VERY-HIGH-LEVEL PROGRAMMING OF KNOWLEDGE REPRESENTATION SCHEMES

Stephen J. Westfold
Stanford University and
Kestrel Institute, Palo Alto, CA 94304

ABSTRACT

This paper proposes building knowledge-based systems using a programming system based on a very-high-level language. It gives an overview of such a programming system, *BC*, and shows how *BC* can be used to implement knowledge representation features, providing as examples, automatic maintenance of inverse links and property inheritance. The specification language of *BC* can be extended to include a knowledge representation language by describing its knowledge representation features. This permits a knowledge-based program and its knowledge base to be written in the same very-high-level language which allows the knowledge to be more efficiently incorporated into the program as well as making the system as a whole easier to understand and extend.

§1 Introduction

A knowledge-based system typically consists of a program and a knowledge base that the program uses. The knowledge base is expressed in a special knowledge representation language that is essentially a very-high-level language that the program interprets. This paper describes a very-high-level language programming system, *BC*, and shows how *BC* can be used to define knowledge representation languages so that they can be efficiently compiled. Furthermore, the knowledge-based program itself can be specified in *BC* using the same techniques with the same advantages of ease of comprehension and maintainability that are associated with the knowledge base. This allows the knowledge base to be viewed as part of the specification of the program, which is the key to its efficient incorporation into the program. In this way *BC* may be viewed as a *knowledge compiler*, pre-processing knowledge so that it is used efficiently in the knowledge-based system.

This research is supported in part by the Defense Advanced Research Projects Agency Contract N00014-81-C-0582, monitored by the Office of Naval Research. The views and conclusions contained in this paper are those of the author and should not be interpreted as representing the official policies, either expressed or implied of KESTREL, DARPA, ONR or the US Government.

BC allows programs to be factored into a description of the problem to be solved and a description of the implementation of the solution. The implementation description can include schemes for representing entities of the problem description or solving particular types of sub-problem. *BC* can be used to define implementation schemes for knowledge representation features such as property inheritance, inverse link maintenance, and procedural attachment. The definitions of the first two of these features are given later in this paper. *BC* is described fully in [Westfold, 1984].

The specification language for *BC* is basically a mathematical language including logic, sets, relations, and functions. This very-high-level language is convenient for defining new language constructs in terms of existing constructs, and there is a mechanism for defining syntax for the new constructs. Thus the system designer can define a language that is convenient for system users; the parser converts this language into relations that are defined in terms of mathematical objects that have properties that facilitate their manipulation (compilation) by *BC*. By use of manipulation such as equivalence transformation *BC* can produce an implemented program whose structure is quite different from that of the problem specification. In other words, convenient, uniform interfaces can be defined for the user and to facilitate the description of the different components of the system, but the implementation can be non-uniform, crossing interfaces and taking advantage of different views of the problem domain in order to produce an efficient program.

The ideas in this paper are being tested by using *BC* in building the CHI knowledge-based programming system [Green et al., 1981]. CHI includes the following components, all of which make use of *BC* in their specification and implementation: data structure selection, algorithm design, parallel algorithm derivation, and project management, the database manager, program analysis, finite differencing, and *BC* itself. Many of these components are useful in building knowledge-based systems, so CHI as a whole is better than just *BC* for building knowledge-based systems.

§2 Overview of BC

BC is essentially a compiler that produces Lisp code from a specification in the form of logic assertions. The specification consists of three parts: the basic definition of the problem domain; the definition of auxiliary objects that are needed in an efficient implementation of the problem domain; and information about how the defining assertions are to be used procedurally. It is convenient to identify and use an intermediate rule language in going from the logic assertion language to procedural Lisp. A rule specifies an action (procedure) in terms of its precondition (applicability condition) and postcondition (what is true after its application). A rule consists of two logical formulas, written as

$$P \rightarrow Q$$

where P is the precondition and Q is the postcondition. (Note that '$\rightarrow$' is a procedural construct and '$\Rightarrow$' is the symbol for implication.)

The part of *BC* that compiles the logic assertion specification into rules is called the Logic Assertion Compiler (LAC). From an assertion, which could be used to make many different inferences, and instructions stating which *particular* inference and in what context, LAC produces a rule that is a specification of that particular inference. The part of *BC* that compiles rules into Lisp is the Rule Compiler (RC). It works by a process of stepwise refinement similar to other transformational systems such as PECOS [Barstow, 1979], TI [Balzer, 1981], and [Burstall and Darlington, 1977]. At intermediate stages of refinement the program contains a mixture of constructs from very-high-level to low-level, so a *wide-spectrum* language must be used that includes all these constructs in a unified framework.

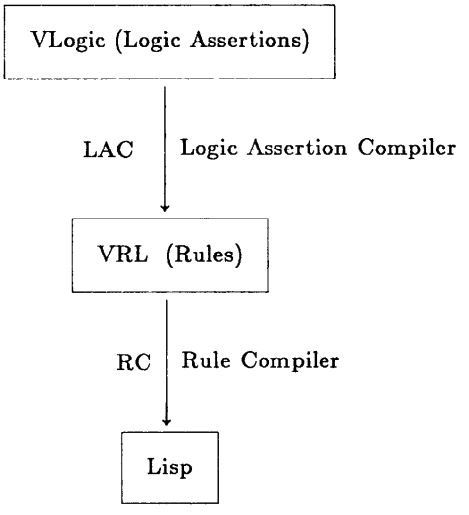

Figure 1. Structure of *BC*

The language used by *BC* is called *V* and it is the language used throughout CHI. V was initially defined by Phillips [Phillips, 1982] and has since been refined and extended by the CHI group. It contains a number of integrated sub-languages: a first-order predicate logic language, VLogic, which is the basic specification language used by *BC*; a rule language, VRL; a procedural language, VP; and the target language Lisp.

2.1 Procedural Use of Assertions

LAC compiles a specification written in VLogic assertions by converting each assertion into an inference procedure specialized to that assertion. The user specifies which particular inference procedure should be used. *BC* provides three dimensions of choice for the type of inference procedure. The first corresponds to the general form of the assertion that is used: either an implication

$$p \Rightarrow q$$

or an equality (with equivalence considered a special case of equality)

$$p{=}q.$$

Each of the general forms may have a precondition which is written as the antecedent of an implication with the form as the consequent. For example: $r \Rightarrow p{=}q$ can be considered an equality with precondition r. It may also be treated as an implication.

The second dimension corresponds to the direction of use of the general form: from left to right or right to left. For implication, the former corresponds to forward or data-driven inference and the latter to backward or goal-directed inference. Considering the assertion as a constraint, the former corresponds to enforcing the constraint and the latter to using or taking advantage of the constraint. An equality is commutative, but typically there is a directionality associated with each one. For example, a function f can be defined using an equality of the form $f(x){=}$def.

The third dimension is choice of compile-time versus run-time use of an assertion. Use of an assertion at compile time provides the possibility for circumventing the clean specification-level interfaces and producing efficient, tangled code. The result of compiling an assertion for compile-time use is a procedure that affects the compilation of other code.

An important use of assertions at compile time is to maintain and use them as constraints. Constraint incorporation is done at the stage of compilation where a procedure is expressed as a rule. Rule compilation involves using the rule to form a statement in logic of the relationship between the computation states before and after the rule application, and then producing a procedure that, given an initial state, will produce a new state that satisfies the relationship. The intermediate statement in logic is a convenient form for performing inference to incorporate constraints stated in logic assertions.

Use of an assertion at run time requires converting it to the run-time constructs available in the target environment. Therefore we need to consider two models of computation: the model of computation as inference at the specification level and the Lisp model which is basically a recursive function model. This means that any run-time inferences have to be put into a functional form. Goal-directed, run-time inference can be implemented efficiently using Lisp functions. This may involve adding an extra definition so that the goal is in the form of a function call.

In order to implement forward-inference procedures we need some extra machinery in the target environment. The procedures need to be attached somewhere so that they are triggered at the appropriate time, and they need to be able to store the values that they compute so that the values are found when wanted. This can be done with a database of [function, argument, value] triples that are indexed by the function and argument. *BC* uses a database that stores objects (the things that may be function arguments) as mappings from functions to values. Functions that are treated in this way are called *properties*. Storing the value of a property in the database may trigger attached forward inference procedures which may store values for other properties. When the value of a property is needed, the database is examined to see if there is a stored value, otherwise a Lisp function for computing the value is called, if there is one.

2.2 Specifying How to Use an Assertion

The ways an assertion is used are specified by attaching simple meta-assertions to the assertion. This section describes the basic options provided by *BC*.

Run-time use is encapsulated as a function. For forward use it is necessary to specify the triggering form that causes the function to be called. For backward use it is necessary to specify the name of the function whose value is to be computed:

triggered-by $form_1$, $form_2$, ...
(the $form_i$ are the triggering forms)

computes fn_1, fn_2, ... (Closed functions)
(the fn_i are the functions to be computed)

Other options are:

memo (Save **computes** values in database)
check (Give an error if assertion violated)

For compile-time use it is necessary to specify whether the assertion is to be used as a constraint for optimization or as a constraint to be maintained (or both), or for transforming some forms into equivalent ones.

compile-optimize form (Backward)
(Use the assertion to remove redundant tests)

compile-in-line form (Forward)
(Add in-line code to maintain the constraint)

compile-transform form
(Transform form to an equivalent form)

For convenience the forms may be referred to by their primary function if this is an unambiguous referent.

These are the basic meta-level annotations. Internally, they are simply meta-level properties of assertions. New annotations can be defined in terms of these basic ones using logic assertions at the meta-level from which *BC* can produce demons that, given the new annotation, generate the equivalent basic annotations.

2.3 The Implementation of *BC*

BC is written primarily in its own languages—VLogic and VRL. A basic version of RC was written in Lisp and then the VRL specification of RC was compiled and this version replaced the Lisp version. The implementation of LAC is at the stage where it can compile assertions given in the exact form needed for the particular use of it. The part of LAC that preprocesses assertions to get them into the correct form has been designed and is in the process of being implemented. *BC* has been developed in Interlisp [Teitelman and Masinter, 1981] on a DEC 2060 machine and then in Zetalisp [Weinreb and Moon, 1981] using the Interlisp Compatibility Package on Symbolics 3600 machines.

§3 Example Implementations of Knowledge Representation Features

The examples begin with a simple database that only provides storage and retrieval of binary-relation triples. This is used as the basis for defining knowledge representation features. The examples presented are for maintenance of inverse links and property inheritance. Other features that have been specified are specialized treatment of transitivity, attached procedures, and memoing of computed properties.

3.1 Maintaining Inverse Links

The first example is the task of maintaining inverse links in a database. This requires that whenever $f(x)=y$ is stored in the database, $f^{-1}(y)=x$ is also stored. The language used is introduced informally as necessary. The basic assertion is:

$$\mathit{inverse}(f)=g \ \wedge \ \mathit{one\text{-}to\text{-}one}(f) \Rightarrow f(x)=y \equiv g(y)=x$$

By convention, unbound variables are universally quantified, so f, g, x and y are universally quantified over this assertion.

3.1.1 Maintaining the Constraint with a Run-time Procedure

One way of maintaining the constraint is to attach a *demon* function that is executed to add the inverse whenever a property is stored. This can be specified as follows:

$$\begin{aligned} &\mathit{inverse}(f){=}g \ \wedge \ \mathit{one\text{-}to\text{-}one}(f) \Rightarrow \\ &\quad f(x){=}y \ \equiv \ g(y){=}x \\ &\quad \textbf{triggered-by } f(x){=}y \end{aligned}$$

where "**triggered-by** p" is a meta-level annotation that means whenever p is asserted (stored) the assertion should be made true.

LAC produces the following demon from this specification:

$$\begin{aligned} &\textbf{trigger } f(x){=}y \\ &\quad \mathit{inverse}(f){=}g \wedge \mathit{one\text{-}to\text{-}one}(f) \wedge \neg DB(g(y){=}x) \\ &\qquad \rightarrow DB(g(y){=}x) \end{aligned}$$

This uses a generalized demon construct which consists of a triggering event—in this case the assertion that $f(x){=}y$, and a procedure body—in this case a rule whose applicability condition (left-hand side) is $\mathit{inverse}(f){=}g \wedge \mathit{one\text{-}to\text{-}one}(f) \wedge \neg DB(g(y){=}x)$ and whose action is to make its right-hand side $DB(g(y){=}x)$ true in the new state. $DB(x)$ is true if and only if x is stored in the database. The DB predicate is used to distinguish something that is true because it is explicitly stored in the database from something being true because it is implied by the database. Thus the condition $\neg DB(g(y){=}x)$ prevents the rule from applying if its action would be redundant. This prevents the possibility of infinite and ineffectual forward chaining.

RC compiles the rule into the following Clisp code:

```
(if (db-get f 'one-to-one)
  then (let ((g (db-get f 'inverse)))
         (if (NEQ (db-get y g) x)
           then (db-put y g x))))
```

which is executed whenever a property is stored in the database. (db-get x y) and (db-put x y z) are functions for retrieving from and storing into the database, respectively. Basically what RC does in this simple example is decide the order in which conjuncts are used and how each conjunct is to be used either tested or used to bind a variable.

3.1.2 Maintaining the Constraint with In-line Code

An alternative way to maintain the constraint is to add in-line code, specified as follows:

$$\begin{aligned} &\mathit{inverse}(f){=}g \ \wedge \ \mathit{one\text{-}to\text{-}one}(f) \Rightarrow \\ &\quad f(x){=}y \ \equiv \ g(y){=}x \\ &\quad \textbf{compile-in-line } f(x){=}y \end{aligned}$$

where "**compile-in-line** p" means that whenever code that makes p true is being compiled, add extra code to make the assertion true.

The compile-time rule procedure for adding this in-line code is:

$$\begin{aligned} &a{=}\text{`}\mathit{Satisfy}(f(x){=}y)\text{'} \\ &\quad \wedge \ \mathit{inverse}(f){=}g \ \wedge \ \mathit{one\text{-}to\text{-}one}(f) \\ &\rightarrow \ a{=}\text{`}\mathit{Satisfy}(f(x){=}y \wedge g(y){=}x)\text{'} \end{aligned}$$

which, for example, transforms $\mathit{Satisfy}(\mathit{lhs}(m){=}n)$ into $\mathit{Satisfy}(\mathit{lhs}(m){=}n \ \wedge \ \mathit{lhs\text{-}of}(n){=}m)$ where $\mathit{inverse}(\mathit{lhs}){=}\mathit{lhs\text{-}of}$. The forms in single bold quotes act as patterns that on the left-hand side match expressions and on the right-hand side cause new expressions to be constructed. $\mathit{Satisfy}(p)$ means change the state to make p be true. It is used as an intermediate form in compiling rules, that is later transformed into code to make the desired change of state.

The constraint may also be used to optimize a test of $f(x){=}y \wedge f^{-1}(y){=}x$ to a test of just $f(x){=}y$. It may also be used to replace $f(x){=}y$ by $f^{-1}(y){=}x$, which is useful, for example, when another rule is looking for x as a function of y.

3.2 Property Inheritance

This section shows how an implementation scheme can be described by stating a single invariant and the ways that it is to be maintained and used. *BC* derives code for each of the procedures that maintain or use the invariant from the single specification of the invariant, so all the procedures are consistent.

The type of property inheritance in this example is all members of a set having the same value for a property. For example, if all elephants are the color grey, and Clyde is an elephant, then we can deduce that Clyde is the color grey. Using VLogic these statements are:

if $x \in$ elephants $\Rightarrow \mathit{color}(x){=}$grey
and Clyde $\in$ elephants
then the database system should deduce that
color(Clyde)=grey
when asked for color(Clyde).

A scheme for doing this is for each property that has this inheritance behavior (e.g. *color*), to introduce a corresponding property that applies to the set as a whole (e.g. *color-of-all*) and connect these two properties by the property *all-prop* (so $\mathit{all\text{-}prop}(\mathit{color}){=}\mathit{color\text{-}of\text{-}all}$).

This scheme can be described by the invariant:

$$(x \in S \Rightarrow p(x){=}\mathit{p\text{-}of\text{-}all}(S)) \equiv \mathit{all\text{-}prop}(p){=}\mathit{p\text{-}of\text{-}all}$$

In the following, I refer to this as the "scheme invariant." We want to use this invariant to

compute $p(x)$ when applicable. For example, the value of *color*(Clyde) is *color-of-all*(elephants) because *all-prop*(*color*)=*color-of-all*. To maintain the invariant we need to update *all-prop* and the instances of *p-of-all*. For example, when $x \in S \Rightarrow color(x)=color\text{-}of\text{-}all(S)$ is asserted, we need to make *all-prop*(*color*)=*color-of-all*, and later, when $x \in$ elephants $\Rightarrow color(x)$=grey is asserted, we need to make *color-of-all*(elephants)=grey. These uses of the assertion are expressed by saying that it is used to compute p and used to maintain *all-prop* and *p-of-all*. The complete specification of this in *BC* is:

$(x \in S \Rightarrow p(x)=p\text{-}of\text{-}all(S)) \equiv all\text{-}prop(p)=p\text{-}of\text{-}all$
computes p
triggered-by $x \in S \Rightarrow p(x)=v$,
$x \in S \Rightarrow p(x)=p\text{-}of\text{-}all(S)$

Before looking at how each of the three procedures is derived from this specification, we mention an alternative, similar scheme to emphasize that this constraint could be used in different ways: instead of *computing p* when needed it could be *maintained*. In this case, when Clyde $\in$ elephants is stored then *color*(Clyde)=grey is also stored.

3.2.1 Computing an Inherited Property

The first case is deriving a partial procedure for computing $p(x)$ from the scheme invariant, for example computing *color*(Clyde) as *color-of-all*(elephants). First LAC converts the scheme invariant to the form $r \Rightarrow p(x)=d$ by treating the equivalence as a right-to-left implication and merging the nested implications into a single implication with a conjunction as antecedent:

$(x \in S \Rightarrow p(x)=p\text{-}of\text{-}all(S)) \equiv all\text{-}prop(p)=p\text{-}of\text{-}all$

becomes

$all\text{-}prop(p)=p\text{-}of\text{-}all \wedge x \in S \Rightarrow p(x)=p\text{-}of\text{-}all(S)$.

From this, LAC produces the partial function:

function p(x)
$all\text{-}prop(p)=p\text{-}of\text{-}all \wedge x \in S$
$\rightarrow value(p\text{-}of\text{-}all(S))$

where *value*(x) means that x should be returned as the value of the function.

3.2.2 Maintaining Inheritance Links

The second procedure is necessary to ensure that *all-prop* is stored whenever a relevant universal statement is made. For example, when $x \in S \Rightarrow color(x)=color\text{-}of\text{-}all(S)$ is asserted, it makes *all-prop*(*color*)=*color-of-all*.

This involves using the equivalence of the scheme invariant as a left-to-right implication, and using the left-hand side as a triggering condition for the procedure. The resulting demon is stated:

trigger $x \in S \Rightarrow p(x)=p\text{-}of\text{-}all(S)$
true $\rightarrow$ *all-prop*(p)=*p-of-all*.

3.2.3 Maintaining Inheritable Properties

The third procedure is necessary to store *p-of-all* when suitable universal statements are made. For example, when $x \in$ elephants $\Rightarrow color(x)$=grey is asserted, it adds *color-of-all*(elephants)=grey (assuming that *all-prop*(*color*)=*color-of-all*).

LAC converts the scheme assertion into the form $q \Rightarrow p\text{-}of\text{-}all(S)=d$ by introducing a new variable v whose value is equal to $p(x)$ and $p\text{-}of\text{-}all(S)$ in order to split the equality $p(x)=p\text{-}of\text{-}all(S)$. This converts the scheme invariant:

$x \in S \Rightarrow p(x)=p\text{-}of\text{-}all(S) \equiv all\text{-}prop(p)=p\text{-}of\text{-}all$

into

$all\text{-}prop(p)=p\text{-}of\text{-}all \wedge (x \in S \Rightarrow p(x)=v)$
$\Rightarrow p\text{-}of\text{-}all(S)=v$.

Choosing the second conjunct as the trigger gives the following demon procedure:

trigger $x \in S \Rightarrow p(x)=v$
$all\text{-}prop(p)=p\text{-}of\text{-}all \rightarrow p\text{-}of\text{-}all(S)=v$.

3.3 Default Inheritance

In many AI systems a variation of the above scheme is implemented in which a specific value of property for an individual may be given which conflicts with the value for the property given by the sets the individual is a member of. In other words, the property value stored on the set is a *default* value to be used only if a specific value for a particular individual is not known. We can express the default scheme in our logic using the *DB* predicate. The default inheritance scheme is basically the same as the direct scheme with an extra condition:

$(DB(p(x)=\perp) \wedge x \in S \Rightarrow p(x)=p\text{-}of\text{-}most(S)) \equiv$
$most\text{-}prop(p)=p\text{-}of\text{-}most$

where $\perp$ means undefined.

In fact, typically a stronger condition is used so that if there are two sets with a *most-prop* value with one set a subset of the other, then the smaller set is used. This can be expressed by adding the further condition $\neg \exists S_1[S_1 \subseteq S \wedge x \in S_1 \wedge p\text{-}of\text{-}most(S_1) \neq \perp]$. The procedures necessary to carry out this scheme are all derived similarly to the ones above.

§4 Related Work

The specification language for *BC* is logic, which can be used to express knowledge. However, the main utility of *BC* with respect to knowledge representation, is the facility with which it allows knowledge representation schemes to be described and implemented. Knowledge representation schemes may be defined that have no relation to logic. However, the ability of *BC* to use logic encourages the specifier to relate knowledge representation schemes to logic. For example, the formulation of property inheritance given in section 3.2 is in terms of sets, quantification, and relations between properties. A similar scheme inheriting properties from *prototypical elements* is a little more difficult to express because the relation to logic is less direct. Hayes and Nilsson, amongst others, have argued that knowledge representation languages should be analyzed using logic in order that they may be better understood and the different languages compared more easily [Hayes, 1979], [Nilsson, 1980]. *BC* allows logic to be used as a tool for synthesis.

Other systems for building knowledge-based systems are EMYCIN [van Melle, 1980], AGE [Nii and Aiello, 1979], LOOPS [Stefik et al., 1983] and MRS [Genesereth et al., 1983]. These systems supply a set of facilities that are useful for building knowledge-based systems. *BC* takes a more programming-oriented view in that it allows useful facilities to be programmed easily. It may be useful for a system builder to draw on a library of knowledge representation features specified in *BC*, but these may be combined flexibly and modified as needed for the particular system and tightly integrated because of their specification in *BC*. MRS, like *BC*, aims to decouple the specification language of the user from the implementation of the system. This goal is in contrast to knowledge representations such as semantic networks and frame systems where the specification language used is more closely linked to the actual implemented representations. MRS provides the user with a few implementation choices whereas *BC* provides tools for the user to specify how to compile knowledge.

References

[**Balzer, 1981**] Robert Balzer *"Transformational Implementation: An Example,"* IEEE Transactions on Software Engineering, January, 1981, pp. 3-14.

[**Barstow, 1979**] David Barstow. **Knowledge-Based Program Construction.** The Computer Science Library, Programming Language Series. Elsevier-North Holland Inc. New York. 1979.

[**Burstall and Darlington, 1977**] Rod M. Burstall and John Darlington. *"A Transformation System for Developing Recursive Programs,"* in **Journal of the ACM.** Vol. 24 No. 1. January, 1977. pp. 44–67.

[**Genesereth et al., 1983**] Michael Genesereth, Russell Greiner, and Dave Smith. *"A Meta-level Representation System,"* Memo HPP-83-28, Computer Science Department, Stanford University, December 1980.

[**Green et al., 1981**] Cordell Green, Jorge Phillips, Stephen Westfold, Tom Pressburger, Susan Angebranndt, Beverly Kedzierski, Bernard Mont-Reynaud, and Daniel Chapiro, *"Towards a Knowledge-Based Programming System,"* Kestrel Institute Technical Report KES.U.81.1 March, 1981.

[**Green and Westfold, 1982**] Cordell Green, Stephen Westfold. *"Knowledge-Based Programming Self Applied,"* in **Machine Intelligence 10.** Ellis Forward and Halsted Press (John Wiley). 1982.

[**Hayes, 1979**] P. J. Hayes. *"The Logic of Frames,"* in B. L. Webber and N. J. Nilsson (eds) **Readings in Artificial Intelligence.** Tioga Publishing Company, Palo Alto, Ca., 1979.

[**Nii and Aiello, 1979**] H. Penny Nii and Nelleke Aiello. *"AGE (Attempt to Generalize): A Knowledge-Based Program for Building Knowledge-Based Programs,"* in **Proceedings of the Sixth International Joint Conference on Artificial Intelligence.** Tokyo, Japan, 1979, pp. 645–655.

[**Nilsson, 1980**] Nils J. Nilsson, **Principles of Artificial Intelligence.** Tioga Publishing Company, Palo Alto, Ca., 1980.

[**Phillips, 1982**] Jorge Phillips, **Self-Described Programming Environments: An Application of a Theory of Design to Programming Systems.** Ph.D Thesis, Electrical Engineering and Computer Science Departments, Stanford University, 1983.

[**Stefik et al., 1983**] Mark J. Stefik, Daniel G. Bobrow, Sanjay Mittal and Lynn Conway. *"Knowledge Programming in Loops,"* in **The AI Magazine** Vol. 4 No. 3, 1983, pp. 3–13.

[**Teitelman and Masinter, 1981**] Warren Teitelman and Larry Masinter, *"The Interlisp Programming Environment,"* Computer, Vol. 14, 4, April 1981.

[**van Melle, 1980**] William van Melle, **A Domain-independent system that Aids in Constructing Knowledge-based Consultation Programs.** Ph.D. Thesis, Computer Science Department, Stanford University, 1980.

[**Weinreb and Moon, 1981**] Daniel Weinreb and David Moon. **Lisp Machine Manual.** Symbolics, Chatsworth, Ca., 1981.

[**Westfold, 1981**] Stephen Westfold *"Documentation for TINTEX,"* Internal Report. Kestrel Institute. Palo Alto, Ca., 1981.

[**Westfold, 1984**] Stephen Westfold, **Logic Specifications for Compiling.** Ph.D. Thesis, Computer Science Department, Stanford University, 1984.