just generic procedures and functions, but arbitrary generic declarations including generic modules, procedures, functions, variables and types. We could imagine using such a facility to declare generics for classic data structures such a stacks, for example:

```
generic SimpleStack (someSize, someType)
      type SimpleStack :
            record
                  depth : 0 .. someSize
                  contents : array 1 .. someSize of someType
            end record
```

Later in the generic dialect program, we might instantiate a stack or two:

```
% Instantiate and use a type for big
% stacks of strings
const bigDepth := 100
instance bigStackOfString : Stack (bigDepth, string)
var bs1, bs2 : bigStackOfString

% Initialize stacks bs1 and bs2
bs1.depth := 0
bs2.depth := 0

% Push the string "hi there" on bs2
bs2.depth := 1
bs2.contents (bs2.depth) := "hi there"

% Assign the entire value of bs2 to bs1
bs1 := bs2
```

The TXL description of this dialect is given in Fig. 5. Two syntactic forms are added to the Turing declaration forms. The *genericDeclaration* form allows any form of declaration in the dialect (including generic declarations themselves) to be made generic. The *InstanceDeclaration* form allows instances of any such generic declarations to be instantiated. The intended semantics is that each instance of a generic declaration declares a new object of the original generic object type, for example, an instance of a generic type declaration has the effect of a type declaration, an instance of a generic procedure declaration has the effect of a procedure declaration, and so on.

The semantic transformations of the dialect describe this semantics as follows. The main rule *replaceGenerics* searches in the parse tree of the dialect program for occurrences of the syntactic form *DeclarationsAndStatements* (i.e. the body of a Turing language scope), and within each such occurrence (i.e. scope) finds each generic declaration. It then replaces the remainder of the scope of the generic declaration by the same scope with the generic declaration removed and instances

```
% Generic dialect of Turing - allows arbitrary generic declarations

% Syntactic forms

define declaration
        choose
                [genericDeclaration]      % new dialect
                [instanceDeclaration]     %       declaration forms
                [constantDeclaration]     % original Turing
                [typeDeclaration]         %       declaration forms
                . . .
                [moduleDeclaration]
end define

define genericDeclaration
        order
                generic [id] ( [list id] )
                        [declaration]
end define

define instanceDeclaration
        order
                instance [id] : [id] ( [list id] )
end define


% Semantic transformations

rule replaceGenerics
        replace [declarationsAndStatements]
                generic Gname [id] ( Formals [list id] )
                        Decl [declaration]
                RestOfScope [declarationsAndStatements]
        by
                RestOfScope
                        [fixInstantiations Gname Formals Decl]
end rule

rule fixInstantiations  Gname [id]  Formals [list id]  Decl [declaration]
        replace [declaration]
                instance Iname [id] : Gname ( Actuals [list id] )
        by
                Decl    [simpleSubst Gname Iname]
                        [simpleSubst Formals Actuals]
end rule

rule simpleSubst  Old [id]  New [id]
        replace [id]
                Old [id]
        by
                New
end rule
```

Fig. 5. TXL description of the generalized generic dialect of Turing.

of the generic replaced by instantiations of the generic. The actual replacement of instances with instantiations of the body is achieved by the second transformation rule, *fixInstantiations*.

Given as parameters the name of a declared generic, its formal parameter list and its body declaration, the *fixInstantiations* rule replaces each declaration of an instance of the generic in the scope of its application with a copy of the generic's body in which the name of the generic declaration is replaced by the name of the instance declaration and the formal parameter names of the generic have been replaced by the actual parameters given in the instance.

Both the substitution of the instance name for the generic name and the substitution of the actuals for the formals is achieved by the last transformation rule, *simpleSubst*. This rule simply replaces each item in its first parameter (which may be a list) by the corresponding item in its second parameter over its range of application. For the sake of presentation, the items in instance actual parameter lists have been limited to identifiers in this example. In practice, arbitrary expressions and type definitions would more likely be allowed in the dialect.

As an example of the kind of transformation done by TXL on a source program of the generic dialect, consider the *bigStackOfString* example given earlier. Given the TXL specification of Fig. 5 and the *bigStackOfString* example as input, the TXL processor would output the Turing language result:

```
% Instantiate and use a type for big
% stacks of strings
const bigDepth := 100

type bigStackOfString :
      record
            depth : 0..bigDepth
            contents : array 1..bigDepth of string
      end record

var bs1, bs2 : bigStackOfString
```

Of course, a more reasonable generic characterization of the stack data structure would be as an abstract data type complete with the operations *Push*, *Pop* and *Top*. Because our generic dialect allows generics of any kind of Turing declaration, we can also do this within the dialect by making a generic Turing module declaration:

```
generic Stack (someSize, someType)
      module Stack
            export (Push, Pop, Top)

            var depth : 0..someSize := 0
            var contents : array 1..someSize of someType



             procedure Push (element: someType)
                   pre depth < someSize
                   depth := depth + 1
                   contents (depth) := element
             end Push

             function Top :' someType
                   pre depth > 0
                   result contents (depth)
             end Top

             procedure Pop
                   pre depth > 0
                   depth := depth - 1
             end Pop
        end Stack
```

Instances of the generic module *Stack* would then themselves be modules with the operations *StackInstance.Push*, *StackInstance.Pop* and *StackInstance.Top*, for example:

```
% Instantiate and use a couple of stack modules
const smallSize := 10
const bigSize := 100

% First, a simple stack of strings
% This time, each instance of the generic is itself a complete module
instance SmallStringStack : Stack (smallSize, string)

% Use the string stack module a bit
SmallStringStack.Push ("Hi there")
SmallStringStack.Push ("Hello yourself")
SmallStringStack.Pop
put SmallStringStack.Top      % outputs "Hi there"

% Next, an integer expression evaluation stack
instance IntStack : Stack (bigSize, int)

% Procedure to perform additions in the evaluation stack
procedure Add
      const rightOperand := IntStack.Top
      IntStack.Pop
      const leftOperand := IntStack.Top
      IntStack.Pop
      IntStack.Push (leftOperand + rightOperand)
end Add

% An example calculation using the evaluation stack
IntStack.Push (10)
IntStack.Push (47)
Add
put IntStack.Top        % outputs 57
```

Using the dialect transformation rules in Fig. 5, TXL would transform each of the instances in this example into a separate module, and the whole result would look like:

```
% Instantiate and use a couple of stack modules
const smallSize := 10
const bigSize := 100

% First, a simple stack of strings
% This time, each instance of the generic is itself a complete modul
module SmallStringStack
      export (Push, Pop, Top)

      var depth : 0..smallSize := 0
      var contents : array 1..smallSize of string

      procedure Push (element: string)
            pre depth < smallSize
            depth := depth + 1
            contents (depth) := element
```

```
    end Push

    function Top : string
        pre depth > 0
        result contents (depth)
    end Top

    procedure Pop
        pre depth > 0
        depth := depth - 1
    end Pop
end SmallStringStack

% Use the string stack module a bit
SmallStringStack.Push ("Hi there")
SmallStringStack.Push ("Hello yourself")
SmallStringStack.Pop
put SmallStringStack.Top      % outputs "Hi there"

% Next, an integer expression evaluation stack
module IntStack
    export (Push, Pop, Top)

    var depth : 0..bigSize := 0
    var contents : array 1..bigSize of int

    procedure Push (element: int)
        pre depth < bigSize
        depth := depth + 1
        contents (depth) := element
    end Push

    function Top : int
        pre depth > 0
        result contents (depth)
    end Top

    procedure Pop
        pre depth > 0
        depth := depth - 1
    end Pop
end IntStack

% Procedure to perform additions in the evaluation stack
procedure Add
    const rightOperand := IntStack.Top
    IntStack.Pop
    const leftOperand := IntStack.Top
    IntStack.Pop
    IntStack.Push (leftOperand + rightOperand)
end Add

% An example calculation using the evaluation stack
IntStack.Push (10)
IntStack.Push (47)
Add
put IntStack.Top        % outputs 57
```

Although simplistic in its syntax and transform, this generic dialect in fact provides full static parametric polymorphism in the sense of Cardelli and Wegner [10]. While at first glance one would assume that TXL dialects are limited to such static features, in fact full dynamic polymorphism is achievable using a more complex transformation that uses pointers and records to represent dynamically polymorphic objects.

6. SCOPE AND LIMITATIONS OF THE TECHNIQUE

The range of possible transformations far exceeds the simple examples shown in this paper. TXL is capable of arbitrary general pattern matching, recursive transformations, arbitrary code motion, generation of unique new identifiers and reference to auxiliary support routines. It has been used to specify and implement several dialects of the Turing programming language including a complex arithmetic dialect [11], an object-oriented programming dialect with object types. Polymorphism, inheritance and dynamic binding [12] and a SNOBOL-like pattern matching dialect [11]. TXL has also been used with base languages other than Turing to prototype the T'NIAL dialect of the NIAL programming language [13], the Abacus concurrent programming language [14] and a few simple dialects of Pascal.

The Turing programming language is particularly well suited as a base language for dialects because of its relative lack of syntactic (in particular, its lack of ordering restrictions on declarations and statements), its value-inherited type inference and its ability to reference external routines. While the lack of any of these base language features would not in theory restrict the range of dialects that can be described, in practice they do help to keep the descriptions of new dialects elegant, concise and readable.

One problem with the general approach of language implementation by preprocessor is the difficulty in providing error diagnostics that are related to the original source when the base programming language processor detects errors in the transformed result. Although a corresponding source location can often be found using a relatively simple source coordinate map such as that implemented by the C preprocessor using line-and-file directives, it can be very difficult to relate the semantics of the base language error message to the semantics of the dialect program, particularly if the dialect implements a fundamentally different programming paradigm.

The range of dialects that can be described using TXL is restricted to some extent by the power of the base language chosen. For example, using Turing or any other Pascal-like programming language as a base does not allow us to describe a dialect containing the paradigm of program self-reference, because no mapping to base language source can successfully introduce that new concept into the execution of the resulting base language program.

TXL is most suitable for rapid prototyping of new programming constructs, notations and dialects, the transformational power of TXL has been shown to be equivalent to that of general Turing machines [11], thus the range of dialects that can be implemented using TXL is in theory limited only by the bounds of computability. All compiling tasks, up to and including code generation, can in theory be specified and implemented as TXL transforms. In practice, however, TXL's range of application is limited by the complexity of the transformation rule sets, which can be very great for dialects such as Objective Turing [12], and the performance of the rule interpreter, which uses a unification algorithm similar to that used in Prolog implementations. As demonstrated by the examples in this paper, it is relatively easy to make working transformations for new features whose semantics are well understood as run time models, but it is very difficult to make efficient ones. In the end, that final task is best left to professional programming language implementors.

7. SUMMARY

We have described a general technique for rapid prototyping of significantly different dialects of an existing base programming language. The technique uses a very high level rule-based specification language called TXL to describe the syntax and semantics of the dialect. In TXL, the syntactic forms of the dialect are described by giving new syntactic form definitions to replace existing syntactic forms of the base language, and the semantics of the dialect are described separately using a set of applicative syntactic transformation rules that transform the syntactic forms of the dialect to semantically equivalent syntactic forms of the base language. A dialect processor that automatically implements prototype preprocessors for language dialects specified in TXL has been used to implement several dialects of the Turing programming language.

Acknowledgements—The TXL technique and dialect specification language were designed by C. D. Halpern-Hamu and J. R. Cordy at the University of Toronto [15]. The TXL processor implementation was prototyped by C. D. Halpern at the University of Toronto and refined to production by E. Promislow at Queen's University [11]. The Turing programming language [16] was designed and implemented by R. C. Holt, J. R. Cordy, M. P. Mendell, S. G. Perelgut and others at the University of Toronto. The development of TXL was generously supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

1. Hailpern, B. Multiparadigm research: A survey of nine projects. *IEEE Software* **3**(1): 70–77; 1986.
2. *OS PL/I Checkout and Optimizing Compilers: Language Reference Manual.* Form No. GC33-0009-4, IBM Data Processing Division; 1976.
3. Kernighan B. W. and Ritchie, D. M. The M4 macro processor. In *The UNIX Programmer's Manual*, 7th edn; 1979.
4. Kernighan, B. W. and Ritchie, D. M. *The C Programming Language*. Englewood Cliffs, NJ: Prentice–Hall; 1978.
5. Standish, T. A. Extensibility in programming language design. *Proc. 1975 Spring Joint Computing Conference, AFIPS 44:* 1975.
6. Griswold, R. E. and Griswold, M. T. *The ICON Programming Language*. Englewood Cliffs, NJ: Prentice–Hall; 1983.
7. Triance, J. M. and Layzell, P. J. CLEF—A COBOL language enhancement facility. Computation Department, Report 273, University of Manchester Institute of Science and Technology; December 1982.
8. Sandberg, D. Lithe: A language combining a flexible syntax and classes. *Proc. 9th ACM Symposium on Principles of Programming Languages;* January 1982.
9. Kohlbecker, E. Using mkmac. Technical Report 157, Computer Science Department, Indiana University; May 1984.
10. Cardelli, L. and Wegner, P. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.* **17**(4): 471–522; 1985.
11. Promislow, E. A run-time model for generating semantic transformations from syntactic specifications. M.Sc. thesis, Department of Computing and Information Science, Queen's University at Kingston; 1989.
12. Cordy, J. R. and Promislow, E. Specification and automatic prototype implementation of polymorphic objects in turing using the TXL dialect processor. *Proc. ICCL '90, IEEE 1990 International Conference on Computer Languages.* New Orleans; March 1990.
13. Jenkins, M. A., Glasgow, J. I. and McCorsky, C. D. Programming styles in NIAL. *IEEE Software* **3**(1): 46–55; 1986.
14. Nierstrasz, O. The Abacus Programming language (Version 1.1). Centre Universitaire d'Informatique, Geneva; 1988.
15. Halpern, C. D. TXL: A rapid prototyping tool for programming language design. M.Sc. thesis, Department of Computer Science, University of Toronto; 1986.
16. Holt, R. C. and Cordy, J. R. The TURING programming language. *Commun. ACM* **31**(12): 1410–1423; 1988.

About the Author—JAMES R. CORDY received his B.Sc. in 1973, M.Sc. in 1976 and Ph.D. in computer science in 1986 from the University of Toronto. From 1974 to 1983 Dr Cordy served as a research associate in the Computer Systems Research Institute at the University of Toronto, and from 1983 to 1985 he was a lecturer in the Department of Computer Science of that same university. He is presently Associate Professor of computing and information science at Queen's University at Kingston, Canada. Dr Cordy is co-designer of the programming languages Concurrent Euclid, Turing and Turing Plus, The Turing programming environment and the S/SL compiler specification language. He is a member of the Association for Computing Machinery, the IEE Computer Society and IFIP working group 2.4.

About the Author—CHARLES D. HALPERN-HAMU received the B.S. degree in computer science from Indiana University in 1984, and the M.Sc degree in computer science from the University of Toronto in 1986. Since 1986, he has been pursuing the Ph.D. degree in computer science at the University of Toronto. His doctoral research concerns direct manipulation, user interface management systems, and the use of robots by the disabled. His research interests also include programming language semantics and programming language design.

About the Author—ERIC PROMISLOW attended the University of British Columbia before receiving the B.Sc. degree in biophysics from the University of Toronto in 1984. He subsequently spent two years at Simon Fraser University and Queen's University studying computing science and expects to receive the M.Sc. degree in computing and information science from Queen's University in 1990. Mr Promislow has been a software engineer at Bell-Northern Research Limited and has recently joined the software staff at Exoterica Systems in Ottawa, Canada.