Translator-Based Multiparadigm Programming

R. Nigel Horspool & Michael R. Levy
Dept. of Computer Science, University of Victoria
P.O. Box 3055, Victoria, BC, Canada V8W 3P6
E-Mail: nigelh@csr.uvic.ca & mlevy@csr.uvic.ca

Abstract

Better programming productivity may be obtained by choosing suitable programming paradigms. For development of complex software systems, multiparadigm programming would usually be appropriate. However, its use may be hindered by a lack of languages and programming support tools. As this paper argues, multiparadigm programming may be supported by translators that convert programs written in one language to another language based on a different paradigm.

Keywords and Key Phrases: Programming Paradigm, Multiparadigm Programming, Object-Oriented Programming.

1 INTRODUCTION

Conventional programming languages (that is, *imperative programming languages*) have been criticized for lacking a simple mathematical basis, for the difficulties they bring to the production of verifiable (and hence reliable) software, for lacking logical clarity and for forcing a sequential way of thinking on the programmer.[1] In spite of these complaints, and in spite of the myriad alternative programming languages offered in their place, most industrial and commercial programming is still performed with imperative programming languages. There are many reasons why imperative languages have not been swept aside, in spite of the obvious validity of many of the claims made by their critics. Although Backus, for example, argues against features of a programming language that tie it to a particular kind of machine architecture, this binding in fact can be advantageous in many situations, and is especially important for producing efficient executable code. A less commonly proposed defence, but nevertheless important one, is the fact that the idea of state-transitions, a fundamental concept for imperative programming, is, in fact, a powerful and useful problem solving paradigm.

On the other hand, it is clear that certain problems are much better solved using non-imperative paradigms. By better solved, we mean that the solution is more clear, more readable, more concise and that it more directly models the underlying problem. When considering the design of a non-trivial software system, it is unlikely that all parts of the system will be amenable to the use of same paradigm. This observation suggests that a reasonable approach to the develop-

1. A persuasive attack on imperative programming can be found in Backus' ACM Turing Award lecture [2].

ment of software would be to develop the software using more than one paradigm. The question to be answered then becomes how best to allow various paradigms to be mixed in a software system. In this paper, we will describe a particular approach to multi-paradigm programming based on programming language translation, and show how it can be applied in software development. The case study we present is the development of a programming language compiler. A compiler was chosen because the architecture of compilers is well understood.

2 BACKGROUND

Every programming language is based on a particular model of computation. In the case of an assembly language, the model coincides very closely to the actual computational hardware. For higher-level languages, the computational model is usually more abstract and a compiler is needed to translate operations defined on the computational model into operations provided by the hardware. The term *programming paradigm* is commonly used to refer to a computational model.

Table 1 lists some common programming paradigms and languages based on those paradigms. The table corresponds to a list published in [19] except that we do not distinguish between the imperative and procedural paradigms. According to [10], the list is incomplete. We should add paradigms such as array-oriented, attribute-grammar, decision table, parallel processing, pattern matching, and others. Several are discussed in [11]. Whether these are truly distinct paradigms is a question that need not concern us here.

Table 1 Representative Programming Paradigms

Paradigm	Language(s)
Imperative	Fortran, Pascal, Algol60, C
Functional	ML, Haskell
Relational	SQL
Logic	Prolog
Object-Oriented	Smalltalk

We are concerned with three major paradigms in this paper. These are the imperative, functional and logic paradigms. Roughly speaking, the imperative paradigm is identified by the use of variables, assignment statements and explicit flow of control. The functional paradigm can be considered to be a subset of the imperative paradigm (supporting expressions but not statements), but it achieves additional power by supporting higher-order objects (that is, it supports expressions that denote functions). The logic paradigm is based on the idea of solving computational problems by specifying the properties of the solution using a suitable specification language such as the predicate calculus.

3 APPROACHES TO MULTIPARADIGM PROGRAMMING

The programming paradigm that we use has a great impact on a programming task. It is not hard to find examples of programs that can be expressed, after a few minutes thought, in a few lines of code using one paradigm, yet the same program may require hours of tedious programming if

another paradigm is used. It is simply common sense to choose the programming paradigm that is most suitable for the task to be performed. But large computer applications or systems programs normally require many different tasks to be performed, and we should not expect a single paradigm to be suitable for all these tasks. For example, if we wish to construct an interactive algebraic system similar to Mathematica, say, we might prefer to use the functional paradigm for the algebraic engine that implements symbolic equation manipulation and use the imperative paradigm to handle keyboard input and screen output. But how can we use two or more different paradigms within the same program? We identify three main approaches.

1. **Language extension using procedures**. The conventional approach is to use an imperative language for the implementation and augment it with collections of procedures that support the desired paradigms. In the interests of software reusability, the procedures might be precompiled and made available as a load library. For example, the Berkeley Unix system provides a library of C functions that supports database operations. Another example is a Prolog— Ada system [17] that makes Prolog operations and datatypes available to Ada programmers. However, if the procedures have complicated interfaces or if calls to the procedures in the package have hidden side-effects, there will be little net gain in programming productivity.
2. **Multiparadigm languages**. Several languages have been designed that support combinations of paradigms. For example, C++ provides both the imperative and object-oriented paradigms. Other languages exist that combine logic programming with imperative [4][20], parallel processing with logic programming, relational with functional [18], to name just a few. Even if a single language that supports the exact combination of paradigms needed for an application is available, there may be reasons that make the language unsuitable. Lack of availability of compilers for different computer systems, and poor execution efficiency are two possible reasons not to use a particular language. Another reason is that, with the exception of C++, the languages are not in widespread use and it would be difficult to find experienced programmers. A lack of support tools and support libraries would also lessen programmer productivity.
3. **Separate processes or modules**. The system is decomposed into modules, and each module is programmed in a language that supports the most appropriate paradigm. One possibility for linking the modules is to having modules in different languages executed as independent processes [27]. They are synchronized by transfers of data between the modules. If the synchronization requirements and data transfers are sufficiently simple, however, the procedure call mechanism augmented with code to convert the representations of parameters may suffice. An interface description language like IDL [14] may be appropriate for ensuring the consistency of interfaces programmed in the different languages. A Unix program constructed as two processes linked by a pipe would be a simple example of this approach, assuming that the processes are coded using different programming paradigms. The main drawback with the approach is that data transfers between the modules may be inefficient, especially if the data must be transformed from one representation to another. Furthermore, compartmentalizing the different languages into separate modules does not necessarily eliminate harmful interactions. A long list of interactions and problems that may occur when combining code in two programming languages based on the same paradigm is given in [7]. We can reasonably expect the situation to be even worse when the paradigms differ.

The first two approaches might be described as *tightly-coupled* in the sense that objects and operations belonging to one paradigm are freely accessible from code based on the other par-

adigm. The third approach is loosely-coupled in the sense that the two paradigms are kept strictly segregated and interaction is restricted to some specific interfaces that may be tied to a particular language mechanism (such as a procedure call).

Another way to view the situation is as a spectrum of possibilities, ranged according to the level of paradigm integration. At one extreme we have special-purpose multi-paradigm languages, where features from the different paradigms may be combined in the code at a low level. For example, an assignment statement (a basic feature of the imperative paradigm) might be backtrackable (a concept of the logic programming paradigm). At the other end of the spectrum, we have the separate process approach where the paradigms are combined at a very coarse level. Procedure level integration belongs somewhere in the middle.

Undoubtedly, there are situations where one of the three approaches represents the ideal method for constructing a software system. We feel, however, that there is a need for a fourth mechanism for supporting multiparadigm programming. This new method, that we call translator-based multiparadigm programming is described in the following sections of the paper. It would be inserted in our spectrum of possibilities as follows.

4 TRANSLATOR-BASED MULTIPARADIGM PROGRAMMING

4.1 Requirements

Consider two arbitrary languages named A and B that are based on different paradigms, and suppose that we have a translator to convert A code into functionally equivalent B code. It should be self-evident that the translator would make it possible to combine user-supplied A code with user-supplied B code. Whether it would be easy or worthwhile to combine the code is a question that we will now consider in general terms. There are several observations that we can make.

1. If the translation from A to B is high-level, in the sense that the objects and operations in the A code are identifiable in the translated code, it should be relatively easy for user-supplied B code to access these objects and operations directly.
2. If B supports abstract data types and type encapsulation, the implementations in B of the objects and operations of the A language can be safely mixed with user-supplied B code.
3. If B supports operator overloading and/or syntactic extensibility, it may be possible to provide access to facilities of the A language from a B program using syntax that is similar to the syntax of the A language.
4. Access to facilities of the B language from within A language code can be provided by external interface mechanisms built into the A to B translator.

5. If B is a systems implementation language (such as C), there is a reasonable expectation that the translated A code will execute efficiently.
6. An application programmed as a mix of A modules and B modules is as portable as an application programmed entirely in B.

To amplify these points further, let us take the particular example of a translator from Prolog to C. (Several such translators exist, including [1], [3] and [25].) Point 1 suggests that if the Prolog code contains a predicate named *append/3*,[2] for example, then the generated C code should contain an object, perhaps a function, whose name is *append3* or similar, and which may be invoked in a similar manner to a Prolog predicate. Point 2 suggests that the user of an *append3* object in the C code should not have access to the internal implementation of *append3*. Such type safety is automatic if *append3* is implemented as a function in C but impossible to guarantee if the object is implemented as a data structure. Point 3 does not apply to the C language but it would partially apply to C++ where existing operators of the language can be given overloaded meanings. Even better would be a target language like Ada where entirely new operators can be defined (this is a form of syntactic extensibility). Point 4 mentions a feature that forms part of many compilers. Many current Prolog compilers and interpreters allow a Prolog predicate to call an external function coded in C (usually treating it as a deterministic predicate). There is no reason why a similar mechanism cannot be supported in an arbitrary A to B translator. Points 5 and 6 account for why C has been popular as a target language for translators. If efficiency and portability were not such large concerns, we would advocate a more flexible language like *Scheme* [21] as being a good target.

Which language is the most suitable target language for a translator? The C language has been widely used in existing translators because of its relative efficiency and high degree of portability (points 5 and 6, above). But C is not ideal when one considers some of the other points. It has no syntactic or semantic extensibility features, other than some rudimentary capabilities provided by the preprocessor. It does not have proper support for abstract data types and encapsulation.

Although it also is not ideal, we propose C++ as being a more suitable target language. C++ is a superset of C, often implemented by means of a translator from C++ to C. It, therefore, retains most of the efficiency and portability advantages of C. C++ is now available on almost all systems. In addition, C++ provides some semantic extensibility because it allows overloaded meanings to be provided for operators in the language. With its **class** construct, C++ provides a method of creating abstract data types and for encapsulating implementations of the abstract data types and their operations. It would be preferable if C++ also provided syntactic extensibility and provided automatic garbage collection, but these are not overwhelming deficiencies.

To demonstrate that translation provides a suitable support framework for multiparadigm programming, it is necessary to provide examples. The following section of the paper describes a

2. In Prolog, a predicate named *append* with two arguments is distinct from a predicate named *append* with three arguments. It is conventional to suffix the predicate name with a notation like '*/3*' to indicate the arity of the predicate and thus unambiguously define which predicate is being referred to.

small compiler project where four different paradigms are used. If additional translators had been available, we might have chosen to use even more paradigms to get the job done.

4.2 Translating Prolog to C++

The TOPIC Prolog to C++ translator [16] is, we believe, the first translator that has been specifically designed with multiparadigm programming as an objective. It allows user-supplied C++ code to manipulate and use Prolog concepts such as queries, predicates, terms and unification in a natural and type-safe manner. For example, if the Prolog predicates *mammal/1* and *swims/1* have been translated into C++ by the TOPIC translator, here is some hand-written C++ code that invokes these predicates.

```
Term *X = Variable;
Query Q = mammal1(X) & swims1(X);
Goal  G(Q);

/* print all solutions to the query
    :- mammal(X), swims(X).
*/
while( G.Next() )
    cout << "X = " << G.Value(X) << "\n";
```

In this sample code, the object Q represents the Prolog query, and the object G represents an environment for invocation of that query. G holds bindings for all variables used in the query and holds state information needed to implement backtracking within the query. The method *Next* associated with a query causes the next solution to the query to be sought, and a true/false result indicates the success of the search. The *Value* method is used to look up a binding for a variable. Note that the readability of the C++ code is enhanced with use of operator overloading. The ‘&’ operator is overloaded so that it can be used for conjunction of subqueries. (Similarly the ‘|’ operator has an overloaded meaning of disjunction.) The ‘<<’ operator, already overloaded in C++ to provide output of simple datatypes, is further overloaded to provide output of Prolog terms.

While people may argue over details of the C++ design used to implement the concepts of Prolog, the objective of making it relatively easy to integrate C++ code with Prolog code has been achieved. Our earlier arguments that the object-oriented features of C++ make it easier to develop a natural translation strategy have, we feel, been borne out in practice. In the next section, we will illustrate the use of the TOPIC translator.

If the logic programming paradigm can be integrated with the O-O and imperative paradigms of C++ this simply, it should be possible to treat other paradigms similarly. While we only have one fully implemented example of the approach and two partial examples (lex and yacc), we believe that the prospects of finding suitable C++ descriptions for concepts belonging to other paradigms are good. As some further justification, the following subsection gives an outline of a translation scheme for the functional paradigm.

4.3 Translating Functional Languages to C++

The functional paradigm is a significant paradigm that is the focus of much research. Inefficient implementations and the limited success of hardware solutions to the efficiency questions hampered past acceptance of functional programming languages. Recently, however, there has been a significant improvement in functional implementation efficiency, using both graph-reduction techniques and super-combinators (see, for example, [8]).

Two aspects of modern functional programming languages form the key distinction with the imperative paradigm. All objects (values and functions) are "equal citizens" and evaluation is lazy (arguments are not computed until the value is needed: it is sometimes called "call-by-need"). Efficient implementation of lazy evaluation is still the subject of ongoing research (see, for example, the discussion by Wray and Fairburn in [26]), and it is not our intention to claim that the translator based approach holds the solution to that problem. Rather, we wish to present a way in which the translation can be performed, achieving both equal-citizenship for higher-order functions and laziness. It is likely that implementation techniques, such as those of Wray and Fairburn, could be adapted to work with the translator-based approach presented here.

To simplify our discussion, we will use λ as an abstraction operation, and the notation [x]e to represent the bound variable and body of a lambda expression. Thus the traditional expression

```
λx.x+1
```

for example, would be represented in this notation as

```
λ([x](x+1)).
```

The advantage of this notation is that it allows us to denote nullary functions, as in

```
λ([] (3+4)).
```

Also note that the lambda calculus notation *(f x)* denotes the application of a function named *f* to an argument *x*. It corresponds to the more usual mathematical notation *f(x)*.

Consider, then, the function *f* defined as

```
f = λ([x]λ([y](x + y)))
```

In most functional languages, *f* would be a first-class object that may be passed as a parameter to other functions. It may also be partially applied, so that, for example, *f(3)* represents a new function that takes just one argument (and has the effect of returning the value of its argument plus 3 as its result). In keeping with the translation strategy used for Prolog, we have devised a possible translation scheme for lambda expressions. The idea is to map *f* into three different C++ classes, as follows.

```
class f2    represents f with two unbound arguments
class f1    represents f with one unbound argument
class f0    represents f with zero unbound arguments
```

The inheritance hierarchy is arranged so that *f0* is a subclass of *f1* and *f1* is a subclass of *f2*. The *f1* class has all the members of *f2* plus an extra (presumably private) member which contains the

value of the bound argument. Similarly for *f0* versus *f1*. Each of these classes contains a member named *apply.* So, if *addfn* represents an instance of class *f2*, then

addfn.apply(3) returns an instance of class *f1*, representing the function *λ([y] (3+y))*

Similarly, if *add3fn* represents the class instance created with the previous example, then

add3fn.apply(4) returns an instance of class *f0*, representing the function *λ([](3+4))*

Our proposed scheme uses the lazy evaluation approach, as mentioned above. Therefore, to obtain the fully evaluated result of 7 from the *f0* class instance, we would call its *eval* method. Each of the classes has an *eval* method, but only in the case of the *f0* class would that method return an integer result.

All classes created from lambda expressions share a common ancestor, named *Closure* which thus forms the root of the class hierarchy. There is no difficulty in representing lambda expressions such as

λ([f](λ([x](f x)))) λ([z] z)

that return functions (class instances) as their results. In lambda calculus, this expression reduces to *λ([x](λ([z] z) x))*. Suppose that the sub-expression *λ([f](λ([x](f x))))* is translated to the three classes *E0*, *E1* and *E2*. They form a hierarchy with *E0* a sub-class of *E1*, *E1* a sub-class of *E2* and *E2* a sub-class of *Closure*. Also, suppose that the other sub-expression, namely *λ([z] z)*, is translated to a class hierarchy containing *F0* and *F1*. Instances of either expression can be created by using C++ declarations. For example

```
E2 e;
```

creates an instance of the first sub-expression. To apply such an object to a single value requires use of the apply method, as in

```
e.apply(t)
```

where t is a Closure instance.

The C++ code to evaluate the complete lambda expression is:

```
E2 e;
F1 f;
Closure *result;
result = e.apply(f);
```

The implementation of a translator based on this strategy has been described in [23].

4.4 Existing Translator-based Tools

Two successful Unix tools use an approach to mixing paradigms based on translation. They are *lex*[15] and *yacc[13]*.

Conventionally, a computational model based on a finite state automaton (FSA) is adopted for the lexical analysis phase of a compiler. While a FSA is not a universal computing machine equivalent in power to a Turing machine, there is no reason why it should not be useful as a computational model in a restricted domain. It is also conventional to define a FSA using regular expressions. Regular expression notation may be viewed as a declarative language. The standard Unix tool that supports a regular expression language for defining a FSA is *lex*. The lex tool comes close to being a translator for supporting multiparadigm programming. It works by translating the regular expressions into a FSA implemented as a C program. It further permits user-supplied C code to be wedded to the FSA execution. The translation scheme is not as high-level as it might be, since C code can only be invoked after a pattern has matched. However, lex does exemplify our approach well.

Similarly, the syntactic recognition phase of a compiler is conventionally based on a push-down automaton (PDA) computational model. The language used to define the execution of the PDA is normally BNF (or similar). BNF may also be viewed as a declarative language. The standard Unix tool that supports a BNF-like language is *yacc*. It too works by translating the input notation into C code, and allows user-supplied C code to be combined with the execution of the PDA.

5 A DEMONSTRATION SYSTEM: A COMPILER

5.1 Selection of Suitable Paradigms

The conventional structure for a (non-optimizing) compiler is shown in Figure 1. Each phase of the compiler represents a well defined task, and it is reasonable to choose an appropriate programming paradigm for the implementation of each one. Tools like lex and yacc have proven their utility for creating the first two phases of a compiler. We briefly review the paradigms that should prove useful for the remaining phases.

The semantic analysis phase involves type checking and enforcing other semantic restrictions of the language being compiled. Some work preparatory to code generation, such as allocating storage addresses to variables, may also be performed. A possible paradigm that might be used for specifying semantic analysis could be a declarative paradigm based on attribute grammars. Many attribute grammar evaluator systems exist, though there is no standard one in the same sense that lex and yacc have become standard in their particular environments. An attribute grammar does not necessarily provide the most convenient paradigm to use when building a compiler, even for a simple language. A telling comparison appears in [22], where the same compiler is specified using two different attribute grammar systems and using a logic programming system named *Gentle*. In terms of brevity and clarity, the Gentle specification is a clear winner. Certainly, the logic programming paradigm is adequate for specifying type checking rules and has previously been used as the basis of an automatic type checker generator [6]. The imperative paradigm is better suited for other work, such as entering and looking up identifiers in a symbol table, and allocating storage addresses to variables.

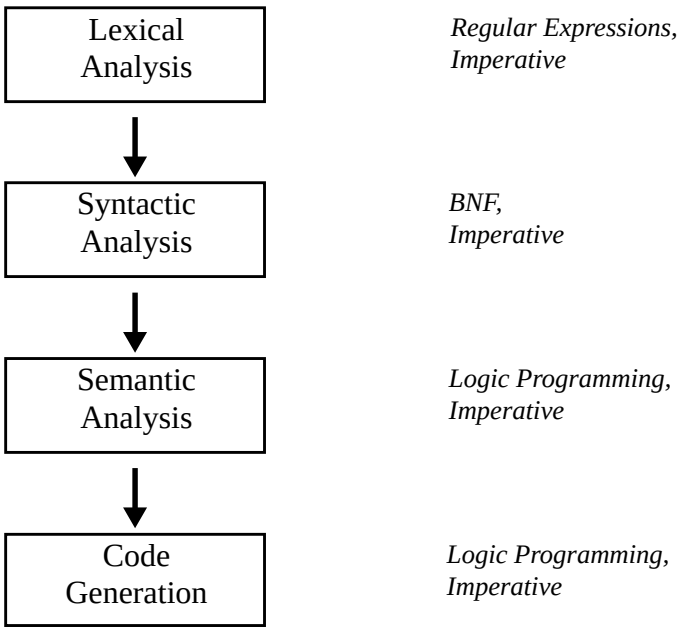
No clear choice of paradigm exists for describing the code generation phase of a compiler either. Perhaps a decision table paradigm might be used for instruction selection, but it would not be appropriate for all applications and no widely available support tool exists. For a reasonably conventional computer architecture, the instruction selection component of the code generation phase is easily described using logic programming rules. Other aspects of code generation, such as register allocation and emission of assembly language instructions, may be implemented using an imperative paradigm.

5.2 A Tour of the Implementation

We have chosen a simple imperative programming language, almost identical to the language used by Warren [24] for our demonstration system. We have added an additional datatype (*set of integer*) to make the type checking rules more interesting. The language is similar to Wirth's PL/0 language. Alternatively, it may be viewed as a subset of Pascal.

The lexical analysis and syntactic analysis phases form the compiler's front-end. The front-end builds an abstract syntax tree (AST) to represent the input program, and this AST is traversed by the semantic analysis and code generation phases. As stated above, we are using a logic programming paradigm to implement most of the semantic analysis and code generation phases. Consequently, most of the work of these phases is expressed in the Prolog language. The obvious way of representing the AST is therefore as a Prolog term. Figure 2 shows the correspondence between a trivial input program, its AST and the equivalent Prolog term. Several unbound variables (T1, T2 ...) are included in the Prolog term. These represent the datatypes of the subexpressions they appear in. The type checking rules of the semantic analysis phase subsequently bind the variables to appropriate datatypes.

Figure 1 Phases & Possible Paradigms for a Compiler



Lexical Analysis

The notation supported by lex allows C code to be attached to regular expressions. For example, the following lex rule is used for matching both identifiers and keywords of the demonstration language.

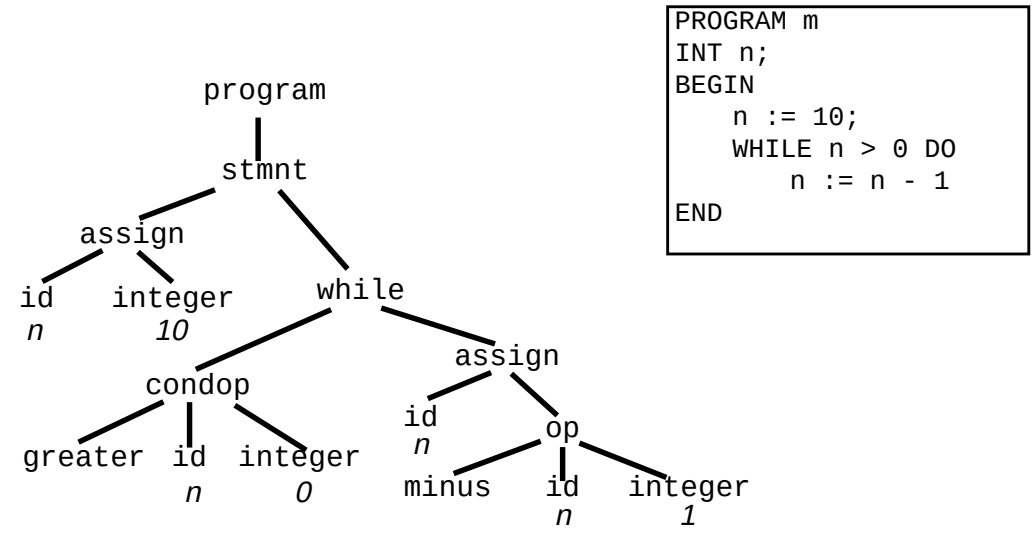
```
{Letter} {Letter_or_Digit}*    { return kwd_lookup(yytext); }
```

The first part of the rule is a regular expression that matches a single letter followed by zero or more occurrences of a letter or digit. (It uses previous definitions for *Letter* and *Letter_or_Digit*.) The second part of the rule is composed of arbitrary C code. In this case, we provide a single C statement that calls a separate C function to look up the matched text in a table of keywords. If the text occurs in the table, the function returns an integer code for the keyword, otherwise it returns an integer code that represents the *Identifier* lexical element of the language. The **return** statement causes that integer code to be passed back as the result of the lexical analyzer function.

The ability to combine the pattern matching power of the FSA paradigm with the imperative paradigm of C code is sufficiently powerful that some sophisticated programs can be built. Typical applications for this combination are tools for converting files from one text formatting system to another, scanning English text files, reformatting assembler files, etc.

Lex is relatively old, predating the idea of multiparadigm programming. It is not surprising that the FSA and imperative paradigms are not perfectly integrated. A redesigned version of

Figure 2 A Sample Program: its AST and Prolog Representations




```
program(stmnt(assign(id(n,T1),integer(10)),
  while(condop(greater,T2,id(n,T3),integer(0)),
    assign(id(n,T4),op(minus,T5,id(n,T6),integer(1))))))
```

lex that integrated the two paradigms more closely would probably provide regular expressions as objects that could be manipulated and used from within C or C++ code.

Syntactic Analysis

BNF notation is generally used for describing syntactically legal patterns of lexical elements in a programming language. When a bottom-up parsing technique like one of the LR methods is used to implement parsing, the underlying computational model is that of a deterministic pushdown automaton. However, we might equally well adopt a more general view that the computational paradigm is pattern matching. The standard Unix tool that uses BNF notation to generate a recognizer is yacc. It is used in a similar way to lex, in that arbitrary C statements may be attached to patterns, written as BNF rules.

In our demonstration compiler, we attach C++ statements to the rules. (Using C++ instead of C causes no difficulties, given the close relationship between the two languages.) The C++ statements are mainly concerned with construction of the Prolog term corresponding to the abstract syntax tree. A sample pair of BNF rules and their associated C++ code are as follows.

```
Expr: number
     {  $$ = new Struct("integer", 1, new Integer($1));  };
Expr: Expr '+' Expr
     {  $$ = new Struct("op", 4, new Atom("plus"),
        new Variable, $1, $3 );  };
```

The first rule specifies that an expression, denoted by *Expr*, can consist of just a number. When this case occurs, the associated C++ code causes a new object to be created. The object is, in fact, the C++ implementation of the Prolog term that represents an AST subtree for the expression. The term has a functor name of *integer*, and the functor has one argument which is another term – the C++ implementation of a Prolog integer. The second rule specifies that an expression can be composed of two subexpressions separated by an addition operator. In that case, we again construct an AST subtree. The new subtree is represented by a Prolog term with the functor name *op*. Two of the functor arguments are Prolog terms for the operands of the addition. Two other arguments specify the operation name, *plus*, as a Prolog atom and provide an anonymous unbound variable to represent the type of the expression.

As with lex, the marriage of BNF notation with the imperative paradigm is not as general as it could, or should, be. The design of yacc was, however, remarkably successful for its time.

Semantic Analysis

As stated above, the type checking rules of a typical programming language are usually easy to express as logic programming rules. For example, the rules for checking two of the statement types in the demonstration compiler are as follows.

```
% Check an if statement. The test expression must have bool type
% and the 'then' clause must also satisfy the type checking rules.
typeCheck( if(C,S) ) :- typeOf(C,bool), typeCheck(S).

% Check an assignment statement. In this language, the source and
% target of the assignment must have the same datatype.
typeCheck( assign(Id,E) ) :- typeOf(Id,T), typeOf(E,T).
```

where a different Prolog predicate, *typeOf*, is invoked for type checking expressions. Some of the rules defining *typeOf* are as follows.

```
typeOf( id(_,T), T ).
typeOf( integer(_), int ).
typeOf( op(plus,int,L,R), int ) :- typeOf(L,int), typeOf(R,int).
typeOf( op(minus,int,L,R), int ) :- typeOf(L,int), typeOf(R,int).
%      similar rules for other operator/type
%      combinations are omitted
   ...
typeOf(E,T) :- write( 'Type Mismatch' ), nl.
```

More sophisticated rules are required for defining the type systems of languages that permit overloading of operators and support automatic type coercions. If the rules become too complicated, a different paradigm, such as that of attribute grammars, may be more appropriate.

The type checking rules, above, assume that every use of an identifier has been associated with the type that was previously declared for that identifier in the program. Compilers, therefore, typically include a symbol table where identifiers and their types are entered when processing a declaration. A look-up operation on the symbol table returns the type associated with a given identifier. Additional operations are provided for handling languages that have block-structured scope rules.

Although it is not difficult to implement a symbol table in Prolog (as Warren did [24]), it is more appropriate to use the imperative paradigm. This enables an efficient implementation of identifier look-up, such as hashing, to be used and it suits the usual state-oriented nature of a symbol table better. We can view the action of entering an identifier in the table as an operation that changes its state. Similarly, the action of exiting a scope block causes identifiers declared in that scope to be removed from the table. We therefore implemented the symbol table in C++, invoking the symbol table operations from within Prolog code.

Code Generation and Peephole Optimization

It is very natural to try to express code-generation patterns in some kind of specification language which is then used as to input to a code-generator generator. We argue that Prolog is a good choice for this specification language. Certainly, the code generation patterns for a conventional target

computer turn out to be very easy to express as logic programming rules. Furthermore, most standard peephole optimizations can be described using additional rules. Simple Prolog implementations for both code generation and peephole optimization are given in [5]. A less direct approach for code generation has been described by Ganapathi [9]. He gives a two-phase technique which has the advantage of allowing the code generation rules to be given independently of the AST format. Some sample Prolog rules that might be used to define translation of the integer addition operation into Motorola 68000 assembly language are shown in Figure 3. Separate driver code, also written in Prolog, is used to traverse the AST. It is responsible for register allocation and for passing components of the tree, one node at a time, to the rules given in Figure 3.

6 CONCLUSIONS

The compiler implementation described in Section 3 illustrates two things. First, that the multiparadigm approach based on translation can be used in a natural way. Second, that the approach has already been in use, in a limited way, using the lex and yacc tools for nearly 20 years. If lex and yacc had been recognized for what they are, some effort might have been expended on improving the structure of the C code generated by these tools. With a different design and, perhaps, a different choice of target language, it would be easier to use the BNF and regular expression notations within any program (not just in a compiler).

We have proposed a new methodology for multiparadigm programming. While a closer integration of two or more paradigms is achievable by designing new programming languages, there is a limit to how many combinations can be reasonably supported or used within a project. Another alternative of implementing different modules using different paradigms and linking the modules using message passing techniques permits many different paradigms to be used together, but achieves a relatively small degree of integration. We think that our approach, based on inter-language translation, achieves a much better compromise between ease of use and degree of integration than either of the other approaches.

Our experience with the compiler application and with other modest applications convinces us that the translator-based approach to software development is productive. It would be

Figure 3 Sample Prolog Rules for 68000 Code Generation

```
% addition of zero is optimized away
code( +(T,0), T ).

% use special instructions for addition of small constants
code( +(T,N), T ) :- integer(N), N>0, N=<8, emit(addql,N,T).
code( +(T,N), T ) :- integer(N), M is -N, M>0, M=<8, emit(addql,M,T).

% perform an in-place addition if the result can be held in the same
% register as the left operand
code( +(T,N), T ) :- emit(addl,N,T).

% handle the general case
code( +(T1,T2), R ) :- emit(movl,T1,R), code(+(R,T2),R).
```

desirable to evaluate the method in the development of a large-scale software system. We also need to develop translators for more paradigms. Work on a functional to imperative paradigm is continuing, while other translators are contemplated.

ACKNOWLEDGEMENTS

We are indebted to Michael Junkin for implementing the TOPIC system.

We gratefully acknowledge financial support for this project from IBM Canada Ltd., and the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] B. Arbab, “C-Log: A Source Level Translator from Prolog to C.” Unpublished manuscript, IBM, Santa Monica, CA (1990).

[2] J. Backus, “Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs.” Comm. ACM 21, 8 (August 1978), pp. 613-641.

[3] J. L. Boyd and G. M. Karam, “Prolog in C.” Technical Report, Carleton University, Ottawa (March 1988).

[4] T. A. Budd, “Blending Imperative and Relational Programming.” IEEE Software 8, 1 (Jan 1991), pp. 58-65.

[5] J. Cohen and T. J. Hickey, “Parsing and Compiling Using Prolog.” ACM Trans. on Prog. Lang. and Sys. 9, 2 (April 1987), pp. 125-163.

[6] T. Despeyroux, “Executable Specifications of Static Semantics” in *Semantics of Data Types*, G. Kahn, D.B. MacQueen and G. Plotkin (Eds.), Lecture Notes in Computer Science vol. 173, Springer-Verlag (1984), pp. 215-233.

[7] B. Einarsson and W. M. Gentleman, “Mixed Language Programming.” Software – Practice & Experience 14 (April 1984), pp. 383-395.

[8] J. Fairburn and S.C. Wray, “Code Generation Techniques for Functional Languages.” Proc. of ACM Conf. on Lisp and Functional Programming (1986),pp. 95-104.

[9] M. Ganapathi, “Prolog Based Retargetable Code Generation.” Computer Languages 14, 3 (1989), pp. 193-204.

[10] B. Hailpern, “Multiparadigm Languages and Environments.” IEEE Software 3, 1 (Jan. 1986), pp. 6-9.

[11] B. Hailpern (ed.), Special issue on Multiparadigm Languages and Environments. IEEE Software 3, 1 (Jan. 1986), pp. 6-77.

[12] R. Hayes and R. D. Schlichting, “Facilitating Mixed Language Programming in Distributed Systems.” IEEE Trans. on Software Eng. SE-13, 12 (Dec. 1987), pp. 1254-1264.

[13] S. C. Johnson, “Yacc – Yet Another Compiler Compiler.” C.S. Tech. Report 32, Bell Telephone Laboratories (1975).

[14] D. A. Lamb, “IDL: Sharing Intermediate Representations.” ACM Trans. on Prog. Lang. & Systems 9, 3 (July 1987), pp. 297-318.

[15] M. E. Lesk & E. Schmidt, “Lex – A Lexical Analyzer Generator” in *UNIX Programmer’s Manual 2*, AT&T Bell Laboratories (1975).

[16] M. R. Levy, R. N. Horspool, and M. Junkin, “The Translation of Prolog into C++.” Unpublished internal report, Univ. of Victoria, (Dec. 1990).

[17] N. Madhav, “An Ada—Prolog System.” Tech. Report CSL-TR-90-437, Stanford Computer Systems Laboratory (Aug. 1990).

[18] Y. Malachi, Z. Manna & R. Waldinger, “TABLOG – A New Approach to Logic Programming” in *Logic Programming: Relations, Functions and Equations*, D. DeGroot and G. Lindstrom (Eds.), Prentice-Hall, 1985.

[19] J. Placer, “Multiparadigm Research: A New Direction in Language Design.” SIGPLAN Notices 26, 3 (March 1991), pp. 9-17.

[20] A. Radensky, “Toward Integration of the Imperative and Logic Programming Paradigms: Horn-Clause Programming in the Pascal Environment.” ACM SIGPLAN Notices 25, 2 (Feb. 1990), pp. 25-34.

[21] G. L. Steele Jr. and G. J. Sussman, “Scheme: An Interpreter for the Extended Lambda Calculus.” Memo 349, MIT Artificial Intelligence Laboratory (1975).

[22] W. M. Waite, J. Grosch & F.-W. Schröer, “Three Compiler Specifications.” Tech. Report 166, GMD, Karlsruhe (Aug. 1989).

[23] X. Wang, “Compiling Functional Programming Languages Using Class Hierarchies.” M.Sc. Thesis, Dept. of Comp. Science, Univ. of Victoria (1992).

[24] D. H. D. Warren, “Logic Programming and Compiler Writing.” Software – Practice & Experience 10 (1980), pp. 97-125.

[25] J. L. Weiner and S. Ramakrishnan, “A Piggy-Back Compiler for Prolog.” Proc. of SIGPLAN ‘88 Conf. on Prog. Lang. Design and Implementation, Atlanta (June 1988), pp. 288-296.

[26] S. C. Wray and J. Fairburn, “Non-Strict Languages – Programming and Implementation.” Computer Journal, 32, 2 (1989), pp. 142-151.

[27] P. Zave, “A Compositional Approach to Multiparadigm Programming.” IEEE Software (Sept. 1989), pp. 15-25.