The Death Of Computer Languages,
The Birth of Intentional Programming

Charles Simonyi

September 1995

Technical Report
MSR-TR-95-52

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

THE DEATH OF COMPUTER LANGUAGES, THE BIRTH OF INTENTIONAL PROGRAMMING

Charles Simonyi
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399 USA

Introduction

This lecture series sponsored by Prof. Randell has a long-standing tradition of grand overviews of the state of the art in some aspect of computer science. For this instance, Brian and I have decided to break with this tradition for a number of reasons. First, currently I am only articulate when I discuss Intentional Programming so this limited the choice of subjects. But the general topic of "Future of Software" suggests that we look forward and it also grants a certain license to speculate. It so happens that this is the first public forum where I have ever discussed these ideas and this should also lend some spice to the occasion.

I intend to discuss one possible future avenue for the evolution of our expression of computer programs which would create exciting new possibilities in the production and sharing of software artifacts. In my second talk I would like to demonstrate, using an operational implementation, how programming might be done in the future.

Why talk about the death of programming languages? Is something ailing languages? At the outset it is safe to say that we all share a feeling of unease insofar as the general state of software is concerned: development is difficult, achieving correctness is difficult, levels of software reuse are low relative to what we would intuitively expect. But how much of this has to do with programming languages as opposed to software engineering? I suspect it has to do a lot with languages, and to see why, I enjoy arranging the common properties of languages in three categories:

- Unassailable: statements which everybody, including me, believe to be good, although there might be arguments about importances. I think they are important in that they are the only unassailable and hence invariant properties. For example, I rank "efficiency" as unassailable.
- Doubtful: these are good and necessary properties except that it is widely believed that a choice has to be made and then that choice must be "right". For example "syntax" falls into this category. Needless to say there is disagreement in what "right" is, at worst due to differences in opinion and at best due to differing requirements. I call these issues "doubtful" because I believe that we have the freedom of deferring the choices to a time and place where information supporting the choice is maximal and the cost of making the wrong choice is minimal, rendering these issues routine.
- Terrible things that are accepted as normal. For example: Different languages are not compatible with each other. These things are so obvious that people do not even think about them. If pressed they would agree that they are bad things which should be alleviated when possible, not unlike automobile accidents which will happen despite our best efforts to reduce the mayhem on the highways. Again, I believe that we underestimate the degrees of freedom we possess in the software world, and therefore it may not be a waste of time to inventory the bad things even if they seem unavoidable.

Time for an admission. While this paper is organized as a derivation of Intentional Programming (IP) from first principles, it must be obvious to anyone with practical scientific experience that the historical development of IP followed a heuristic, pragmatic and iterative path, and the present logic has been reverse engineered from the results of this development for pedagogical clarity. Indeed, the pedagogical purpose is only enhanced if the pretense were lifted and the list were taken to be a description, as well as a motivation for IP. The "unassailable" items will form the basis for IP and indicate the emphasis; the “doubtful” items will be all delegated back to the users for routine resolution, and the “terrible things” will all find solutions.

So let us complete the lists. First the unassailables. The list is rather short:

- The purpose of a language is to encode the programmers’ contributions: The reasons for making this obvious fact explicit is that the list of essentials is so short that we may be in the danger of losing track of the baby in the bath water. This statement just says what the baby is. The program is, or should be, simply a representation of what only the programmer could have known; nothing more, nothing less.
- Abstraction mechanisms: The act of abstraction is the obtainment of the general in favor of the specific. Our human civilization rests on the power of abstraction insofar as the volume of specific cases handled by an abstraction is typically much greater than the overhead of the abstraction mechanism itself. This is especially true in software so the ability to abstract is key to any encoding.
- Directness (simplicity) of expression: At some level, all parts of a program are specific: either they deal with specific facts or they describe a specific abstraction. Directness of expression just means that the units that the programmer thinks of as specific correspond closely to the units of encoding. Note that this is not the same as saying that the encoding be simple because there is no guarantee at all that the programmer’s thoughts (or the programmer’s problems) are simple. All that is required here is that the mapping between the two be straightforward.
- Accessibility of desired implementation (implementation efficiency): The programmer’s goals might well include a degree of performance which depends on a particular implementation method or the exploitation of operating system or hardware features. While this requirement is far from universal, in a competitive environment performance becomes a goal for most pieces of software. This rule basically says that there should be no incentive to go outside of the normal language framework (for example by descending to machine code) even if a specific implementation is desired

In the best of possible worlds, there would be no conflict between abstraction and efficiency, indeed the best way to express a specific implementation would be by abstracting it into its computational intent. The notion of directness is also related to abstraction, in that if something is not done directly one should be able to abstract it to become direct. IP’s “intentions” are such universal abstractions

Next the doubtful ones. If you disagree with any of the points, please re-read the definition of the technical term “doubtful” above. IP in each case will focus on finding ways to delegate the choice to the user, not to try to propose new super-duper choices. IP will not solve the problems, but enable the user to make the easier, specific choice in the user’s domain. The precise methods for delegation will be described in the sequel.

- Simplicity of language: The word “simple” is sometimes used in software engineering as if it were a synonym for “good” when it is really value-free akin to, say, “small”. While the simplicity of expression of the solution is unassailable, the simplicity of the language itself is doubtful. The existing tension is between the language designer’s desire on one hand to keep the language simple in order to reduce learning and system implementation costs, and on the other, to try to optimize ease of programming in some domains and often to also consider the speed of execution. At one extreme of the simplicity scale is, of course, the fearsome “Turing tarpit”, where the speed of programming and object program execution both are brought to a virtual standstill due to simplicity of the language primitives.
- Syntax: Syntax has long been considered somewhat unimportant as the term “syntactic sugar” indicates. This term is used when there already exists some basic syntax which needs to be made more palatable for the user by the admission of the sweetener syntax, the syntactic sugar. But if, at the extreme, we consider the underlying syntax to be an abstract syntax tree, all other syntaxes become just sugar. IP takes this view. Users will directly manipulate the program tree while they

view the program using an arbitrary and non-permanent syntax. Present day syntax had been predicated on a character stream that could have been input from punch cards, or teletypes. IP's "syntax" will allow arbitrary typographical or even graphical notations. Decades ago, Algol 60 has already expressed the longing toward clarity of notation when it defined a typographically rich publication language separate from the so-called reference and implementation languages.

- Types and type checking: The type system of a language is really a compile-time sublanguage, so the usual language issues, except for execution efficiency, will apply to types as well. The benefit of types comes mainly from type checking. Type sublanguages are not "universal" in the Turing sense, so there is no guarantee that some given type-like semantics of a class of quantities in the user's program can be expressed at all. This is acceptable, because type checking is just a convenience, albeit an important one. There are two common forms of escape when the "true" type is not expressible: either an available superclass is used instead, or an explicit "coercion" is used to paper over the difficulty. An example for the former is when an integer type is used for a logical name, such as a window number. An example for the latter might come up if the window number can have its own type but this prevents it from participating in superclass integer operations, for example I/O. However there are numerous other cases where a typical modern type system is simply unable to express directly what the user wishes to create. IP will delegate the problem of exact type creation to the user by providing for universal computation in the type machinery at compile time.
- Other standards enforcement:The remarks on types are also applicable to instances where a language enforces or encourages some general software engineering discipline, such as goto-less programming, object oriented programming, use of interfaces, or modularization. In each of these cases a sublanguage exists and there is a potential tension between the user's requirements, the expressiveness of the sub-language, and the escape mechanism. The latter effectively becomes the extension machinery in cases when the requirements and the sub-language do not match.

Finally we have the list of "terrible things" which are taken for granted:

- Languages are not compatible with each other: For example, my system is in C, I can not combine it with Ada. Or, I can't use a nice feature of Ada in C, as I would use a foreign *bon mot* in normal English text. It is true that modules compiled from different languages can be often linked together, but this just shows that the difficulties are not necessarily caused by semantic incompatibility. For a permanent and maintainable combination, the declarations would have to be sharable and the code mixable.
- Languages are not compatible with themselves: For example, I share two independently developed modules in my C program. Unfortunately, both modules include a definition for a macro called X which cause a "name collision". This is as if in real life if a John Smith worked for a company as an accountant, a second John Smith could not be hired as another accountant or even as a lawyer without inordinate repercussions, such as legally changing the name of one of the Smith's or replacing the first Smith.
- Languages are not compatible with a fixed point: For example, if the string desired by the operating system, proprietary software package, or efficient algorithm is not represented the same as in the language, a potentially costly and unreliable conversion has to be employed. In effect, the languages act as additional fixed points when they should be the accommodators.
- Computational intent and implementation details are intermingled. Consider, for example the intention of selecting a member *m* from a container *c*. There are a large number of implementation possibilities, as implied by the following access codes:

```
c.m
c->m
c[m]
m[c]
c(m)
m(c)
```

```
(*c)[m]
(c>>m)&1          etc. etc.
```

Classes, inlining, and other abstraction machinery can hide implementation detail, but still leave the uniform reference at the declarations unresolved. What is the form of a class declaration for a container with a single boolean member (i.e. class *c*{ bool *m*;};) which could be implemented in any of the above ways? This is not a frivolous question because of the fixpoint problem, namely that there may be strong reasons elsewhere in the program under construction which support the fixing the implementation in a certain way. It may not be critical that any container should be smoothly re-implementable, for example as separate arrays for each member, indexed by container instance number (the m[c] example), but the issue is rather that *given* some such existing arrays, can we still consider them as containers? Saying that the already fixed implementation should be changed, or that the objects in question should not be considered containers under the circumstances ignores realistic boundary conditions on one hand and violates sound software engineering on the other.

- Progress in programming languages is limited, among other factors, by the small number of language designers.
- Many pieces of program text are not abstractable, or abstractable only in pre-processors. Examples in C are iterators, or repeated pieces of code in procedures which could be easily expressed as local procedures if such existed. Reliance on pre-processors creates a whole different set of problems in terms of naming, and the limited computing capabilities which are available.
- Meta-work (i.e. consistent rewriting of a program) is not expressible in languages even though it comprises a substantial part of programmer's workload. Lisp is an exception in this regard, but only at the cost of failing most other tests for usability.
- Domain-specific knowledge can not be injected into the compilation process. Programmers are often given complete suzerainty over run-time, but their special knowledge of the abstractions, or of the rules of combining the abstractions, constant folding, initializations, and so on, would not be used at all by the compiler.
- Programmers are actually encouraged to make part of their contributions in the form of non-machine processable text (i.e. comments). Either this text is not very useful, and then we make fun of it; or it is useful, in which case we might well ask why the information (describing perhaps an invariant, a meta-work procedure, a type limitation, test information, etc.) is not in processable form. It is only a slight exaggeration to say that every good comment in a program represents a small failure of the language. (Cobol had a good idea in this regard: the Identification Division made machine processable at least a small fraction of the information programmers typically encode into comments.) Even overview comments describing program contents or structure are often derivable from underlying program structure.
- Many "natural" notations are not acceptable. For example, $a^2+2ab+b^2$ would be an unthinkable notation in current computer languages even though its meaning is clear. Notational freedom will become more important when programmers get a say in language design. Some mathematician who develops a great type system, compile-time optimizer and constant folder for matrices with special properties may well choose to use a traditional notation.

In summary, we are seeking a system for encoding the programmer's contributions in units that correspond to the programmer's intentions, with arbitrary abstraction that does not incur run-time costs. Implementation detail should be separable from computational intent, and any implementation should be accessible for a given intention, including those matching any given fixed point. The issues of syntax, types, standards, and naming should be delegated to the programmers, but in a way that their contributions remain compatible even when different choices or tradeoffs are made by different programmers. In syntax, any notation should be acceptable. Arbitrary type calculus, meta-work, domain-specific compilation knowledge, and information traditionally kept in "good" comments should be expressible as part of the machine processable part of the program.

But why talk about the death of languages? Would not any system satisfying these requirements be considered a language by definition? To be able to answer this question negatively, I have to rely on a metaphor from the realm of biology. You might have followed the incredible flowering of evolutionary biology that followed a shift in emphasis, originally proposed by Dr. Richard Dawkins, from the evolution of species to the evolution of genes. In this new view the genes occupy the center stage, they vie for survival by replication efficiency, while the individuals in a given species are "survival machines" which the genes have built so to speak "in order to" protect and replicate themselves. This new view helps connect a lot of additional biological phenomena with Darwinian theory of evolution. Unfortunately my task is not to convince you that this is great for biology (one can easily show that it is) but instead I would like to posit a parallel structure in the world of computer languages: languages are the species, which adapt to the users needs, while the specific features of the languages are the genes (or more precisely, information "memes") that they carry. The whole lives in a nurturing and sometimes violent universe of language designers, language implementors and users.

For example "infix operator" is a meme. Algol, Fortran, and C carry it, Lisp does not. Garbage collection is another meme: Lisp and Smalltalk carry it, Algol, Fortran and C do not. Other memes can be identified which distinguish the individual languages or language families, such as "labelled common", "fi", "types" and so on.

IP represents a similar shift in emphasis from languages to memes of language features which we shall call "intentions". Each intention carries the definition of its associated notation ("syntax") and implementations ("semantics") as attached methods. A program consists of a tree of nodes, each node being an instance of some intention. In this view the languages still exist but only as quasi-ephemeral conglomerations of memes. Languages would not be designed and they would not be named, hence they will "die" as identifiable artifacts. (Apropos names: the early history of programming languages has a famous example of a language which was really just a pragmatic collection of features, named by Mr. Jules Schwartz, a rebellious programmer, as "JOVIALl" that is "Jules' Own Version of the International Algorithmic Language". This was one small step toward the intentional world).

One key property of intentions, as described in the sequel, is that they can coexist with each other "syntactically" and can be made to coexist "semantically". This means that the existing legacy code of the users is no longer an albatross retarding innovation, but a valuable asset to which new services can be tied without limit. Once encoded in terms of intentions, software assumes an invariant "immortal" form, free from inherent obsolence.

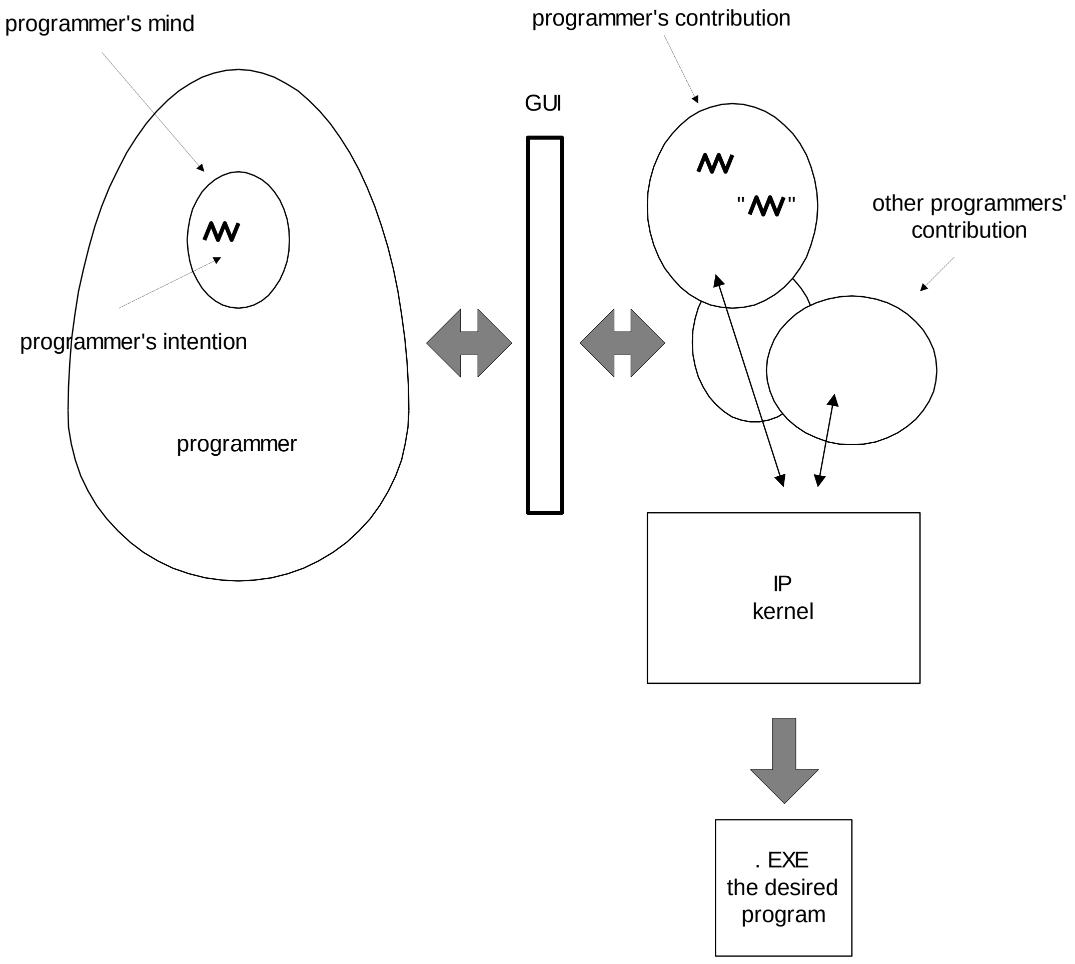
The fate of languages will roughly parallel the fate of dedicated word processors. For example, we can imagine having asked in 1974 the following question: word processing is a key need in myriad aspects of business and in everyday life so how realistic is it to talk about the death of Word Processors? In fact the question was debated at the time and many reasonable people took the conservative stance that the specialized and highly optimized and hence continually improving Word Processors will continue to be the best delivery vehicles for word processing solutions. As we know today, dedicated Word Processors have become dimestore curiosities if not completely extinct and people's word processing needs are satisfied by an incredible range of software products which run on standardized commodity computing platforms, in other words PCs. In the parallel picture, IP will be the platform which can support a multiplicity of continuously evolving intention libraries.

It is interesting to compare the eco-system of old-style dedicated Word Processor development with old-style language development. In each case mutation, hence improvement, was limited by the small number of manufacturers and language designers respectively. Stability was further ensured by direct dependence on the hardware box in case of word processing, and the dependence of legacy code on the languages. Were someone to create a new style of word processors, users would have had to buy a new hardware box; likewise a new style of language would have made it necessary that the users rewrite their legacy programs, a considerable obstacle to evolution in both instances. We can now observe the benefits of enabling evolution at least in the word processing case. Once the relatively expensive hardware box was made universal, the number of word processors skyrocketed and the quality and power of the products greatly increased. There emerged a small number of dominant word processing packages, but there are a myriad of niche products also. Moreover, the dominant products can remain on the top only by continually evolving and improving and they are orders of magnitudes better in every respect than their pre-evolutionary prototypes.

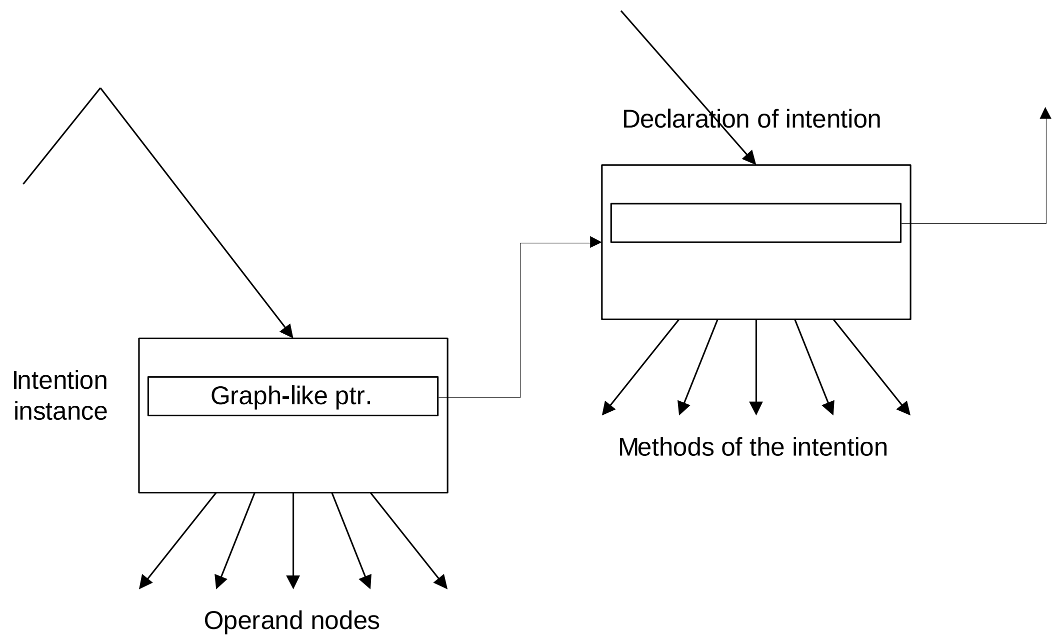
Indeed, the major promise of IP may not be the fixing of a finite number of mundane problems, but the enabling of the larger scale evolution of programming technology. One factor that enables evolution is the delegation of many decisions to the user, which raises the question of whether the user could, or would take on this additional burden. In fact, the PC software industry has been built on a model where a relatively small segment of the "user" population specializes in the creation of solutions for the rest of the users, the "end users", whenever a system is open to modifications. Even the small former segment, the "solution providers", can number in the thousands, so it is much larger than the handful of gurus who used to control the means of abstraction. There is every hope that this model of the industry will be equally valid and valuable for programming technology also.

The Source Tree

So let us provide form to the thoughts. Without being overly pedantic, we should set the stage by admitting the obvious: programming systems exists because human programmers have ideas to contribute. So we place the programmer in the picture:


programmer's mind
programmer's contribution
GUI
"
"
other programmers' contribution
programmer's intention
programmer
IP kernel
. EXE the desired program

IP maintains the contributions from several programmers who may be cooperating as a team, or the contributions can be loaded from separately acquired libraries. The contributions are in the form of a tree of nodes, called the IP Source Tree. At the time pictured here, the programmer is interacting with IP via a graphical user interface (GUI). The interface displays the source tree and facilitates its editing. A key element in programming is the forming of some intention in the programmer's mind. By programmer's intention we mean a desire that something be accomplished. Its emergence is best provoked by a direct question: "What do you *really* intend here?". The possible answers may range from "add these two numbers and remember the result", through "call a specialized instance of procedure P transformed with respect to second and third parameters and partially inlined up to medium level; adapted to the present implementation of the first parameter, etc, etc.". To actualize the programmer’s abstract desire, specific nodes are created by the programmer in the source tree. The nodes are each identified as an instance of the particular primitive intention by a “graph-like” pointer to the declaration of the intention. The nodes also contain “tree-like” pointers to any number of arguments as shown in the following figure:



Needless to say, the declarations, and the details of the intentions are themselves also intention instances and reside within the source tree, typically, but not always, in a standard library. Thus the programmers’ contributions form a “tree” of nodes. Some nodes, that is those pointed to by graph-like pointers, are declarations. Each node is identified as to its intention by the graph-like pointer. Each node can be embellished with arbitrary detail by hanging additional nodes below it.

> There is just one more small detail: nodes may also contain literal data stored as bits. So if the constant 2 needs to be stored in the tree, the bits are stored in the node and the graph-like pointer will define how the bits are to be interpreted — that is, what data representation is used. There is no requirement that such nodes be terminal nodes in the tree.

So far so good. One could say that the tree-like pointers are a little like S-expressions in Lisp, while the graph-like pointers define the object type in the object-oriented view. The source tree provides a solid substrate for representing arbitrary intentions in a modular fashion. Identification is unambiguous and invariant, via the graph-like pointers. Detail can be added at any point, without disturbing any other part of the tree.

Defining Intentions

What is missing? Just syntax and semantics, that is, looks and meaning. True to the program we set up earlier, we just invite the user (some user) to furnish the computation for:

- Imaging: to display an instance of the intention in some notation, and
- Reducing: to produce a particular implementation of the intention by performing an arbitrary program transformation on a copy of the source tree, until the copy tree contains only a limited set of primitive intentions from which machine code can be generated by the usual technologies. Once its data has been processed, the reduced tree will be immediately discarded.

Continuing the biological metaphor, we call these methods Imaging, and Reduction Enzymes respectively. Their descriptions can be hung under the declaration of the intention they describe and they are called from the

IP Kernel by special method calls. For example, when the user interface needs to show the image of the tree starting at some node N, the interface code will simply execute the “image yourself” method on the object N.

The enzymes can be written in IP itself, and new intentions may be also used in the enzymes themselves. Is this circular? Of course it is, but standard bootstrapping methods can ensure that a working copy of the system is kept at any time while the new enzymes are debugged.

How is this different from opening up the insides of a compiler, or having a retargetable compiler? First of all there is no parser there is only an arbitrary notation that the imaging enzymes produce. This means that the problems and limits of extending the notational space are merely that of comprehension and aesthetics and no longer technical in nature. Of course, this raises the issue of how the tree is input by the user, which will be treated in the next section. The second difference between IP and an open compiler is the particularly uniform and familiar way that the extensions are introduced. In this regard IP inherits the best property of Lisp: in both environments the programmer interacts essentially with the same structure as the programmer’s extension programs do. If anything, IP’s tree is better optimized for representing programs, since it is not intended to serve as the main run-time data storage abstraction as Lisp’s S-expressions do.

Completeness of Development Environment

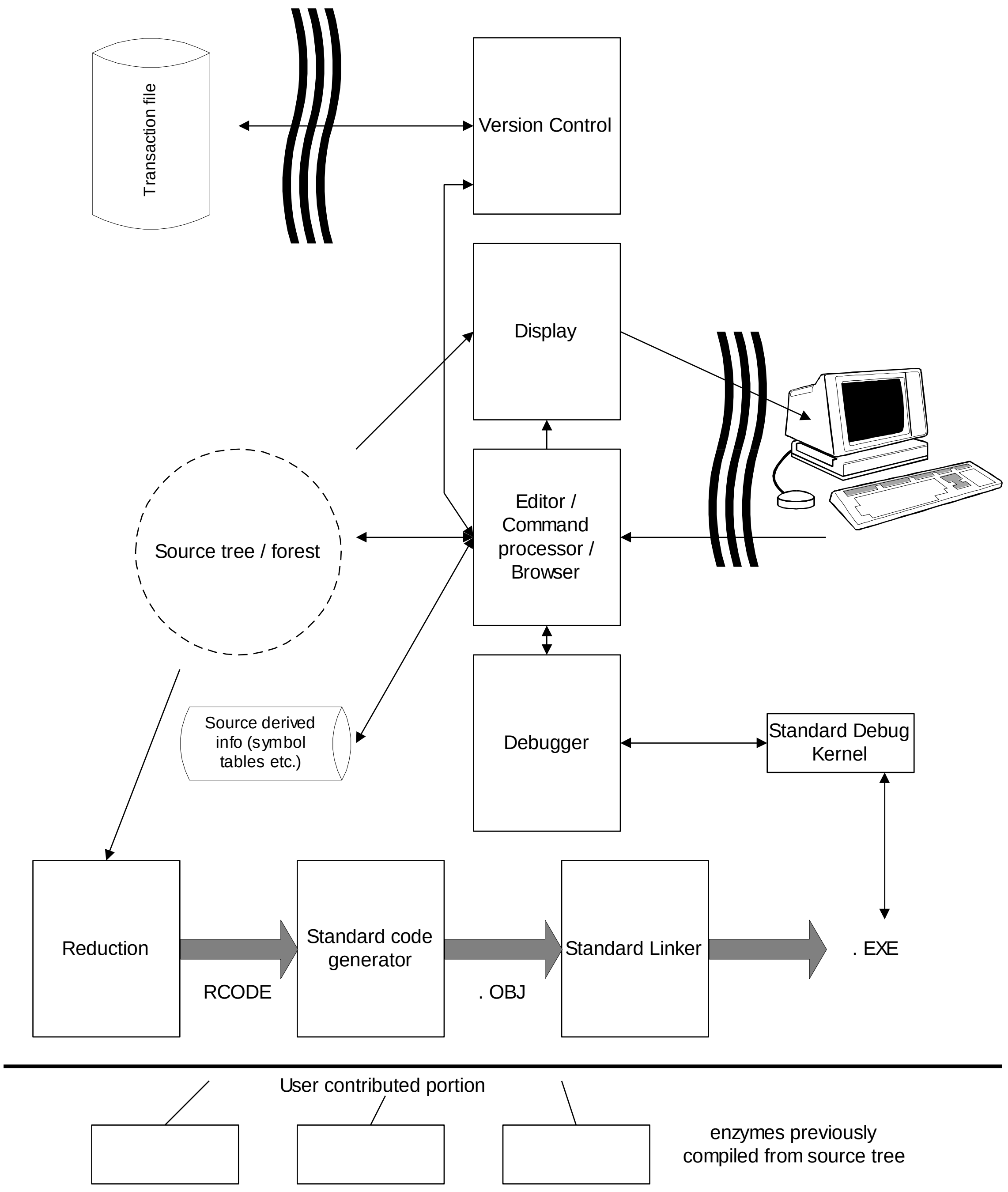
Integrated Development Environments (IDEs) are the cornerstones of modern software engineering. When one looks at Ada, C++ or Lisp IDE sales literature, the support of the IDE for the specific language is always emphasized. Of course what is being specifically supported is not the language per se, but some language features. In IP the system is divided into a shareable kernel IDE and the extended behaviors contributed by the intentions to each IDE component. These components are as follows:

- notation / display / pointing
- editor / user interface / browsing
- version control
- reduction / compilation /change propagation
- library system
- debugging
- profiling, testing (future)

When a new intention is defined, the default methods for notation, editing, version control, library access, and debugging will do a decent job in any case. Even the semantic definition can be left to default to a simple procedure call semantics. Of course, a new intention can be made much more attractive to use as more and more of the methods in the various components are redefined to match the intention specifically. For example, a new type may come with its own routines to display values of the type in the debugger. Or, the editor can perform specific computation to propose default operands when an instance of the intention is created.

It is worth noting that the development environment is also modeless insofar as the user perceives only one interface and editing of the source tree is always possible. There is no debugger, per se, there are only debugging-oriented commands. Setting a breakpoint or inserting a new statement, for example, are both accomplished using the same selection method for selecting the place where the breakpoint or new statement should go.

The following figure illustrates the components and their interrelationships:



Input

Imaging enzymes can be quite arbitrary and they work in one direction only which raises the question of how the source tree is input. If we wanted to parse the input, the imaging computation would have to be invertible. The problem is avoided by denying the premise of a strict one-to-one connection between input and display.

> A not-entirely-frivolous argument is that the premise does not quite hold even with today's parsable languages. When programmers interact with the source, they use some text editor. For example, to enter the C program text fragment:
>
> if (alpha < 0)
>
> the programmer might have actually typed:
>
> if9^(alfa^^pha <= 0)^^^

where ^ denotes a backspace, cursor positioning, mouse click or delete. The point is that programs have been always expressed in terms of editor commands and this overlay has not been considered a burden by programmers.

In IP intentions are input by direct manipulation of the tree. Various forms of selection are used to designate the operand or the place in the tree, and then the usual cut/copy/paste operations can be performed. Text typed in from the keyboard is tokenized and parsed by a very simple command processor. The tokens are presented to all declarations for recognition, and then the appropriate methods are executed on the declaration which matched the token. It is easy to arrange things so that typing "if" will result in the insertion of an "if" intention, and typing "alpha" will paste the reference to the declaration of "alpha".

The above example, if entered IP entirely from the keyboard with no errors would be:

```
if_alpha<0_
```

where _ denotes a space. This is neither less efficient, nor radically different from what had to be done in the old paradigm.

In a number of instances the departure of the input language from the image can be quite radical. For example, we found it convenient to type "cast" to enter a cast, or type "proc" to declare a new procedure, even though in the currently most popular imaging idiom, C, these appear as:

```
(...)... , or ... ...() {}
```

respectively, where ... indicates incomplete parts. Experience shows that the "burden" of having to remember the words "cast" or "proc" is not insufferable. Some users prefer to set up their keyboard interface instead so that CTRL-(can be used to enter a cast (no shift key is necessary).

Again, we can look back at the development of WYSIWYG (what you see is what you get) word processing systems. To enter italic text by strict analogy to how plain text is typed, one would need an italic keyboard, or at least a lockable "italics" shift, just like the current "Caps Lock" key. This was not done, because it would not have been sufficient, as the number of character formatting options proliferated through thin, semibold and bold, *ad infinitum;* because of practical difficulties, and because there were other perfectly intentional ways of specifying "italics" though menus, toolbars, dialog boxes, or keyboard commands such as control-I. These input methods generally do not correspond to the image of the intention, but rather they have just a mnemonic connection with it. IP extends the same strategy to computer programming.

It is worth noting that identifiers or names in IP are used only in interactions with the user. Furthermore, the use of names is not the only tool available to the user to designate identity - pointing to instances or specifying properties can be also used for identification. Once an object is identified, a link to the declaration is stored in the source tree as a graph-like pointer. This means that the name machinery can focus on optimizing communications with the programmer and be otherwise completely ignored by the semantic portion of the system.

The problem of inputting legacy code written in legacy languages is quite separate. To do this, first the features of the language have to be defined in terms of IP declarations, and then a commercial parser for the language has to be modified to produce the source tree. There are some interesting complications which stem from the use of pre-processor directives in some languages. These are discussed in the sequel.

Review of the Requirements

With the enzymes in place, IP possesses, in addition to generality of representation, modularity and invariance, the following desired properties:

- Extensibility: the features of the system are determined by declarations which a user might provide.
- Notation: arbitrary notations may be used including those that are not parsable and ambiguous. Naming is extremely flexible.
- Arbitrary semantics and implementation flexibility: since the semantics are expressed in terms of program transformations.

It is worth reviewing how IP helps with the “terrible problems” which were enumerated earlier:

- Languages compatible with each other: once read into IP with the appropriate legacy parser, the nodes are self identifying with respect to notation and semantics. Compatibility becomes a tractable issue of choosing or writing implementation methods for the language intentions which can work together. Even particularly obnoxious legacy constructs can be accommodated, for example, by providing a simple but inefficient implementation to bridge over the period during which instances of the construct are replaced by more robust abstractions. The benefit is that the system under development will remain testable while the abstractions are changed. It is also possible that the simple implementation will suffice in the first place.

- Name conflicts: since names are not used for identification, except only during initial input (including legacy code parsing) when there is ample opportunity to designate scope or otherwise disambiguate between different possibilities, name conflicts are impossible in IP. Names are still used to communicate with the user. If the user is confused by name ambiguity, there are many remedies which are guaranteed not to affect semantics. Browser commands are available to show the declarations, or the names can be changed in the declaration quickly and safely, with all references to the declaration updated in constant time.

- Computational intent and implementation detail: In IP the intent and the implementation detail can be separated. First, the intent can be expressed by creating a new intention declaration, or by hanging new attributes under an existing node. Next, the reduction transformation associated with the intention can be changed or widened to create the desired implementation.

- Abstraction: Any abstraction in IP can be implemented as an intention which does not reduce to R-code, but instead to different intentions which have to be reduced in turn. Any node in the tree can be replaced with an abstraction and formal operands can be added as required.

- Meta-work: This is just equivalent to program transformations which are in turn integral to the operation of the system. The IP editor provides a way to apply an enzyme (“Editing Enzyme”) to the source tree so that the results are permanent as if they were input by the programmer. Here are some examples of editing enzymes:
    - Lift Abstraction: Changes selected code into a procedure. Depending on various options, the variables referenced may be made into formal parameters. The original copy may be replaced with an appropriate call to the new procedure. For example, by pointing at:
      ```
      A + 1
      ```
      a procedure: PROC ???(A) A+1; is created and the original A+1 is replaced by ???(A). The unknown name ??? can be easily edited because single-point renaming is simple in IP. A variant of this enzyme can further parameterize a function. Select 1 in the procedure, and execute the enzyme. We get a new parameter and all calls will be edited, too.
      ```
      PROC add(A, ???) A+???;  ........add(A, 1)
      ```
      (assuming that the procedure name has already been changed to “add”.) The enzyme can also create names automatically according to selected naming schemes.

    - Change sequences of C #defines into C++ enums: for example
      ```
      #define A 0
      #define B 1
      ```
      would be replaced by:
      ```
      enum { A; B;} ...;
      ```

    - Apply De-Morgan’s law, or other transforms to expressions: for example
      ```
      if (!a && b) would be transformed into: if (!(a || !b))
      ```

- Domain-specific knowledge: type calculus, constant folding, and the basic node processing functions of the reducer are all extendible. Any information that makes intentional sense can be included in the source tree and appropriate processing for the information can be defined as methods in the intention declarations.

Obvious Fears

IP represent a substantial change from past practices. What are the possible downsides?

- What will happen to legacy code: IP deals very well with the legacy problem. Not only can it run legacy languages, but it is an excellent platform for re-engineering legacy code.

- Amateurs playing with syntax, causing confusion: In fact no permanent harm will come from using badly designed notations, just that interaction will be difficult while the notation is in effect. The programmer can return to a standard or commercial notation at any time without loss of data.

- Run time efficiency: Since IP uses commercial code-generator back ends, the quality of code for the usual level of abstraction should be unchanged. There is also the strong hope that by using domain specific optimizations and specialization, code efficiency can be improved.

- Editing efficiency: Direct manipulation of the source tree raises the specter of Syntax Directed Editors which have never become successful in large part because they were unpleasant to use. This issue will be discussed in the sequel.

- Efficiency of representation: measurements show that the source tree corresponding to a C legacy program is larger by a factor of 3 than the C source encoded as Ascii. However, raising the abstraction level and increasing code sharing can reduce the number of nodes in the program, so the factor of 3 should be thought as an upper limit. In addition, much of the “extra” bits in IP help browsing functions to the extent that some browsing databases become unnecessary.

- Capital costs: Unfortunately, text-based tools can not be used with IP, so change-control, source check-in, source level debugging and other important auxiliary systems have to be also re-implemented. The high capital costs of doing so might explain in part why such systems have not emerged earlier. On the positive side, the source-tree based approach, once implemented, can provide new functionality. For example a source control system now can have access not only to the history of edits that a user made, but also the semantics of the nodes which have been edited. The system can use this knowledge to merge changes made by different people, and to detect conflicts. Of course, end-users would not be noticeably affected by capital costs.

- New abstractions: are there useful ones? Only time will tell. Right now, there is a backlog of abstractions, such as partial evaluation, which have been used in research contexts only and which would have obvious uses when applied to everyday systems. In the rest of the paper we’ll be discussing a number of abstractions which make program sharing easier. We should always remember that in the old world the cost of a new abstraction was very high: typically it meant that a new language would have to be used. Consequently, the benefits had to be also very high for the abstraction to make sense. Method calls prior to C++ could be cited as a rare abstraction which took root in Smalltalk and other languages. With IP, the decision to use a new abstraction can be completely routine, not unlike the decision to use a published algorithm or a library. The initial costs are low, and the cost of reversing the decision is also limited. In effect, the IP abstractions do not have to revolutionary to be useful.

- Complexity of writing enzymes: Here the bad memory is that of “program generators” which were difficult enough to construct that their use has not become widespread despite of their power and benefits. IP has two advantages over the earlier attempts: first, the tree is particularly easy to generate with programs - a cynic would say it is optimized for generation by programs rather than by programmers - and second, new intentions may be created for the common clichés used in enzyme writing.

Notational Space

The exciting fact about notations in IP is that it will be able to evolve over time, without loss of information. Two components cooperate to create the display of the source tree: The first component consists of the set of the imaging enzymes, which are created or collected by the user. There may be more than one image for each intention, and there will always be at least one default image. The user can select, say, C-view, directing the system to invoke the C enzymes when available.

The second component is a built-in formatter which lays out the screen image, handles fonts, breaks long lines at the appropriate points, and maintains the information for decoding the user's pointing actions. For the latter, the system has to maintain a data structure which can map a particular pixel on the display back to the tree node which is "behind" the pixel. This data structure is also used to trigger the recomputation of the image when the supporting node is edited.

The interface between the components is a simple rich-text formatting language, not unlike a mini-$\TeX$. There are no plans to make the output language extendible by the user other than by copious parameterization. Its current capabilities include fonts, font modes (bold, etc.), many underlines, overlines and strikethroughs, colors, and simple formula layout (matrices, integral forms, roots, fractions, etc.) More capabilities can be added to new versions of the system. Only the imaging enzymes contain knowledge of the imaging language; the source tree has none, with a specific exception which has to do with the programmers' special formatting instructions.

In IP, the choice of notation will be just a personal decision just as indentation and comments used to be. Arbitrary manual formatting with spaces and tabs is no longer possible: the choice of view determines not only what used to be the "language" that is C, Pascal or Cobol, but also the formatting conventions, such as the K&P style indentation:

```
if (f) {
   i = 0;
   }
```

or indentation with the brackets lined up:

```
if (f)
   {
   i=0;
   }
```

As the systems develop, these styles may include more heuristic rules to create for example:

```
if (f) {i=0;}
```

Any additional control of the formatting by the programmer will require cooperation by the imaging enzymes. A simple and useful trick is to introduce an intention for a blank line:

```
if (f)
  {

  i=0;
  }
```

The semantics of the blank line is just a noop, but it is displayed without the extra semicolon. If there are reasons for formatting the same intention in different ways, the intention must have a parameter expressing the choice. For example, a comment associated with a statement might appear in different positions:

```
// comment 1
i=1;
i=2; // comment 2
```

```
    // comment 3
        i=3;
```
The choice between positions 1 and 2 is expressed in terms of an intentional property under the comment node: left vs. right comments. The special indention of choice 3 is expressed by associating a general formatting property on the comment node: outdent by so-and-so much. The main reason for worrying about these issues at all has to do with legacy code. The better job IP does of preserving the structure of the legacy code, which in the past had to be expressed as formatting and comments, the easier the re-engineering process will be for the programmers.

The imaging language allows multiple fonts and the 16-bit Unicode character set becomes readily available. The constants in the source tree are self-identifying so they may or may not use Unicode or any other encoding.

Since the source tree does not depend on names for identifying graph-like links, the only restriction on names in IP are those that the programmers choose for simplifying the input of new programs. It is possible to establish a link without using the name, just by direct pointing to the destination of the link: for example add 1 to the variable declared *there* — click. Similarly the operation "add", and in principle even the common constant 1 could be selected by clicking. However, some simple restrictions on the names make it possible to tokenize the keyboard input:
```
        J+1_
```
into Name token: J, Name token: +, Constant token: 1, which then can be arranged to have the desired effect, that is the creation of the source tree +(J, 1). But even this restriction for tokenizations does not necessarily mean that the names have to be restricted the same way. Lookup for input can be performed on the basis of partial matches. More importantly, declarations can have multiple names:

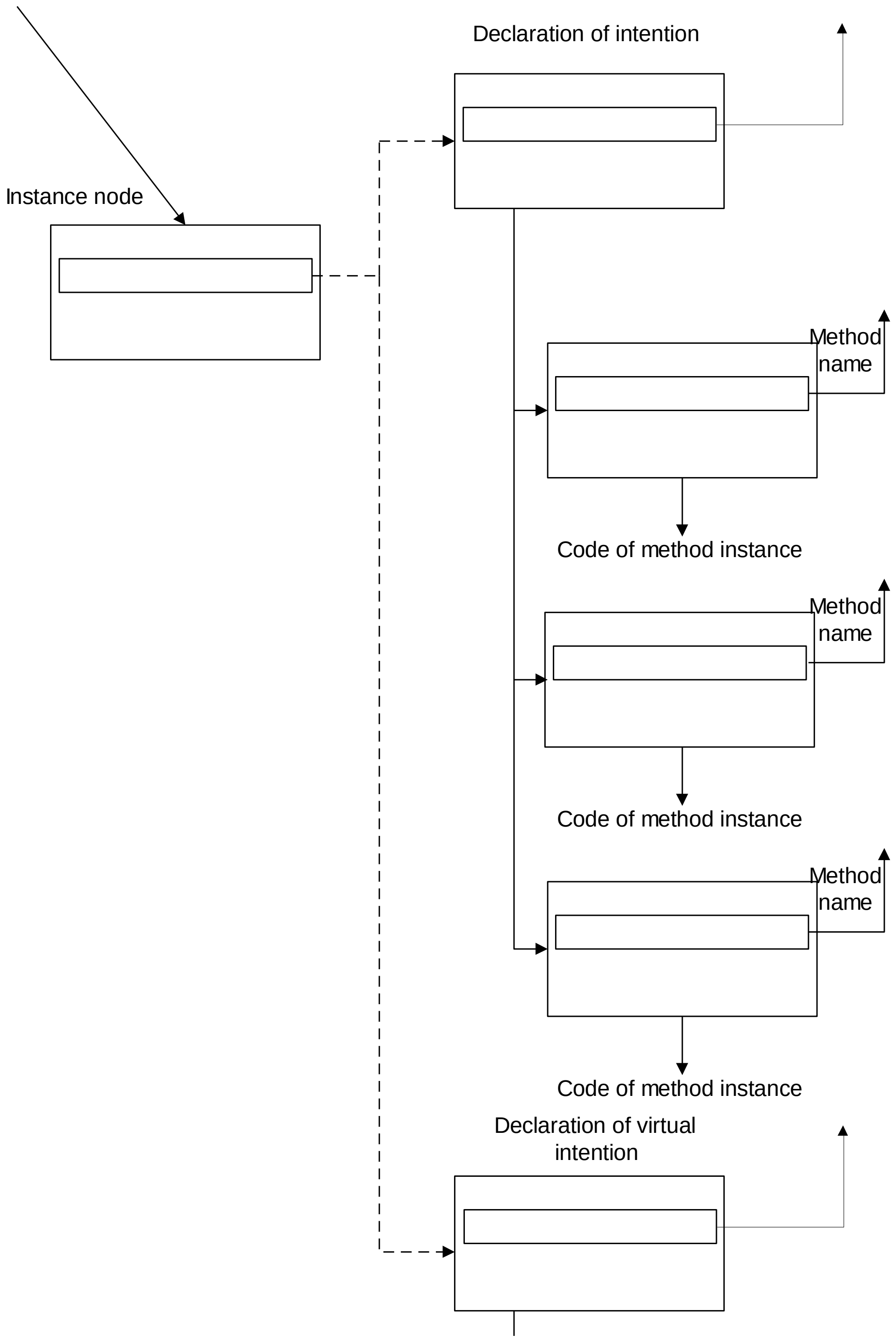
- link name which remains invariant even when the declaration is renamed
- normal, or short name used for input and for expert overview
- discursive, or long name, which could be used by programmers unfamiliar with the program
- foreign names, needed when several different natural languages are used in development

Note that the lack of restrictions on names opens up the realistic possibility of programming in kanji and Chinese, not to mention the use of Greek, cyrillic and other alphabets.

The notion of "literate programming" was introduced by Knuth so that parts of programs can be exposed in an order that helps the reader, for example in the order of stepwise refinement. IP is an obvious vehicle for implementing such displays in an interactive setting, where the same intentional structure could be displayed both in "literate" and in "literal" forms. Many other views are also possible: for example, one might want to group the object–method pairs by objects or by methods. The simplest variable view command, outlining, has already been implemented.

Extension Machinery

At the most abstract level, extension simply means that parts of the user's code may be executed by IP. This means that parts of the user's code are suitably identified as some extension of IP. There has to be a library of interface specifications which is exported by IP and to which the user extensions must conform.
Extension code is then culled together and a dynalink library is created which can be used immediately by IP. Most, but not all, extensions are associated with intentions. These follow the object paradigm, as shown in the following figure:



where the method names identify the aspect of the system that is extended by the intention, and the procedure is the contribution from the user which performs the extension. The dotted graph-like pointer indicates that an appropriate multiple inheritance scheme is provided. Instead of just following the graph-like pointer to the immediate declaration, the inheritance computation can generate a set of underlying intentions of which the node in question is an instance. For example, a node pointing to the declaration of an integer variable may inherit methods from the “virtual” intention “simple variable” and from the actual type “integer”. Effectively, the inheritance computation can classify nodes as representing several “virtual” intentions starting from, and in addition to, the one pointed to by the node. The root of the inheritance hierarchy is implemented entirely in the

IP kernel. It describes the most common behaviors for intentions as the default. This means that an intention need not define all methods. For example, a procedure can use the existing call semantics by default but define its own imaging. Or, a new intention with new semantics can still use the default parenthesized functional notation as its imaging. The inheritance computation itself can be extended by “come from” hooks, described in the sequel.

Clearly, extensions are only possible when there is some recognition by the creators of the IP kernel of the need for extension. Once this is achieved, the new method name is defined in an interface file. The default implementation for the method gets coded and the appropriate method calls get inserted into the IP kernel. Now the users can define new extension methods and this will not disturb the existing extensions. The fact that methods can be defined for behavior in all parts of an integrated development environment also distinguishes intentions from classes. Classes differ from each other only during execution of the program. They cannot be individually distinguished by any other part of an integrated system: the editor, version control, change dependency propagation, browser, or the debugger. Intention can define specifically appropriate behavior in all of these components.

As suggested earlier, some desirable extensions may not be associated with any given intention, for example if a change in the inheritance mechanism is desired. Also, there are many issues raised by the modularization of the information necessary for extension. What if library (or user) A desires to add a method to an intention from a different library B? What if type T of an interface definition should be an opaque box for the user, but a detailed structure for the implementing routines in IP? Fortunately, IP is implemented in itself, so the required intention can be pragmatically defined and used to solve the problem.

Editing

Probably the least natural aspect of IP seems to be the requirement that the source tree be edited as a structure and not as text. Users have historically accepted direct manipulation for graphics, for example. However, when the source data is shown in terms of characters, the tradition has been to edit the source as it were a character stream. The *syntax directed* editors have come the closest to approximating the IP editing problem. In the syntax directed paradigm, the syntax rules are immediately applied to the user’s input, with two important results: first the input of illegal constructions is prevented at the outset; and second, as a result of parsing, the system has knowledge of what has been typed and this knowledge can be used in turn to help the user with prompts, automatic formatting, defaults and so on. Despite these advantages, syntax directed editing has never been a commercial success. One can only surmise the reasons for the user’s dislike of such systems, but one good guess is that a successful system should be modeless and intentional. Modelesssness means that work on one section of the code can be always suspended and the focus transferred to some other portion. Intentional here means that the editor commands should directly relate to the intentions rather than the representation of the intentions.

Modelessness is important because programs are seldom constructed in a strict order. The programmer must be able to express code fragments whether or not they make sense in their intermediate forms. There should be no interruptions, demanding prompts, or beeps. IP does have two subtle forms of reminders: incomplete operands are typically highlighted in color, and there is also a “To Do” list with current references to all incomplete or otherwise incorrect code as discovered by a background type checker. After finishing some coding the programmer can simply scan the screen for highlights and then scan the todo list for any remaining errors. Of course, there will always be a few obscure errors which will be uncovered only during the reduction/compilation of the program.

The editing model is based on the standard cut/copy/paste operations, but with about seven different selection types as opposed to the single text selection used by text editors. Just one of the selection types, for example, is necessary for selections within constants, names or comments which can all be edited as text. The other types select subtrees, lists of subtrees, individual nodes, places in lists, and left or right operands for operations. While this is quite a bit more complicated than basic word processing, the programmer users are well equipped and motivated to handle the conceptual load.

Having a reliable and multi-step undo facility also help create a comfortable environment where the consequences of mistakes are very much limited and where experimentation is encouraged.

Example for an Intention

Let us consider the construction of a new intention for a dialog box, as in a user interface for a program. The intention is not a class: the class already expresses some data structure that is a particular implementation of dialog boxes. The intention is simply a repository of the user’s contribution. So the process of defining the intention starts with cataloguing the independent quantities, such as:

- size (width/height)
- title
- collection of items, for each
    - type of item
    - position
    - title
    - code to define default value
    - code to call after value changed
    - specific fields depending on type of item
    - button:
        - code to call when button is pressed
    - radio buttons:
        - collection of items, for each
            - position
            - title
            - value to be returned when selected
    - etc. etc.

This process can be elaborated without limits. The quantities in the intention may be identified by labels or by positions. The quantities themselves may be constants, expressions, code, pointers to declarations, graphics or whatever is required by the problem: their presence does not commit to any particular form of processing or interpretation yet. If there are shared quantities, they should be placed in a declaration, and then the sharing is expressed by graph-like pointers to the declaration (for simplicity, the above list did not include sharing).

Constants can be interpreted in the context of the intention. Some constants will be lengths, these can be encoded as discrete choices, taken from the analog length of a line segment, or as subtrees or character strings with units provided, for example 1 3/4” or 10cm.

One nice thing about intentions is that they can be extended even when they are already in use. Of course, the order of parameters which were identified by position may not be disrupted, but labeled or subsidiary information can be attached to any node without disturbing the existing uses. In the most extreme cases, an editing enzyme could be supplied to perform the transformation necessary for an upgrade.

Once the intention is formalized, the first implementation must be decided. This may be as simple as ignoring everything initially, or may use a sophisticated class-based library. In either case, the Reduction Enzymes have to be written. The implementation may be changed at any time.

An Imaging Enzyme can be also written which can echo all contributions and perhaps also provide a preview of the dialog box. A more sophisticated implementation of the interface code in the future might provide tools to allow direct manipulation of the graphical items as well, so that the programmer could adjust the size parameters in the intention by dragging the borders of the preview. Failing any this, the intention could be still displayed and input in standard functional notation.

Useful Abstractions

The prediction is that IP's combination of expressive power and availability will give rise to an incredible burst of creativity from many corners. The purpose of the following examples is not to demonstrate unique creativity but to illustrate the degrees of freedom. Especially in the beginning, the plate will be full with grand old ideas which somehow have gotten stuck in some niche language such as Lisp, Smalltalk, or even BCPL. Just bringing these abstractions into general use will be a major undertaking with far reaching consequences.

Valof. A short-lived feature of BCPL was the "valof" primary which could be used to write an arbitrary computation as part of an expression, for example:

```
        sum=valof{int i, sum =0; for (i=0,i<20;i++)sum+=a[i]; resultis sum; }–check_sum;
```

(where the resultis construct defines the result and breaks to the end of the valof braces) It got left out of C, probably on the valid basis that the same code could have been written more clearly as an equivalent sequence of statements. However, the construction is invaluable as a way station in a series of transformations. If an enzyme were to expound on the expression:

```
        sum = Σa – check_sum;
```

it might well wish to choose the inline loop as the implementation of the summing intention, and it would prefer to do the substitution in place, without the need of analysis. In effect, valof is the key primitive ingredient of the various inlining features of languages such as C++. But how can valof itself be implemented? It is worth enumerating the ways, because they are typical for a whole class of primitive intentions:

- The necessary complex analysis may be performed to displace the code to the nearest legal place which is executed before the actual use and use a temporary variable to transfer the value.
```
    int T;
    int i, sum=0; for (i=0;i<20;i++) sum+=a[i]; T = sum; goto end; end:
    sum = T – check_sum;
```
- Use a trivial if suboptimal implementation to verify that the abstraction is useful. Frequently, the benefits of use outweigh the inefficiencies, or problems unrelated to efficiency can be identified once an implementation is available. When the feature proves itself, new implementations can be introduced without negating in any way the investments made into using it, that is without having to change existing code. The trivial implementation for valof is to introduce a function:
```
    int T(int a[])
            { int i, sum =0; for (i=0,i<20;i++)sum+=a[i]; return sum; }
    .....
    sum = T(a) - check_sum;
```
- If the abstraction is useful and sufficiently basic, it can be absorbed into R-Code, which simply means that the optimizations of the code-generating back end are allowed to posses knowledge of the abstraction in question.

Separating computational intent from implementation detail. The idea here is to provide parameters ("annotations" and "stipulations") to declarations or operations which determine how the intentions should be implemented; that is, what reduction enzyme should be applied. A few clarifications are in order: IP allows an arbitrary number of arguments to any node, but how would one separate annotations from standard operands? The answer is essentially that intentions for annotations are expected to answer true to the "are you an annotation?" method. (In the rare cases when annotations themselves must be operands, special intentions, which ignore this answer, provide the required context.) By separation, IP means an arbitrary distance, so that the change of implementation can be effected without having to change the code that uses the intention in question. But the choice of implementation has to be made at the site where the intention is used, so how does the information get there? By successive abstraction. Annotating the actual intention instance would be possible but probably rare. Next, one could annotate the declaration of an operand, the declaration of the type of the operand, and so on, at each level affecting more and more quantities with fewer and fewer annotations. Implementation information can be also propagated through procedure interfaces using "specialization" (cf.) and completely remoted from its place of activity by "proxy contributions" (cf.).

Finally, the notion of “implementing” is to be interpreted under the assumptions that are implied by the annotations and stipulations. For example, a four-element fixed array may well be an “implementation” for a general collection intention, provided that the number of elements in the collection will never exceed four, the element sizes match the array type, zero is available to mark the absence of an element, etc., etc. Annotating the collection with the name of the implementation (that is, the reduction enzyme) would be understood by IP as a stipulation by the user that sufficient conditions for the use of the implementation are satisfied. Or, conventions could be introduced where the specific stipulations could be made by the user which could be interpreted by the enzymes to see which enzyme should be awarded the job.

But is it worthwhile to optimize code to the extent implied by this facility? The answer may or may not be yes, but the question could be more usefully refocussed on the act of writing the code in the first place. It has been said that “premature optimization is the root of all evil”, yet all programmers have experienced the quandary of writing code that has a chance of performing well versus writing a functional prototype which will have to be rewritten for performance. Where does “designed-in performance” end and “premature optimization” begin? IP removes the quandary. When you have the urge to commit premature optimization, you can express both the prototypical intention and the more complicated implementation at the same time. Perhaps the expression of the implementation as an enzyme may be harder than writing the code directly but as a bonus, the particular implementation trick now becomes a sharable and marketable object, to be used whenever a similar situation might arise. Also, the optimization, premature or not, becomes replaceable by other implementations with different performance characteristics.

How is this different from classes in Object Oriented Programming? First, the implementation choice applies to every intention, not just objects, so the system does not have to be one-hundred-percent pure object oriented to have the benefits of separation. Even when the object paradigm itself is considered an intention, it might benefit from various implementation models that do not involve OOP. Conversely, OOP itself includes many implementation choices and different structures might benefit from different implementation choices. Second, under IP any implementation tricks are acceptable, not only those which are expressible in terms of methods. For example a procedure might request space in the caller’s frame, and demand a pointer to it to be passed. A field in a structure can be a fixed size array, a relative pointer to a variable size array, with the size stored separately, or a heap pointer to an allocated block with the size computed from the block size. These choices could be all simulated in OOP at one level, but you could not define an embedded object which, if included into another class as a field, would create any one of the above choices. Thirdly, in IP the “independent” quantities which form the programmer’s contribution to an intention can be kept textually separate and without any impact on the run-time program, from the “dependent” quantities which comprise the implementation. In OOP, similar separation within a class is impossible and can be only approximated by public/private labels, comment, and program formatting. The form of expression of the intention is difficult to separate from the run-time representation in that anything a programmer writes in a class constructor, for example, has to have some run-time existence by definition.

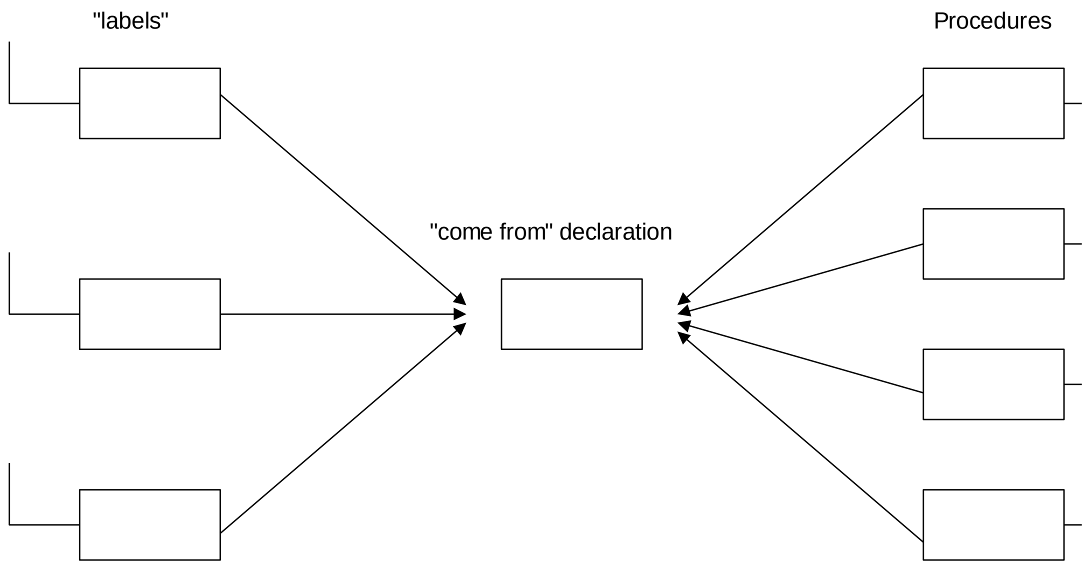
The intention/implementation abstraction does not replace procedures. Roughly speaking, a procedure is an intention with one implementation which is in the form of a subroutine. Casting the implementation into transformational form becomes effective only when an intention will admit to multiple possible implementations, or if the actual parameters expressing the intentions differ markedly from the formal parameters which implement the intention.

Specialization. The act of specialization is satisfying a procedure call by the creation of a new procedure instance by partial evaluation of the old instance with respect to some additional information about the values of the actual arguments, typically the information that some argument has a given constant value. This should have absolutely no effect on the results but for two very important factors: First, some interesting quantities must be compile time constants: for example static types, implementation choices, field selectors, code sequences, efficient array limits. Absent specialization, formal parameters to procedures are, by definition, variable. Ergo: many interesting quantities cannot be used as parameters to procedures unless one has specialization. Specialization thus removes the incentives to use source-level macros or other preprocessor constructs when abstractions with compile time constant formal parameters are needed. Much clutter in C production code is caused exactly by the macro definitions and “ifdefs” which express otherwise quite normal abstractions, but using a very primitive and unattractive language.

The second key factor that favors specialization is efficiency. Existing procedures can often be speeded up in key contexts by judicious specialization. “Inlining” is a common feature in programming languages that achieves the same or greater performance improvement than specialization, except that inlining is profligate in its use of code-space. In any case, specialization provides for a spectrum of tradeoffs. But the most important effect of efficiency will be not on existing code, but on new code. Once it is clear that parametrization does not involve run-time costs (nor a change in the mode of expression, for example, having to use macros instead of procedures) the programmer can express more and more variability in the code in terms of parameters. This will give rise to larger, very general, “archetypal” procedures which will have one or maybe a few instantiations in any given using program. The archetypes would be extremely reusable because of their generality. Also they would be *usable* because from any single program’s point of view they would be as efficient and have as few free parameters as if tailor-made. In the 1968 Nato conference McIllroy made the famous prediction that a universal component library could be developed and it might contain “100 of just sine routines alone”, not unlike a hardware component catalog in which there are at least 100 kinds of op amps. This prediction has not come true maybe because 100 is too small — when the combinatorics of possible number representations, wordlengths, desired precisions, preferred approximations, space/time tradeoffs, etc. are taken into account. The number 100 is also too large, when one considers the size and organization of catalogs and deliverables, and the problem of the marketer and customer to exchange information about supply and demand.

With specializations, the universal library would have just one sin routine (or perhaps just one trig archetype where the particular trigonometric function is also just a parameter) and users would be able to specify, or let default, a dozen or more parameters. The number of possible specific routines which could be gotten from the archetype would be combinatorially huge.
.

Come from statement. This novel abstraction helps with hiding information between modules. It stands in contrast with a normal procedure call, where the caller knows what procedure to call, but the procedure only knows what it implements, not who might call it. In a “come from” procedure, the potential call site, the “come from label”, knows only what invariants are true at the site, but does not know what procedures, if any, might be called. The procedures, in turn specify what “come from labels”, that is under what conditions, they should be called from.



For example, when cache C is empty, the following come from label may be executed:
CacheEmpty(C):

and the following procedure would be called:

proc comes from CacheEmpty(CACHE c) { ... }

note that parameters would be allowed. All this looks perfectly normal except for the fact that the procedure must also import the name "CacheEmpty" just as the label does, and that there could be, say 3 labels and 4 procedures referring to the same name. At each of the 3 labeled places, 4 calls, one to each of the 4 procedures would be executed. Information hiding would be perfect as the only definition they would all have to know is the name of the invariant they all apparently have a stake in.

Proxy contributions. This intention is another abstraction to help maintain information hiding, without having to resort to run-time interpretation. For example, to extend the behavior of an intention defined in a system library, the method can not be written as part of the declaration, instead, it is "contributed" by proxy. In a similar manner, a module can add fields to a data structure, or add values to an enumerated type that it does not own.

Legacy Programs

IP promises to be a great vehicle for re-engineering legacy code. The method for incorporating a legacy language L into IP is as follows:

1. Identify the intentions in L either with existing intentions (such as +, int, goto) or with specific new intentions (such as MOVE CORRESPONDING.)
2. Adapt an existing parser for L to create the source tree of these intentions. It helps if this parser is written in a language for which a legacy parser already exists (such as C.) Otherwise, the parser has to be re-coded..
3. Choose implementations for the new intentions and define the transformations for them. At this point choices have to be made about semantic compatibility with existing intentions. Fortunately, these choices will not be permanent, so engineering tradeoffs can be made between requirements, schedule, and implementation costs. If necessary, compatible implementations may have to be added to existing intentions.
4. Typically, imaging enzymes ought be defined to be able to display the code true to the language L. This is very easy to do and will help a lot in verifying the parser has worked and attracting legacy users. By the way, it is acceptable if intentions outside of L, and even rare L intentions do not have imaging enzymes: the inherited or default notations will typically suffice in such cases.

Step 3 might still be a sizable project for a large language such as Cobol, Pl/1, or Ada. Of course the incentives for an independent software vendor to do this work will be also very large. For smaller, locally popular languages, where the semantics are more standard than the syntax, the total costs might be of the order of a month of a programmer's time depending on the availability of a parser to start with. We have implemented a full Pl/m system in IP in addition to the C legacy system that was used for bootstrap. With Pl/m in place, a private collection of legacy code written for some long defunct platform found a new lease on life. Indeed, if there will ever be a computer museum for languages, IP would be an ideal framework in which dead languages could be resurrected and maybe even find use in a luxurious new development environment which would be beyond the languages' creators' wildest dreams.

Once a program written in L is imported into IP, it is open to re-engineering by continuous improvement. Instances of inconvenient old intentions could be removed or consolidated one-by-one and the program would remain in runable and testable state throughout the process. Old parts could be viewed in L or in C, while new parts would show in C (or whatever the best available notation is.) The re-training of programmers skilled in L could also proceed in parallel with the improvement of the program code. Large-scale systematic changes to the legacy program could be effected by writing editing enzymes.

One interesting question of legacy parsing is what to do with pre-processor information and with comments? As to comments, they historically have comprised an important part of the programmer's contributions and must be preserved. The real issue is to develop heuristics as to what node in the tree the comments should be attached to, that is to parse the tacit syntax of comment bindings.

In languages such as C, much of the "intentional" information is encoded in macros. For example, consider the C fragment:

```
#define transform(x) x + 5
...
print(transform(a));
```

It would be a tragic loss of information if the legacy parser's total output would be:

```
...
print(a+5);
```

which is what a typical compiler rightly expects from its parser. Optimally, we would expect the IP legacy parser to recover the underlying intention:

```
anytype transform(callbyname anytype x) {return x+5;}
...
print(transform(a));
```

Again, this calls for heuristics which go beyond the original language definition, but which, defined and applied judiciously, not only preserve more of the intention of the source, but also serve as a first step to re-engineering away from macros.

Experience has shown that it is useful to classify C macros into a few categories:

- constant-like: such as #define a 1
- procedure-like: such as #define transform(x) x+5
- token-like: a catch-all category, meaning that calls to it must be expanded and that information will be unfortunately lost. The expansion may be labeled with the name of the macro so that it may be identified by an editing enzyme during re-engineering.
- statement-like: #define forever for(;;) // note last "parameter" is the next statement
- type-modifier-like: #define LINK **

Similarly, #ifdef's can be also classified into "token-like" which will be expanded with information loss and "honorable", which can be made intentional. Some classification can be done automatically, and some of the rarer ones must be classified by the user using a pragma comment. Using the above classification and a few intentions which express the underlying intent, practically all macros and ifdefs in very complex programs can be intentionalized. For example, an honorable ifdef applied to statements can be expressed as a simple if. However when it is applied to a declaration, it is expressable only as a new intention: an annotation attached to the declaration which contains the condition under which storage need to be reserved for the declaration.

Business model

As mentioned earlier, some users of IP will specialize in creating packages of intentions and collaborate to establish standards which ensure semantic compatibility of the intentions. Let S denote the number of standards, I the number of intentions, and U the number of end-users. The expectation is that S will be small, I large, and U very large:

$$S << I << U$$

I will grow because the market will the very large U. This is as opposed to today where the market for language features is stagnant, maybe with a few big sales every decade.

The number of standards will be small, but standards can be changed if the benefit to U exceeds the cost of recoding the I's, in other words the shift will be possible when cost/benefit < U/I which will be in the thousands. The cost would be borne by the ISVs (independent Software Vendors) but they would in turn charge the users according to the benefit of the change. This is very much different than today, when the cost/benefit must be less than 1 for change to take place.

Pressures from U would be satisfied by new I's. There will be very few impediments in the way to growing I. In the old world, pressures from U would have to be handled by essentially S. Now, pressures from I's would create new standards, as described above. Accumulated difficulties faced by ISVs would have to be relieved by periodic updates to the IP kernel, extending the ways that the kernel itself can be extended.

The market will be very large because of the absorption of more and more legacy languages, and because of the increase in code sharing due to the separation of computational intent and implementation detail, and because of specialization.

Summary and Status

We have presented the idea of the intention as an abstraction mechanism, and an integrated development system which may be used to develop systems using intentions. Software encoded intentionally can be said to be immortal, in that its meaning can be sustained independently of the long term progress in programming notation and implementation techniques. The independence and self-sufficiency of intentions might well create first a market in abstractions or "language features", followed by the long sought-after dream of a software componentry market. Legacy code can be integrated into the new paradigm with minimal or no loss of information and there are considerable prospects for "hot" re-engineering or continuous improvement, which can be performed while the legacy system is kept in operating condition.

IP is currently under development at Microsoft Research. Several US Patents have been applied for, covering various aspects of the system. The system achieved complete self-sufficiency March 1, 1995, and since then all further development of IP has been performed in IP itself. The size of the system as of September 1995 was about 1.7M nodes (intention instances) in the source tree. Plans include the creation of component libraries; the support of additional legacy languages, such as C++; operational use of the system elsewhere within the company; and finally productization before the year 2000.

References

ABRAHAMS, P. W., Typographical Extensions for Programming Languages: Breaking out of the ASCII Straitjacket.

BACON, D. F., GRAHAM, S. L., SHARP, O. J., Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, Vol 26 No 4, December, 1994.

BALLANCE, R. A., GRAHAM, S. L., VAN DE VANTER, M. L., The Pan Language-Based Editing System for Integrated Development Environments. *SIGSOFT*, 1990.

BASSETT, P. G., Frame-Based Software Engineering and Iterative Design Refinement. *Software Engineering: Tools, Techniques, Practice*, April 1991.

BASSETT, P. G., Frame-Based Software Engineering. *IEEE Software*, July 1987.

BATORY, D., O'MALLEY, S., The Design and Implementation of Hierarchial Software Systems with Reusable Components. *ACM Transactions of Software Engineering and Methodology*, Vol 1, No 4.

BERLIN, L., When Objects Collide: Experiences with Reusing Multiple Hierarchies. *ECOOP/OOPLSA '90 Proceedings*, Oct 1990.

BOSWORTH, G., Objects, not classes, are the issue. *Object Magazine*, December 1992.

BURSON, S., KOTIK, G. B., MARKOSIAN, L. Z., A Program Transformation Approach to Automating Software Re-engineering. *IEEE*, 1990.

CAMERON, R. D., Efficient High-level Iteration with Accumulators. *ACM Transactions on Programming Languages and Systems*, 1989, Vol 11, No 2, pp. 194 - 211.

CHEATHAM, T.E. Jr., Reusability Through Program Transformations. *IEEE Transactions on Software Engineering*, Vol SE-10, #5.

COHEN, H. H., Source-to-Source Improvement of Recursive Programs. Ph.D. dissertation, Division of Applied Sciences, Harvard Univ., May 1980.

DEWAR, R. B. K., GRAND, A., LIU, S., SCHWARTZ, J.T., Programming by Refinement, as exemplified by the SETL Representation Sublanguage. *ACT Transactions on Programming Languages and Systems*, Vol 1, No 1, pp 27-49.

DEWAR, R. B. K., SHARIR, M., WEIXELBAUM, E., Transformational Derivation of a Garbage Collection Algorithm. *ACM Transactions on Programming Languages and Systems*, Vol 4, No 4.

DYKES, L. R., CAMERON, R. D., Towards high-level editing in syntax-based editors. *Software Engineering Journal*, July 1990.

FEATHER, M. S., A Survey and Classification on some Program Transformation Approaches and Techniques. *ACT Transactions on Programming Languages and Systems*, Vol 13, No 3, pp. 342-371.

GRISWOLD, W. G., BOWDIDGE, R. W., Program Restructuring via Design-Level Manipulation. *Proceedings of the Workshop on Studies of Software Design*, Baltimore, May 1993.

GROGONO, P., Comments, Assertions, and Pragmas. *SIGPLAN Notices*, Vol 24, No 3.

JORDAN, M., An Extensible Programming Environment for Modula-3. *ACM*, 1990

KAELBLING, M. J., Programming Languages Should NOT Have Comment Statements. *SIGPLAN Notices*, Vol 23, No 10.

KOTIK, G. B., MARKOSIAN, L. Z., Automating Software Analysis and Testing Using a Program Transformation System. *ACM*, 1989.

KOTIK, G. B., ROCKMORE, A. J., SMITH, D. R., Use of Refine For Knowledge-Based Software Development. *IEEE*, 1986.

KRUEGER, C. W., Models of Reuse in Software Engineering. Carnegie Mellon Report CS-89-188, December 1989.

MERKS, E. A. T., DYCK, J. M., CAMERON, R. D., Language Design For Program Manipulation. *IEEE Transactions on Software Engineering*, Vol 18, No 1.

MINÖR, S., Interacting with structure-oriented editors. Lund University, Sweden

PARNAS, D. L., SHORE, J. E., ELLIOTT, W. D., On the Need for Fewer Restrictions in Changing Compile-Time Environments. *Naval Research Laboratory Report*, 7847.

PRIETO-DIAZ, R., Status Report: Software Reusability. *IEEE Software*, May 1993.

RIEHLE, R., Objectivism: “Class” Considered Harmful. *Communications of the ACM*, August 1992, Vol 35, No 8.

SAKKINEN, M., The Darker Side of C++ Revisited. *Structured Programming*, 13: 155-177.

SCHERLIS, W. L., Abstract Data Types, Specializations, and Program Reuse.

SHAW, M., WULF, W.A., Toward Relaxing Assumptions.in Languages and their Implementations. *SIGPLAN* 15(3), 45-61 1980

VOLPANO, D. M., KIEBURTZ, R. B., The Templates Approach to Software Use. Software Reusability, Edited by Biggerstaff, T.J., Perlis, A.J. Addison-Wesley