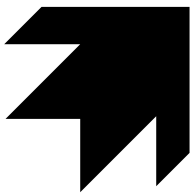
This is Chapter 11 from the book

Krzysztof Czarnecki and Ulrich Eisenecker.
*Generative Programming: Methods, Tools, and Application*s.
Addison-Wesley, Boston, Massachusetts, 2000

Chapter 11

Intentional Programming



Language exists to conceal true thought.

—Attributed to
Charles Maurice
de Talleyrand-Périgord

11.1 Why Is This Chapter Worth Reading?

This chapter presents groundbreaking concepts and technologies that have the potential to revolutionize software development as we know it today. We will discuss *Intentional Programming* (IP), which represents a perfect implementation platform for Generative Programming. IP is an extendible programming and metaprogramming environment based on *active source*, that is, a source that may provide its own editing, rendering, compiling, debugging, and versioning behavior. The benefits of the IP technology are exciting, in particular:

- It enables the achievement of natural notations, great flexibility, and excellent performance simultaneously.
- It provides optimal, domain-specific support in all programming tasks (supports effective typing, rich notations, debugging, error reporting, and so on).
- It addresses the code tangling problem by allowing you to implement and easily distribute aspect-oriented language features.
- It allows you to include more design information in the code and to raise its intentionality (or as some like to say, "to raise the level of abstraction").
- It helps with software evolution by supporting automated editing and refactoring of code.
- It makes your domain-specific libraries less vulnerable to changes on the market of general-purpose programming languages.
- It can be introduced into an organization in an evolutionary way with exciting benefits for the minimal cost of initial training.

- It supports all of your legacy code and allows you to improve it.
- It provides third-party vendors with a common infrastructure and a common internal source representation for implementing and distributing language extensions and language-based tools.
- It promotes domain specialization and facilitates more effective sharing of domain-specific knowledge.

The main ideas of Intentional Programming include

- Representing both general and domain-specific programming abstractions directly as language features (called *intentions*).
- Replacing traditional, fixed programming languages with configurations of intentions that can be loaded into the system as needed.
- Representing program source not as a passive, plain text, but as active source allowing the programmer to interact with it at programming time.
- Allowing for domain-specific extensions of any part of the programming environment including the compiler, debugger, editor (which displays active source and allows its entry), version control system, and so on. This feature enables you to provide domain-specific optimizations, domain-specific notations (including graphical ones), domain-specific error-handling and debugging support, and so on.

In this chapter, you'll learn about the philosophy and technology behind Intentional Programming.

11.2 What Is Intentional Programming?

Extension libraries

Programming versus metaprogramming environment

Intentional Programming (IP) is a new, groundbreaking extendible programming and metaprogramming environment based on active source, which is being developed at Microsoft.[1] As a programming environment, IP optimally supports application programmers in their programming tasks and allows them to load *extension libraries* to extend it with new general-purpose and domain-specific programming language extensions as needed for a given application. Extension libraries may extend any part of the environment including the compiler, debugger, editor, version control system, and so on. As a metaprogramming environment, IP optimally supports language implementers developing extension libraries. It

1. See www.research.microsoft.com/ip and [Sim95, Sim96, Sim97, ADK+98, Röd99].

provides metaprogramming capabilities including a code-transformation framework, protocols for coordinating the compilation of code using independently developed language extensions, special debugging facilities for debugging metacode, and sets of standard APIs for extending the various parts of the environment. Similar to application programmers, language implementers can also take advantage of the extensibility of IP and load extension libraries providing notations and facilities that help to do their job more efficiently.

Active source, methods, intentions

One of the main ideas of IP is to represent program source not as plain ASCII text, but as *active source*, that is, a graph data structure with behavior at programming time. The behavior of program source is implemented using *methods* operating on the source graph. The methods define different aspects of the source including its visualization, entry, browsing, compilation, debugging, and versioning behavior. User programs are written using language abstractions loaded into the system. In the simplest case, you may load a set of C programming abstractions, which will let you write C code. However, any useful combination of general-purpose and domain-specific language abstraction can be loaded and used in a program. In the IP terminology, language abstractions are referred to as *intentions*. IP lets you implement new intentions by declaring them and implementing their methods. The methods are then compiled into extension libraries (which are basically a general kind of active libraries discussed in Section 8.7.4). When you want to use certain intentions, you load the corresponding extension libraries into the development environment. Thanks to the code in the extension libraries, the IP system will know how to display source using these intentions as well as know how to support its entry, browsing, compilation, debugging, versioning, and so on.

Intentions versus object classes

Intentions are not to be confused with object classes: The methods of an intention are called at programming time, and they support all the different aspects of program development that uses this intention. This is also the added value of intentions: By implementing abstractions as intentions rather than classes, your abstractions can actively support editing, compiling, optimizing, profiling, testing, and browsing programs that use these abstractions.[2]

Rendering and optimizing domain-specific abstractions

An exciting aspect of IP is the possibility of implementing and using domain-specific abstractions as intentions. This gives you the opportunity to implement domain-specific notations, for example,

2. Please note that intentions live one level above object classes: An intention implements a language feature, that is, you can implement the class construct as an intention.

mathematical formulas can be displayed using true two-dimensional notation (e.g., variables with subscripts). You can also have graphical notations or embed GUI controls in the source (see Figure 11-14 through Figure 11-20 for IP screen shots demonstrating these capabilities). There are virtually no limitations to the kind of notation you can implement. Also, you can truly interact with an instance of an intention in a program. For example, the program source can contain an instance of a decision table, which is rendered as a table control, and the data you enter in the table is checked for consistency during entry. However, the most exciting point is that domain-specific notations may implement their own domain-specific optimizations. For example, you can have a nicely visualized mathematical formula (just like in a math textbook) and still have the system generate very efficient code for it. In fact, the code could be more efficient than what you would manually write in C or even Assembler. This is so because the system can apply complex optimizations of the application code that would not be practical to perform manually (e.g., a set of matrix intentions could implement various complex cache-based optimizations of matrix code). Because the system allows you to implement different alternative visualizations that you can switch between, you can have different views showing selected aspects of the source code in different textual and/or graphical notations. This is similar to the way CAD systems work, by letting you choose a particular view on a complex structure.

Eliminating the need for parsing

Another feature of IP is the elimination of the need for parsing. This is very useful because parsing limits the extendibility of current programming languages. For example, the grammar of C++ is context sensitive, extremely complex (contains several reduce/reduce ambiguities), and difficult to extend without making it unparsable. Furthermore, standard C++ has a number of artificial syntax rules that had to be added to the language in order to keep it parsable. An example of such a rule is the necessity to separate two closing angle brackets in template expressions by an extra space (e.g., `foo<bar<foobaz> >`) in order to distinguish them from the right-shift operator (i.e., `>>`). Another example is the strange way of distinguishing the declaration of the postfix variant of the increment operator from the prefix one (i.e., `++()` versus `++(int)`). IP does not have such problems because it eliminates the need for parsing altogether. The source graph is built as you type. When you enter a name of an intention, the system will create an instance of it and may activate some of the intention's methods to help enter further nodes. For example, if you type in the name of a procedure declared elsewhere, the system will create a procedure call

and insert a placeholder for each argument. Next, you can tab through the placeholders and replace them by the actual arguments. You can think of the editing process as working with a WYSIWYG[3] word processor rather than with a simple text editor. That is, while editing, you are modifying a more complex underlying representation rather than just simple text. Furthermore, each text element may exhibit its own behavior. However, it is important to point out that the IP editor is not a syntax-oriented editor, and this is good. You are in full control over what kind of source graph you are building. There are commands that help you navigate in the graph and make any changes you wish. In other words, you can enter programs that would not compile (the structure of the program is checked when you compile it). This is very useful during coding because the requirement to adhere to some program syntax at any point in time is impractical. All in all, direct editing allows IP to support extendible and feature-rich domain-specific notations.

Intentional encoding

Thanks to the possibility of introducing new language abstractions, you can capture more analysis and design information in the source code. This is so because you can introduce new intentions to represent the analysis and design abstractions for your software and use them right in the source. In many cases, the information that would otherwise make a great comment can actually be represented in a machine-processable form. You can achieve this in IP by introducing new abstractions to represent this information. Furthermore, programmers can introduce intentions to capture common patterns and idioms they would otherwise have to enter manually over and over again. Most importantly, such intentions would not be just simple textual macros, but proper language features. Thus, the main goal of IP is to help to represent source as *intentionally* as possible, that is, without any loss of information or obscure language idioms and clutter.

Refactoring and software evolution

The main reason refactoring is so difficult to do today is that most of the design information is missing from the conventional implementation source. IP changes this situation. Because the source is represented as an extensible abstract syntax tree, and you can provide higher-level representations of your program, it becomes possible to automate refactoring and evolution to a high degree. You don't have to parse textual source and recognize code patterns and higher-level design structures in it before refactoring (which is not possible to do automatically in implementations

3. What You See Is What You Get

using conventional general-purpose programming languages) because they are present directly in the source.

Investment protection

A problem faced by library vendors today is that general-purpose languages come and go, and the vendors are often forced to reimplement their domain-specific libraries in a new language. This is different with IP because your domain knowledge is represented using domain-specific abstractions and structures. Thus, the changes you apply to the source are dictated by the development in the domain and by the need to improve your source to better reflect the domain rather than by changes on the market of general-purpose languages. As a result, your investment into domain-specific code is better protected.

Sustainable and iterative improvement

Given the possibility of concentrating on domain-specific representations and the support for automated refactoring, you can actually constantly improve and evolve your software. As you "grow" your domain-specific libraries, you can improve the domain-specific notations, add more and more sophisticated domain-specific optimizations based on the newest research, provide interoperability with other domain-specific libraries that are likely to be used with yours, and so on. Unlike the integration of runtime components, such as JavaBeans, COM, or CORBA components, there are no runtime performance penalties for the integration of extension libraries. This is so because extension libraries interact at compile time to generate efficient code. All in all, IP allows you to improve the sources of your libraries in an iterative and sustainable way, so that you can become and then grow as a champion of your domains.

Metaprogramming

One way of thinking about IP is to view it as a perfect environment for metaprogramming. This is particularly true if you compare it to template metaprogramming in C++ (see Table 14-18 in Section 14.4.3). Template metaprogramming gives you only a limited set of abstractions to express metaprograms. In IP, on the other hand, you can use the same facilities at the metalevel as those available at the base level. That is, you can use the same intentions to write application code and extension libraries. However, if you wish to use special declarative notations to specify language extensions, you may provide them as extension libraries. A further shortcoming of C++ is that you cannot provide your own library-specific compile-time warnings. This is different in IP because extension libraries may attach user-defined error warnings to erroneous locations in the source they help to compile. Another problem with template metaprogramming is that you cannot debug template metaprograms because you cannot debug the compilation process. The IP environment, on the other hand, provides a debugger that you can use to debug the metacode. In particular, it lets you debug code transformations so you can view the different

intermediate results of code transformation and also step through the execution of the code at the source level or the different intermediate transformation levels. Additionally, IP lets you debug the execution of the metacode itself. Finally, template metaprograms only extend the compilation process, whereas extension libraries may also contain metacode that implements other aspects of intentions including their visualization, entry, debugging, and so on.

Aspect-oriented programming

You can also think of IP as a perfect implementation platform for aspects (see Chapter 8). You can implement aspect languages and aspectual composition mechanisms as extension libraries. In addition to compile-time weaving, you can also implement different aspectual views as different kinds of visualization of a single source.

Working with legacy code

Finally, it is important to note that existing code written in any language can be imported into the IP system. In order to import legacy code in a given language, you only need to provide a parser and a set of intentions for that language (import parsers for C, C++, and Java are already available in the IP system). Once imported into the IP system, you can start improving the legacy source by introducing higher-level abstractions that were not present in the original language (e.g., introducing templates or preconditions and postconditions, intentions representing common idioms in the source, and domain-specific abstractions) and restructuring the code. All the refactoring capabilities of IP can be used in this process. So, the introduction of IP can be an evolutionary process instead of a revolutionary one. You can keep your legacy code and actually reengineer and improve it using IP. Thus, the introduction of more radical domain-specific notations does not have to be a revolutionary change. IP allows you to perfectly mimic the traditional text-based way of typing in source code (you'll see this in Section 11.4.1).

Short history

IP is the brainchild of Charles Simonyi,[4] who has been leading its development since the early nineties. He refers to IP as an "OS for abstractions" [WTH+99], meaning that IP provides a set of basic APIs and an infrastructure for developing language abstractions. The initial version of IP was implemented in C and after implementing the C intentions, the system was bootstrapped, that is, all IP sources were imported into IP.[5] This way, the system "lib-

4. Charles Simonyi hired and lead the teams that originally developed the Microsoft Word and Excel products.

5. C++ would have certainly been a more appropriate choice for implementing such a complex system. According to the IP development team, completing the first bootstrap had the highest priority, however. This was the reason IP was originally implemented in C. Implementing C intentions was by far a simpler job than implementing C++ intentions. The C++ intentions were implemented later.

erated" itself from C, and since then, new intentions have been added and also used in its implementation. Since the first bootstrap in 1995 [Sim95], all IP development has been done in IP itself. This provides the IP development team with an excellent opportunity to refine and improve the system as they use it.

11.3 Technology Behind IP[6]

11.3.1 System Architecture

The IP system is an extendible programming and metaprogramming environment containing all the usual components of a typical integrated development environment (IDE) including an editor and browsing tools, a compilation component (*reduction engine*), a debugger, a version control system, and parsers for importing legacy source. All these components operate on the IP source graph. The architecture of the IP IDE is shown in Figure 11-1.

Compared to the components of a conventional IDE, the components of the IP IDE have special capabilities including

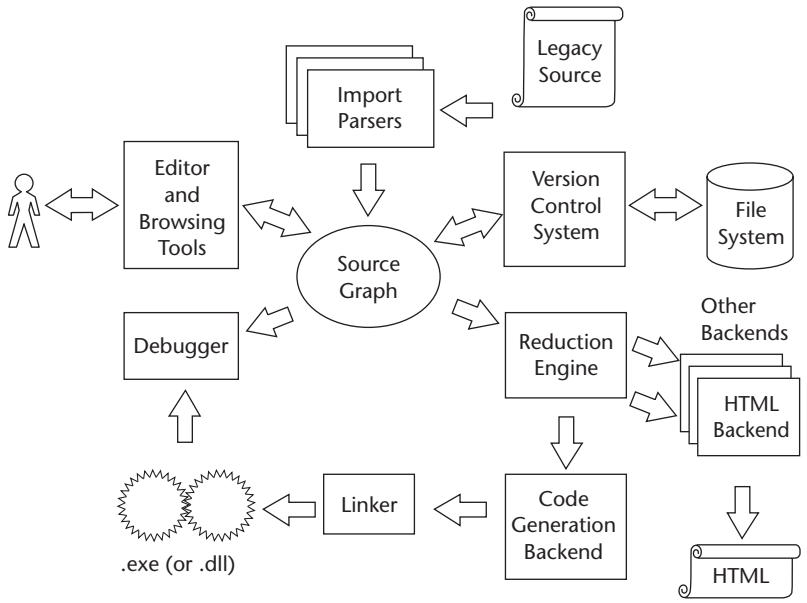


Figure 11-1 *Main components of the IP system*

6. As of this writing, the IP system is still under development, and the rest of this chapter describes the status of the system at the end of 1999.

- *Editor and browsing tools:* As in any IDE, the editor is used for viewing and entering program source. However, the IP editor is not simply a text editor, but rather a graph data structure because the program source in IP is not text (more precisely: syntax tree with links to declarations). This data structure is built up immediately as the source is being typed in (i.e., we can say that the editor replaces the parser of a traditional IDE). Furthermore, the IP editor supports true two-dimensional textual and graphical source views and allows the programmer to interact with the elements of the program source at programming time. In particular, mouse clicks and keystrokes entered on a given program element may be handled by this element (more precisely: by its methods). The IDE also provides a set of browsing tools and facilities for navigating in the program source. Because the program source has the form of a syntax tree with links to declarations, full and up-to-date browsing information is available all the time. We will discuss the editor and browsing tools in Section 11.4.
- *Reduction engine, code generation backends, and linker:* The IP reduction engine is the code-transformation framework of IP, and it plays the role of a compiler. There are several differences between the reduction engine and a compiler, however. First of all, there is no parser because the source is already available as a syntax tree with links to declarations. Furthermore, the reduction engine implements a transformational approach to compilation. After checking the structural correctness of the source, the reduction engine applies a series of transformations in order to generate an implementation consisting of a limited set of primitive abstractions called the *reduced code* or *R-code*. The ordering of the application of transformations is done partly by the reduction engine and partly by the transformation writers. Different R-codes can be defined for different target platforms, and the source can be reduced to different R-codes. Finally, an appropriate backend generates the platform-specific object code from a given R-code. For example, the backend from the Visual Studio product family is used to generate machine code for the Intel processors, and a different backend is used to generate Java bytecodes. The resulting object files are linked by a standard linker into an executable or a library. The reduction engine, together with some appropriate backend, can also be used to generate artifacts other than executable programs. For example, we can use this capability to generate documentation from the program source or implement higher-level language for generating Web pages.
- *Debugger:* The debugger allows the programmer to step through the execution of the program at different levels: the source level

or any selected intermediate level produced by the transformations. You can also use the debugger to debug the IP system itself and the extension libraries (i.e., the metacode you write).

- *Version control system:* IP provides a team-enabled version control system, which works on binary IP source files. In contrast to conventional text-based version control systems, the IP version control system allows you to automatically merge two versions of a source, where one contains renamed and relocated functions and the other incorporates modifications in the bodies of these functions. After merging, the modifications will be incorporated in the renamed and relocated functions.
- *Import parsers:* Language parsers, for example, for C++ or Java, are needed for importing legacy code into the IP system. A given legacy code needs to be imported only once. The imported code is then saved in the IP program source format.

In contrast to traditional IDEs, any part of the IP system is extensible. The editor, reduction engine, debugger, and version control system may call system-provided default methods or user-provided methods (which may override the default ones) to handle new language abstractions (i.e., intentions). Methods defining the programming-time behavior of intentions are compiled into extension libraries, which can then be dynamically loaded into the IP system. For example, you can load the extension library for the C++ language, which will enable you to write C++ code. You can also load other sets of general and domain-specific intentions (which can extend the editor to handle a new kind of textual or graphical notation) and use them all at once.

11.3.2 Representing Programs in IP: The Source Graph

Source trees and source graphs

One of the main ideas behind IP is to represent source code directly as abstract syntax trees (ASTs) and let the user enter, modify, and compile them without ever having to work directly on program code stored as plain ASCII text. In IP, each AST node has a link to its declaration (e.g., a variable has a link to its declaration), for this reason IP actually represents programs as *source graphs*, which can be thought of as sets of interlinked *source trees*.[7] In the following four

7. In compiler design, ASTs whose elements have links to their declarations are referred to as "resolved ASTs." Conventional compilers usually first build unresolved ASTs using a parser, and then turn them into resolved ASTs by adding links to declarations during the semantic analysis stage. Formally, resolved ASTs are graphs, but not trees.

sections, we'll take a closer look at the structure of source graphs. This knowledge will help us understand how IP works internally. However, please keep in mind that as an application programmer using IP, you will not be permanently confronted with these details.

11.3.2.1 Treelike Structures

Tree elements and operands

Suppose that you want to represent the expression x+x as a source tree, that is, as an AST. To do so, you need a parent node representing the occurrence of the operator + and two child nodes representing the two occurrences of the variable x (see Figure 11-2). In IP, the nodes of a source tree are also referred to as *tree elements* (or *TE*s), and the child nodes are often called *operands*.

11.3.2.2 Graphlike Structures

Declarations and instances

In IP, anything you use has to be declared and defined somewhere. So, although the source tree in Figure 11-2 contains nodes representing just the *occurrences* of the operator + and variable x, the actual operator + and variable x have to be declared somewhere. Furthermore, every tree node in IP maintains a link to its *declaration*. This is illustrated in Figure 11-3. The figure shows our familiar source tree from Figure 11-2 plus the links to the declarations. The parent has a link to a node representing the declaration of the operator +, and the children have links to another node representing the declaration of the variable x. The links to declarations give meaning to the three tree elements on the left: Now, we know that the parent node is an occurrence, that is, an *instance* of the operator +, and the children are two instances of the variable x. It is interesting to note that thanks to the links to declarations we do not have to store any names in the tree elements on the left. If we

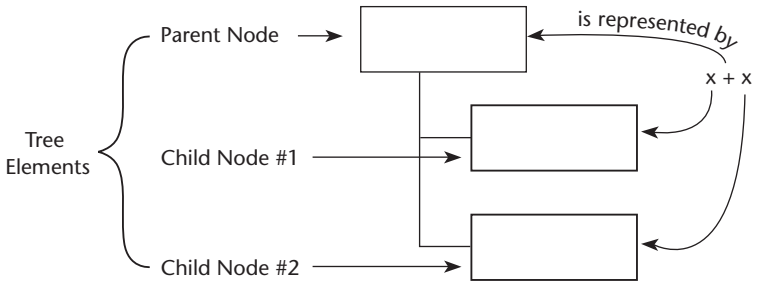


Figure 11-2 *Source tree representing the expression* x+x. *Child nodes are always drawn below their parent node and are counted in the top-down direction.*

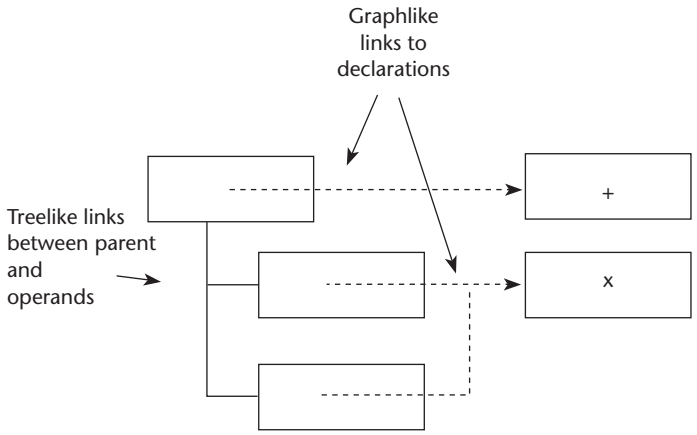


Figure 11-3 *Source tree of* `x+x` *plus declarations. Graphlike links to declarations are drawn as dashed arrows. Nodes with a name label are declaration nodes.*

need the name of any of the tree elements (e.g., the variable name "x"), we can retrieve it from the corresponding declaration. This representation has an enormous advantage over textual code: If you want to change the name of an abstraction (e.g., the variable name), you only need to change it in the declaration and do not need to update the places *referring* to the declaration. You'll see how this works in practice in Section 11.4.1.

Treelike and graphlike links

Even though the links between parent and child nodes always span a tree, this is not the case with the links to declarations. The

NOTE

It is important to note that names are only needed for the communication with the human programmer. The system does not need names because it references nodes using links, which are either pointers (for declarations and operands that are in the same address space) or globally unique identifiers (for declarations that are in a different address space). Furthermore, IP gives us the opportunity to use names that are not stored, but computed on-the-fly based on a given source pattern. For example, the generated name for a function parameter of type `char*` could be `pchar`. This is not always satisfactory, but if it is, it may save the programmer some tedious work. Finally, IP also allows you to give more than one name to an intention. For example, you may use short names for easy typing and then display source with long names for easy maintenance.

latter may point to declaration nodes located in the same tree or a different tree, and several nodes may have links to the same declaration (e.g., several occurrences of `x` point to the same declaration `x`). That's why we refer to the links between parent and child nodes as *treelike links* and the links to declarations as *graphlike links* (see Figure 11-3).

DCL—the declaration of all declarations

As stated earlier, every node is an instance of some abstraction, and every abstraction is declared somewhere. Therefore, there is also a declaration declaring the very abstraction of "declaring," and every declaration node has a graphlike link to it (see Figure 11-4). This special declaration is called `DCL`. Because `DCL` is also a declaration, it has a graphlike link to itself. Furthermore, declaration nodes may also have child nodes—just as any other node. Figure 11-4 illustrates this for the declaration of `x`. The first child node of this declaration node represents the variable type, which is `int`. The second child is optional and represents the initializer—in our example the value `1`. This node has a graphlike link to the declaration `constant` (i.e., it is an instance of the abstraction `constant`). The actual value is stored as binary data attached to the node, which is indicated as `"1"`. It is interesting to note that declaration names are also stored by attaching their binary representation to—in this case—declaration nodes.

Declarations versus references (to declarations)

Graphlike links always point to *declarations*, which are the nodes with graphlike links to `DCL`. Examples of declarations in Figure 11-4 are `+`, `x`, `int`, `constant`, and `DCL`. Nodes that are not declarations (i.e., nodes without a graph-like link to `DCL`) are called *references to* declarations to which they have graphlike links. For example, the first child of the declaration node `x` is a "reference to

NOTE

In IP, treelike links are bidirectional, that is, you can navigate through these links in both directions. Graphlike links, on the other hand, are unidirectional and point towards a declaration node of an intention; that is, an instance knows its declaration, but a declaration does not know its instances. Furthermore, the point where a link "leaves" a node can be identified through a name tag (i.e., bidirectional links have tags at both ends, and unidirectional links have a tag at their origin). This is similar to annotating associations with role names in UML class diagrams. For example, the link connecting an instance of `+` with its left operand `x` could have the tag `leftOp` at the `+` end and `parent` at the `x` end. IP provides an API for accessing nodes and links based on name tags.

NOTE

It is interesting to note that IP does not introduce an explicit "variable intention" (i.e., an abstraction of variables). The declaration of a variable (e.g., `x`) represents its own variable intention.

`int`." According to the terminology introduced at the beginning of this section, you can also call it an "instance of `int`."

11.3.2.3 Source Graphs: The Big Picture

Intentions

Any programming abstraction you want to use in your program has to be declared first. You can then use it by referring to its declaration. For example, if you need to use a variable, you have to declare it first. As seen earlier, this is also true for language abstractions, that is, *intentions*, such as `+`, `int`, `constant`, `if`, `while`, and so on. Of course, to be really useful, the abstractions not only need to be declared, but also defined, which can be accomplished by associating appropriate methods with their declarations. These methods define their semantics, appearance, debugging behavior, and so on. We will discuss methods in Section 11.3.3. Obviously, as a programmer, you don't have to declare and implement basic intentions, such as `if` or `while` yourself—they will come with libraries distributed with the IP system. Furthermore, you will use third-party libraries for domain-specific intentions or general-purpose intentions with some interesting behavior. And you will also be able to implement your own intentions for your area of specialty.

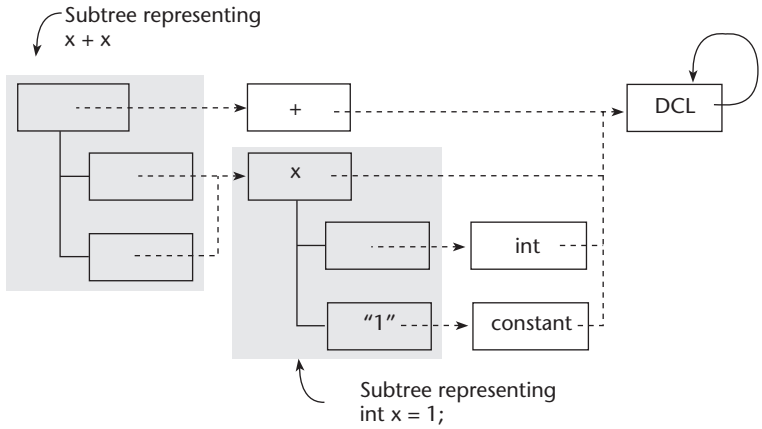


Figure 11-4 *Source trees representing* `int x=1;` *and* `x+x`

To get the big picture of how IP represents programs, let us take a look at a larger example of a source graph (see Figure 11-5). This graph contains the source tree of the following piece of C code:

```
int x;
x = 1;
while (x<5)
   ++x;
```

The source tree of this program is shown in Figure 11-5 on the left. It consists of a list of three statements. Each subtree representing one of the statements is enclosed in a gray box. Please note that the two child nodes of the third statement (i.e., the `while`-statement) represent the condition expression and the iteration statement, respectively. The nodes on the right-hand side represent the declarations of the intentions used in the user program. These intentions are located in extension libraries loaded into the IP

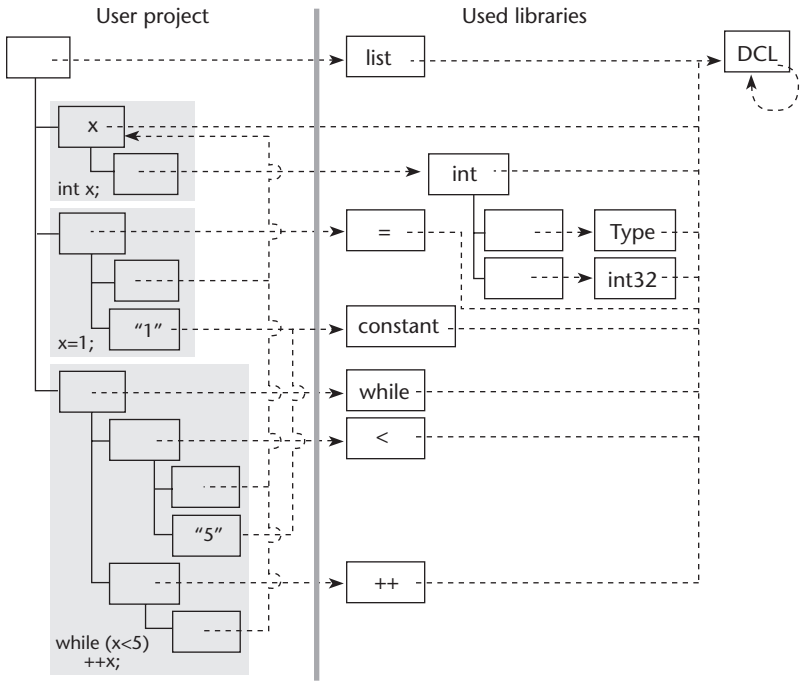


Figure 11-5 *Larger source graph example*

system. For simplicity, not all children of the declaration nodes on the right-hand side are shown in this figure.

11.3.2.4 The Essence of Source Graphs: Abstraction Sharing and Parameterization

Treelike links represent parameterization

As stated previously, source graphs contain two kinds of links: graphlike and treelike links. Graphlike links represent the concept of referring to a declaration or being an instance of the declared *abstraction*. Treelike links represent the concept of *parameterization*: An instance of an abstraction can be specialized and concretized for a given context by attaching child nodes to it. That is, the child nodes of an instance can be seen as its actual parameters.

Graphlike links represent sharing

The idea of abstractions is central to source graphs. As previously stated, abstractions have to be declared, and then you can use them by referring to them. For example, think of a source tree containing two identical sequences of statements in two different places. Obviously, you would like to eliminate this duplication to have only one, shared sequence of these statements. We can achieve this by introducing an abstraction: We declare a procedure whose body is the statement sequence and replace the original sequences in the original tree by calls to (i.e., instances of or references to) this new procedure. Now, think of this abstraction process in general terms: Any duplicate subtrees can be eliminated by introducing an abstraction, and then referring to it. In other words, the purpose of graphlike links is to enable *sharing* of structures in a source graph.

Of course, the subtrees that we want to abstract need not be identical. If they are not identical, they have some differences, and we can turn these differences into parameters of the abstraction. When we are using an abstraction, we have to be able to supply actual parameters to a particular instance of use. That's what treelike links are for. For example, an instance of a while-intention takes two parameters: the continuation condition and the iteration statement. A procedure call (i.e., an instance of a procedure) takes actual parameters. A template instantiation takes actual template parameters.

11.3.3 Source Graph + Methods = Active Source

Rendering, type-in, and reduction

One of the main ideas behind IP is to represent programs as *active source*, that is, source with behavior at programming time. Active source knows how to display and compile itself and provides convenient and domain-specific ways of editing and debugging it. In IP, this is achieved by defining methods (i.e., pieces of code) operat-

ing on the source graph. Different sets of methods implement different aspects of source behavior: There are methods for implementing the visualization of the source graph (i.e., *rendering*), supporting its entry (i.e., *type-in*), implementing its compilation (i.e., *reduction*), debugging, and automatic editing and refactoring.

Abstractly, the idea of methods in IP is similar to the idea of methods in object-oriented languages. Just as the methods attached to a class define the behavior of the class's instances, the methods associated with a declaration of an abstraction in an IP source graph define the behavior of the abstraction's instances.

There is an important difference, however: Methods in object-oriented languages are designed to be executed on class instances at runtime, whereas IP methods are specially designed to operate on source graphs at programming time. Consequently, the method calling, lookup, and inheritance mechanisms in IP are radically different from those found in object-oriented languages. In particular, IP methods are associated with patterns of nodes rather than just single nodes and are specially designed to support efficient source-graph traversals. We will discuss the IP method mechanism in Sections 11.5.1 to 11.5.4.

Default methods

The IP system is organized like an object-oriented framework: The system calls sets of system-defined *default methods* and user-defined methods. The idea is that the default behavior is defined by default methods, but language implementers can define specialized behavior for new intentions by overriding the inherited methods.

Please note that methods are written by developers of extension libraries only, so that application programmers using IP do not have to worry about them. However, just for the purpose of understanding how IP works, let us take a look at the different kinds of methods we can have in IP.

11.3.3.1 Kinds of Methods

We classify extension methods according to their purpose. The main categories are as follows [Sha98].

- *Rendering methods:* Rendering methods display the source graph on the screen. They use the rendering API to produce display representations of the source graph. This can be simple text or any kind of two-dimensional representation, for example, two-dimensional mathematical formulas, diagrams, tables, bitmaps (see Figure 11-14 through Figure 11-23), and so on. Furthermore, you can define several sets of rendering methods, each one implementing one specific visualization of the source graph (including domain-specific modeling notations, different

formatting conventions, call graphs, metrics, rendering names in different natural languages, and so on).

- *Type-in methods:* Type-in methods are called when the source tree is entered or manipulated. They assist the programmer in typing in source. For example, you can define a method that inserts the appropriate number of place holders when you type in a reference to an intention (e.g., after typing in the name of a procedure, the appropriate number of place holders for the arguments are inserted) or when you want to do any other kind of special editing (e.g., typing in a type will replace the type by a declaration of a variable of this type). There are methods that define how to select the elements shown on the screen, the tabbing order, and so on.

Reduced code (i.e., R-code)

- *Reduction methods:* The process of transforming source trees into lower-level trees is referred to as *reduction*. The final result of this process is a tree containing only the instances of a predefined set of low-level abstractions for which machine code can be directly generated (in the phase called *code generation*). This representation is referred to as *reduced code* or *R-code*. When you tell the system to compile your program, the system will ask the root node of your program for its R-code, which involves activating the reduction method for this node (which in turn activates the reduction methods of its subnodes, and so on). Reduction methods compute the R-code by successively reimplementing higher-level constructs by lower-level ones. An example of a reduction step could be implementing a while-loop using an if- and a goto-statement, for example:

```
while (x<5)          TEST: if (x<5)
    ++x;      ──────►        {  ++x;
                             goto TEST;
                         }
```

As you will see in Section 11.5.5, the lower-level representations are actually attached to the original source tree rather than replacing the parts being reduced. Indeed, a reduction method is not allowed to delete any links in a source graph, but only to add new ones. In other words, during reduction, the source graph grows monotonically as more and more lower-level abstractions are attached to the tree. This way, it is easy to make sure that the reduction process is actually progressing and will terminate at some point. Of course, just as in any compilation process, reduction methods also perform structural analysis (e.g., syntax and type checking) and code optimizations. For this purpose, a method

operating on a node can ask for information not only about the near context of this node, but also remote parts of the source graph. Finally, there may be several sets of R-code intentions—each set defined for a different target platform, for example, the Intel 86 family of processors or Java bytecodes. In many cases, you can implement methods for the high-level intentions such that they can be reduced towards different target platforms. However, there can be intentions that cannot be reduced towards some specific platform, that is, they cannot be implemented using the platform-specific set of R-code intentions. For example, pointer arithmetic cannot be "reduced" to Java bytecodes in an efficient way.

- *Debugging methods:* The standard functionality of a debugger is to allow you to watch the execution of some executable at its source level and inspect the runtime values of its variables. Implementing this functionality requires being able to identify the place in the source that corresponds to the lower-level construct being currently executed. This is not a problem as long as the mapping between the corresponding locations in the source and the executable is a linear one (i.e., continuous blocks of source map to continuous blocks of executable code). (We have discussed the concepts and problems of linear and nonlinear code mapping in Section 8.7.2.) Debugging code with a linear mapping is handled by the debugger automatically. Unfortunately, if we want to support code optimizations and/or aspect-oriented language features, this mapping will be nonlinear. For example, code optimizations may eliminate parts of the source or change the ordering of instructions. Furthermore, code weaving needed for aspect-oriented language features will merge different code pieces into a single one. And because we want to be able to write reduction methods implementing domain-specific optimizations and code weaving in IP, we need to deal with the problem of nonlinear code mapping. The solution to this problem is to provide *debugging methods*, which can compute the desired mapping from lower-level implementation back to the higher-level source for the nonlinear case. For example, a reduction method can optimize away certain variables present in the source. If we want to step through the execution of the optimized code at the source level, we need to provide a debugging method that recreates the values of the variables in the source from the values of some other relevant variables at the execution level. Only in this way will we be able to inspect the source-level variables during debugging. So the idea is that, if an intention provides a reduction method with nonlinear code transformations, it also needs to provide an appropriate debugging method that

allows the debugger to do the mapping back. Furthermore, aspect-oriented and domain-specific language features can require some special debugging support, for example, highlighting timing constraints when they are violated. Such extra, domain-specific debugging features can also be provided as debugging methods. Finally, because IP is both a programming and metaprogramming environment, it supports the debugging not only of application code, but also metacode (i.e., extension libraries). For example, in order to debug reduction methods, you can make calls to a special function that takes a snapshot of the intermediate state of the tree being reduced by a given reduction method. This way, you can later inspect the different intermediate representations produced during the reduction process. The debugger will also let you debug the execution of the executable produced by the reduction methods at any of these intermediate levels. Therefore, you may also want to equip your new intentions with debugging methods that support not only debugging at the source level, but also at the intermediate levels.

- *Editing and refactoring methods:* Because program source in IP is represented as an AST, it is quite easy to write methods for mechanical source editing and restructuring. You can have simple editing methods, such as applying De Morgan's laws to logical expressions or turning a number of selected instructions into a procedure and replacing them by a call to this procedure (in IP this is done with the *lift* command). Other methods may perform complex design-level restructuring of legacy code (i.e., refactorings), for example, extracting the interface between different parts of the source and modifying the module structure of a software, replacing inheritance through aggregation, finding and eliminating duplicate or similar code, and so on.
- *Version control methods:* Version control methods allow us to define specialized protocols for resolving conflicts when two or more developers edit the same piece of code. In general, versioning is subject to intentions-specific handling because you can treat language abstractions (e.g., modules, procedures, classes, and methods) as units to be locked and versioned separately.

There are also other methods that do not fit in any of these categories.

11.4 Working with the IP Programming Environment

IP document

The IP programming environment supports all the usual programming activities: writing and versioning code, compiling, and debugging. Figure 11-6 shows a screenshot of a typical program-

ming session. The editor subwindow contains a simple "Hello World" program using C abstractions. The program can be stored on a disk in a single binary IP source file referred to as a *document*. In general, a program can consist of more than one document. The smaller subwindow located to the right of the document editor subwindow is the *declarations list tool*. This tool allows the developer to do a name-based search for a declaration among the currently loaded declarations. A click on one of the displayed names opens the document containing the corresponding declaration in the current editor window. There are other browsing tools, such as the *references list tool*, which shows all the references to a certain declaration; *libraries list tool*, which enumerates all the currently opened libraries; *to-do list tool*, which displays a list of to-do annotations in the current document; and so on. There is also a tree inspector, which graphically shows the exact tree structure of the selected code. Finally, you can jump to the declaration of a selected node using the go-to-declaration button located on the menu bar.

The "Hello World" program can be compiled by simply pushing the compile button on the menu bar. This initiates the reduction process, which, if successfully completed, is followed by the generation of the executable. If there are syntax or semantic errors in the source, the error notifications are attached to the appropriate

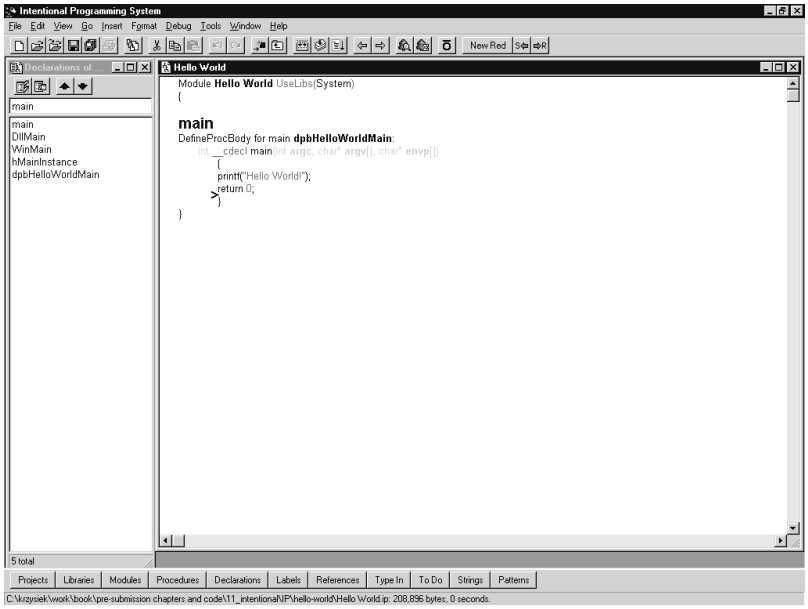


Figure 11-6 *Screenshot of a typical programming session with the IP system*

nodes, so that they appear on the screen in a different color next to the erroneous positions in the code, and you can use a "jump to next error" tool to visit these positions. After a successful compilation, it is possible to step through the intermediate results of the reduction process (these are recorded by a snapshot function, which can be called at various places in the reduction methods). Finally, you can debug the program by stepping through its execution at the source level or any of the intermediate levels.

11.4.1 Editing

Probably the most unusual experience to a beginning IP programmer is editing. This is because you edit the tree directly, which is quite different from text-based editing. To give you an idea of how tree editing works, we will walk through a simple editing example. Figure 11-7 shows you how to type in the following simple program:

```
int x = 1;
int y;
y = x + 1;
```

Each box in Figure 11-7 shows you the editing screen after typing the text shown below the preceding arrow. We start with the empty screen and type in "int". While typing it, the gray selection indicates that we have still not finished typing the token. We finish typing it by pressing <tab> or <space> (the first is preferred because it automatically positions the cursor at the next reasonable type-in position). After pressing <tab>, the system will perform a couple of actions behind the scenes leading to the creation of a variable declaration with type `int` (see box 2 in Figure 11-7). Let us take a closer look at these actions. After we've pressed <tab>, the system first tries to find a binding for the token we have just typed in. In our case, the system finds that the token "int" matches the name of the declaration of the type `int`. Next, the system calls the type-in method of `int`. This method creates a variable declaration with type `int`, that is, a declaration node with a reference to `int` as its child. To be more precise, `int` does not provide its own type-in method, but inherits one from `Type`, an intention that is the type of all types (see Figure 11-5). The name of the newly created declaration is set to "`???`". We can see the result in box 2 in Figure 11-7. Because we've pressed <tab> last, the cursor is now positioned at the next reasonable type-in position. In our case, the declaration name (i.e., "`???`") is now selected, and we can type in the name of the declaration, for example, "x". The result is shown in

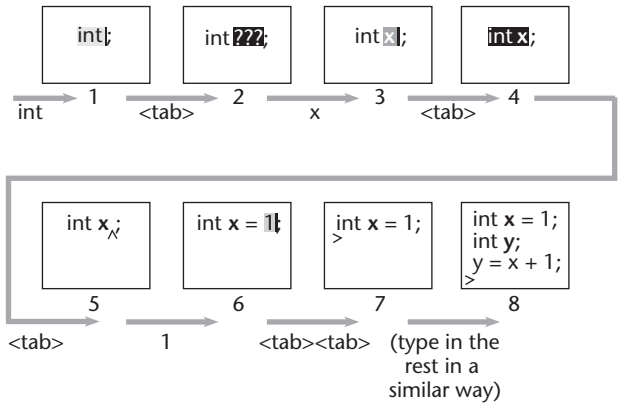


Figure 11-7 Typing in a simple program in IP

box 3. We finish typing this token by pressing <tab> and then press an extra <tab> to position the cursor for typing in the initializer (see box 5), which in our case is "1". The result of entering "1" is in box 6. The source tree corresponding to what we've typed in, that is, `int x=1;`, is shown in Figure 11-4. Please note that we did not explicitly type in the equal sign. This character is merely a display artifact displayed by the rendering method of the declaration. Next, by pressing an extra <tab>, we position the cursor behind the declaration statement in the current statement list and are ready to type in the next statement (see box 7). We type in the subsequent two statements in an analogous way. Please note that we actually have to explicitly enter an equal sign in the third statement because here it denotes an assignment.

Earlier, we said that changing a name (of a variable, function, class, and so on) in IP is very easy because we only need to change it in the declaration and do not need to search for all the places where it is used in order to replace it. Let us take a look at this in practice. For example, we might want to rename `x` in the sample code we've just typed in, let's say, to `z`. All we have to do in IP is

NOTE

IP also lets you to finish typing the name of a variable declaration by typing "=" instead of pressing <tab>, in which case you'll be expected to enter the initializer as next. In other words, IP not only supports alternative renderings, but also alternative ways of type in—to suit your most natural way of typing code.

select the name of the first declaration (box 1 in Figure 11-8) and change it to `z` (see box 3). Because all references to the declaration `x` do not store its name (but the display method retrieves it from the declaration for display), the third statement displays the correct name immediately (see box 3). This simple example illustrates the power of the IP source representation compared to text-based representations. Also, if we select `z` and then push the go-to-declaration button, the cursor will jump to the `z` declaration. So, you can always verify which abstraction is meant by a given token on the screen. In fact, we could have changed the name of the declaration `y` to `z` as well. In this case, the last statement would contain two references to `z`, that is, it would read `z=z+1`, but both references would still correctly point to the two different previous declarations (we could verify this using the jump-to-declaration button).

As a second example, let us take a look at how to enter the "Hello World" example from Figure 11-6. First you need to open a new document named "Hello World" using the "new" button from the menu bar. This will give you the window shown in Figure 11-9. The document now contains a new module named `Hello World`.

Next, we need to import the system library, which provides the declaration of the function `main()` and the declaration (and imple-

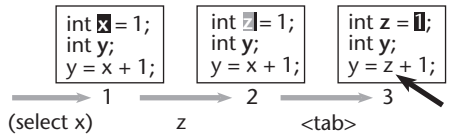


Figure 11-8 *Changing the name of a declaration*

NOTE

It is interesting to note that if there is more than one declaration with the same name in a given scope, typing in the name will actually not bind it to any of them. The name (i.e., the token we've typed) would turn yellow instead, indicating a dangling reference. We could still bind it using a list tool listing the candidate declarations and select the one we would like to bind it to. Another interesting observation is that scoping rules are implemented by the editing methods.

Figure 11-9 *New document named "Hello World"*

mentation) of the function `printf()`. This can be achieved by using a command or typing it in directly. The result is shown in Figure 11-10.

Now, we are ready to define an implementation of the `main()` function. The idea is that the function `main()` is declared in the system library, but we want to define its body here. This can be done using a so-called "define procedure body" declaration, which is an example of a special intention provided by IP. Starting with the situation in Figure 11-10, we type "DefineProcBody" between the braces and press <tab>. This creates a declaration with the type `DefineProcBody`. The result is shown in Figure 11-11. The first "`???`" should be replaced by a reference to a procedure declaration that we want to define the body for. The second "`???`" should be replaced by the name we want to give to our "define procedure body" declaration.

Next, we replace the first "`???`" by "main", which will create a reference to the declaration of the function `main()`. At this point, the rendering method for the declaration `DefineProcBody` will display the argument list of `main()` as declared in the system library. This is shown in Figure 11-12. In other words, the argument list you see in this figure is not part of the source tree you just typed in, but is merely a display artifact (which is indicated by its gray color). The idea is that the argument list is defined only in one (remote) place (which is the declaration of `main()` located somewhere in the system library), and you can still see the argument list while implementing the procedure body.

Now you can start typing the body of the procedure. For example, when you type in "printf" and press <tab>, this will be

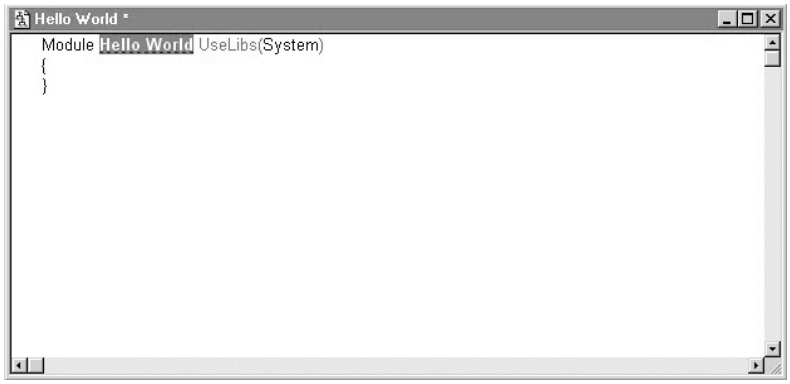


Figure 11-10 *"Hello World" module importing the system library*

expanded to a call to `printf()`. Because the declaration of `printf()` located in the standard library indicates that the function takes one argument, the system will provide a placeholder for that argument in the call (see Figure 11-13).

By replacing the argument placeholder with the string "Hello World", entering the final return statement, and replacing the remaining "???" with the name "dpbHelloWorldMain", you get the desired result shown in Figure 11-6.

11.4.2 Further Capabilities of the IP Editor

Different types of selections

Clicking on a token on the screen does not select its letters (as it would in the case of a traditional text editor), but instead selects

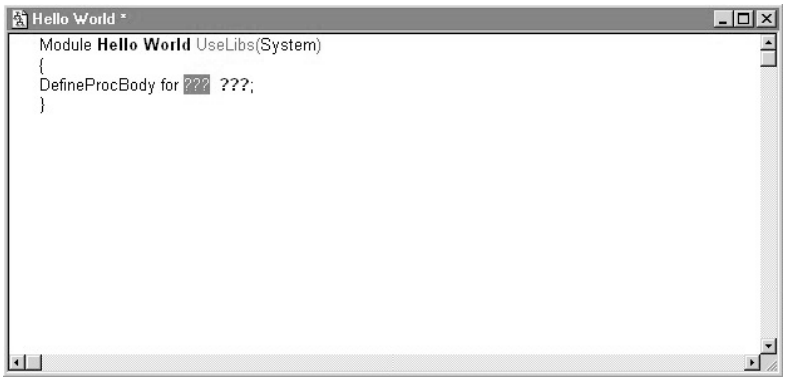


Figure 11-11 *"Hello World" module with an empty* `DefineProcBody` *declaration*

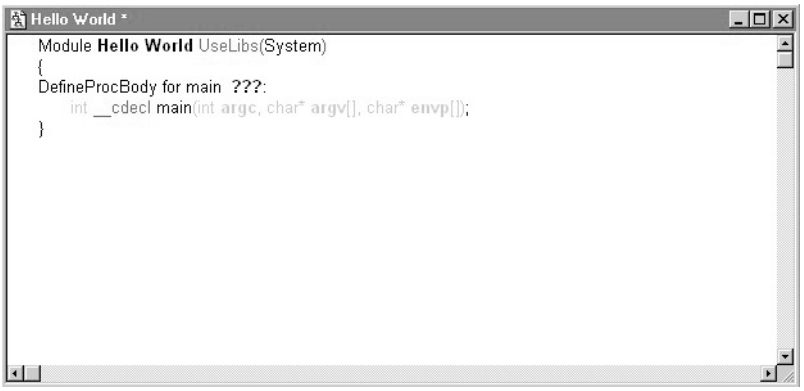


Figure 11-12 *"Hello World" module with an empty* `DefineProcBody` *declaration for* `main()`

the corresponding tree node or subtree. In fact, there are a number of different selection types. You can select one node (*crown selection*), or a node including all its subnodes (*tree selection*), or you can select a place between two nodes (*place selection*). It is also possible to select the token as such and change its name (*contents selection*). If the token is a reference, then the reference will be rebound based on the new name. If the token is a name of a declaration, the name will simply be changed (as in Figure 11-8). There are also other types of selections (which will not be discussed). You can achieve the desired selection type by holding down an appropriate modifier key while clicking on a given token.

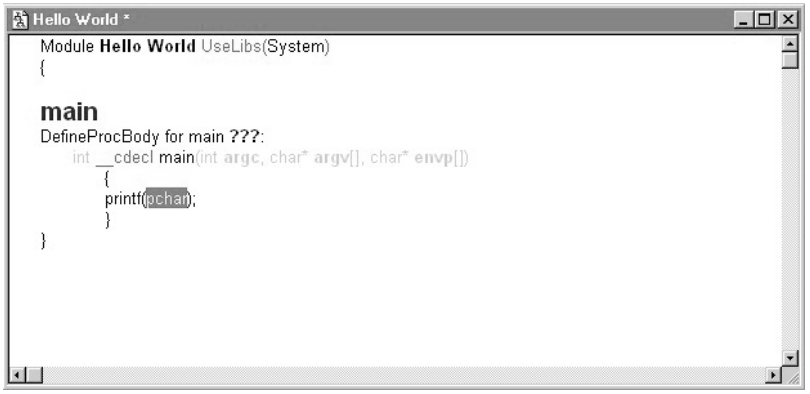


Figure 11-13 *"Hello World" module with a* `DefineProcBody` *declaration for* `main()` *with a body containing an incomplete call to* `printf(0)`

As already stated, the IP editor is not a syntax-oriented editor, that is, it is perfectly acceptable for the edited tree to be in a state that would not compile. For example, if you type a name that cannot be bound to any declaration (because there is not one with this name), the token will turn yellow, and you'll know that you need to fix this before attempting to compile the source. Also, as you type, the structure of the tree can be syntactically incorrect. On the other hand, through type-in methods, intentions may provide the programmer with type-in templates and suggest what to type in next (as we saw in the previous section). In effect, syntax errors are quite rare, and you still have the freedom to type in the source as you wish without being forced into the straightjacket of some syntax at every moment.

The major advantage of editing active source is that we are not dealing with passive text, but instead, the source has an interactive behavior. The intentions can even exhibit different type-in behaviors based on their tree context or the currently active view. Furthermore, they can interact with the developer through menus, dialogs, and so on.

Achieving clean encoding and high performance at the same time

The IP source rendering provides unique opportunities. First, intentions can be rendered in a true two-dimensional way allowing you to provide pretty mathematical notation. For example, Figure 11-14 shows an implementation of the Bessel function using C and some special mathematical intentions. This example is attractive in two ways. First, it is expressed in an easy to understand and pretty mathematical notation, which is close to what you find in math books. Second, there is a somewhat more complicated but reusable optimization transformation attached to the helping function `t()`. The display of this transformation is suppressed in Figure 11-14. The transformation transforms the costly, recursive function `t()` into a linear-cost function (basically by remembering previously computed values in a temporary). Thanks to it, the Bessel function compiles into a very efficient implementation—one that we otherwise could only get by writing hand-optimized, but messy code. But thanks to IP rendering and the possibility of contributing domain-specific optimizations, we get an easy to understand and nice looking formulation of the Bessel function plus one messy, but reusable optimization transformation instead of one efficient, but messy Bessel function.

Avoiding the parsing problem

Figure 11-15 demonstrates a few mathematical notations for handling matrices. An interesting point about these naturally looking notations is that they would be extremely difficult to parse if we wanted to provide them using traditional input technology utilizing text and parsing. In general, mathematical notations found

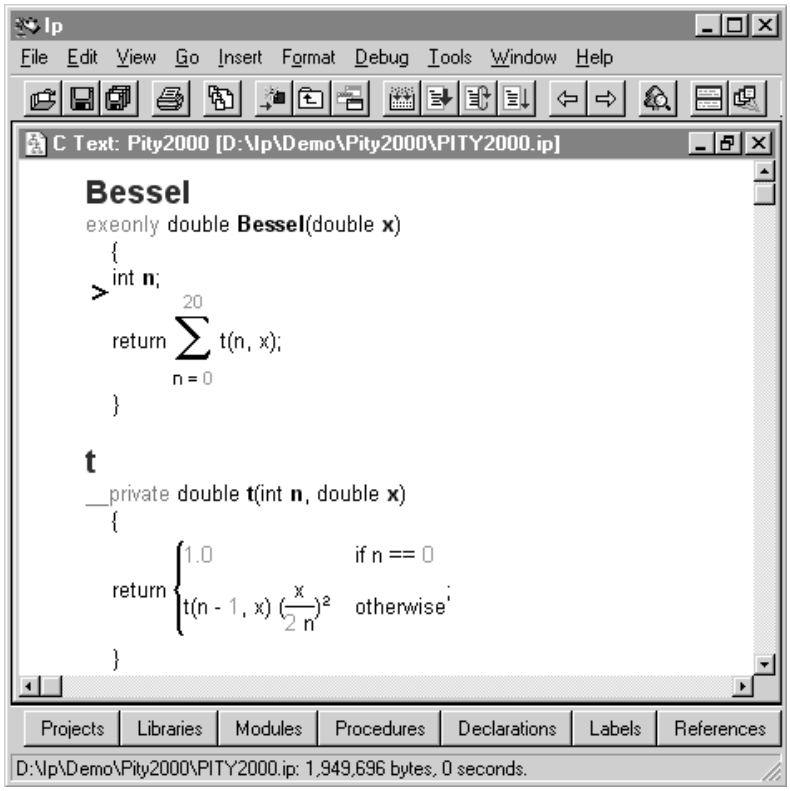


Figure 11-14 *Example of a domain-specific notation for mathematical formulas (from [Sha98])*

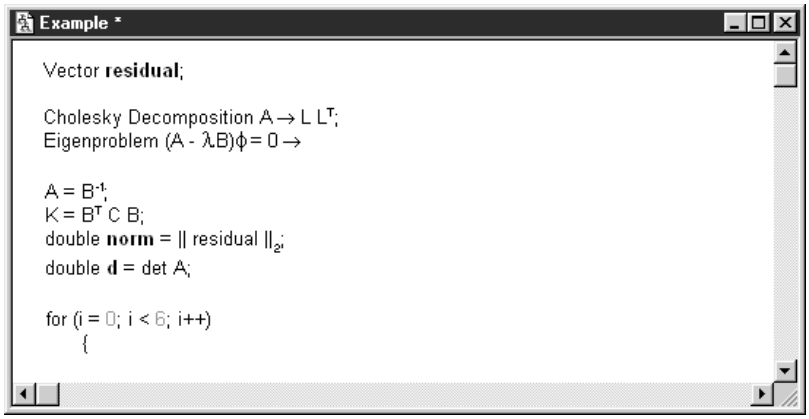


Figure 11-15 *Example of specialized notations for handling matrices*[8]

8. Courtesy of Lutz Röder.

in math books are too ambiguous to be parsable. This is not a problem in IP because the rendering of some code can be ambiguous, but the underlying source is not (the rendering shows less information than what's in the source). And you can use commands to enter unambiguous source that is rendered ambiguously.

The IP source rendering also allows you to embed graphics and graphical notations in your programs. A view containing graphics can also be provided as an alternative view to a textual view. For example, Figure 11-16 and Figure 11-17 show two alternative renderings of the same C function implementing some logical formula. One rendering uses the usual C notation, and the other one uses a graphical notation based on a circuit with logical gates. As Figure 11-18 shows, textual and graphical notations can also be easily mixed. You can readily imagine that you can use these capabilities to provide general-purpose and domain-specific graphical modeling notations.

Another interesting capability of the IP editor is shown in Figure 11-19, where a bitmap (in this example, one representing an icon) is passed as an argument and its actual contents is rendered in the source (see the little bitmap passed to `__BitmapName()` in Figure 11-19). This is much more expressive than passing a literal

NOTE

You may wonder why a syntactically ambiguous representation may still be useful. This is so because syntactic ambiguities can often be disambiguated based on additional semantic information. This idea is frequently used in math books to make math formulas easier to understand. In all but the simplest cases, such ambiguities cannot be resolved in the parser because parsers do not have access to higher-level semantic information. In IP, however, the disambiguation is done by the programmer while typing.

As Eric van Wyk pointed out, there are two issues resulting from the separation of display and type-in from the actual source structure that IP novices may find somewhat confusing. First, most programmers are accustomed to the idea that the code they see is actually the source with all the details it contains, but as stated previously this is often not the case in IP. The IP programmer may have to navigate in the code or use the detailed source tree view described at the end of this section to discover the actual structure of the source. Second, just viewing some code in IP will not tell you how to type it in. This problem may be easily solved in the future by having intentions provide "show me your type-in" commands, which would explain to the programmer how to use a given intention.

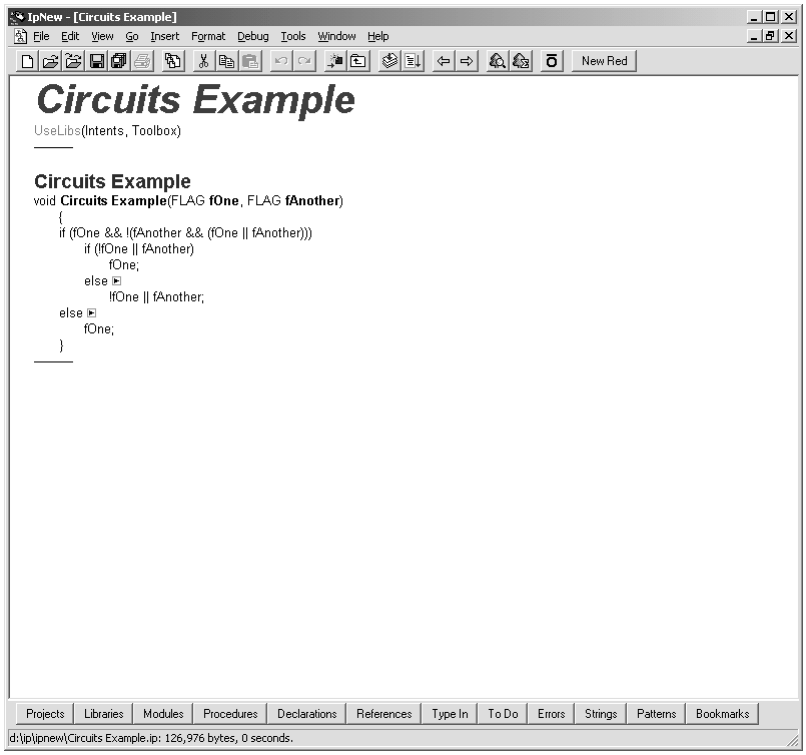


Figure 11-16 *C-like, textual rendering of logical expressions*

array containing the numbers that represent pixels. Furthermore, you can simply capture the bitmap somewhere from the screen and paste in the source.

IP rendering also allows you to enrich programming notations with any kind of GUI controls. A very common kind of controls used in programming are tables (e.g., decision tables). An example of a special table control used in program source is shown in Figure 11-20. This table is used to manage the specifications of menus for the GUI of the Microsoft Outlook product.

The IP editor lets you embed references to declarations in a comment. For example, a comment about some function may contain references to other similar functions. References basically amount to hyperlinks because you can use the go-to-declaration button to jump to the declaration being referenced. References embedded in comments are rendered underlined. This is shown in Figure 11-21. For example, one of the comments starts with `gregsh:`, where `gregsh` is a reference to the declaration declaring the developer Greg, who is the author of this comment. The

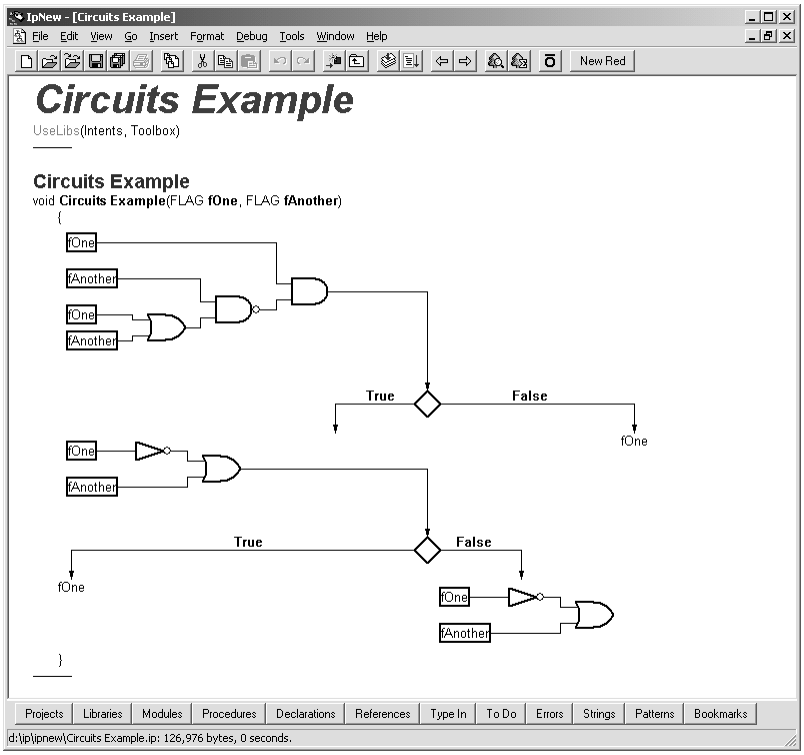


Figure 11-17 *Graphical rendering of the logical expressions from Figure 11-16*

reference `FotTypeface` is a reference to the declaration of the function `FotTypeface()`. The important point about references embedded in comments is that—just as in the case of references embedded elsewhere—you don't need to update their names after changing names of the declarations being referenced (e.g., after changing the name of the function `FotTypeface()`).

Aspectual views

The ability to provide alternative renderings for a single source is in the spirit of aspect-orientation and is a useful addition for implementing aspects by separate modules. You can use alternative renderings to show certain aspects and suppress other aspects of the source. For example, you can use alternative renderings to suppress the display of exception specifications in function signatures, when needed. It is important to note that not only can we compute aspectual views on the underlying source graph, but also support their editing. In general, renderings can provide radically different views of the source including program visualization and metrics.

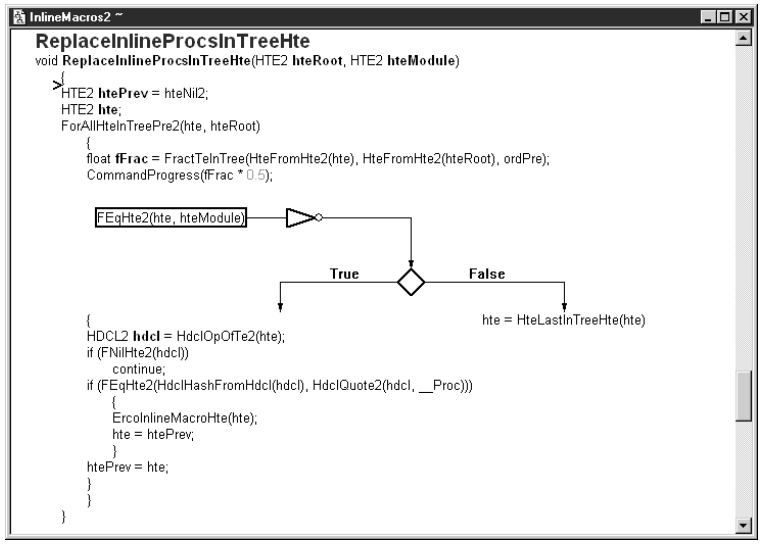


Figure 11-18 *Example of mixing textual and graphical notations*

Renderings in different languages

Another exciting use of alternative renderings is the possibility of rendering intention names in different languages, such as English or Chinese. This is illustrated in Figure 11-22 and Figure 11-23. This feature is very useful in the age of globalization because many organizations already engage in program development that involves teams from different countries (e.g., outsourcing the development of system components to different countries).

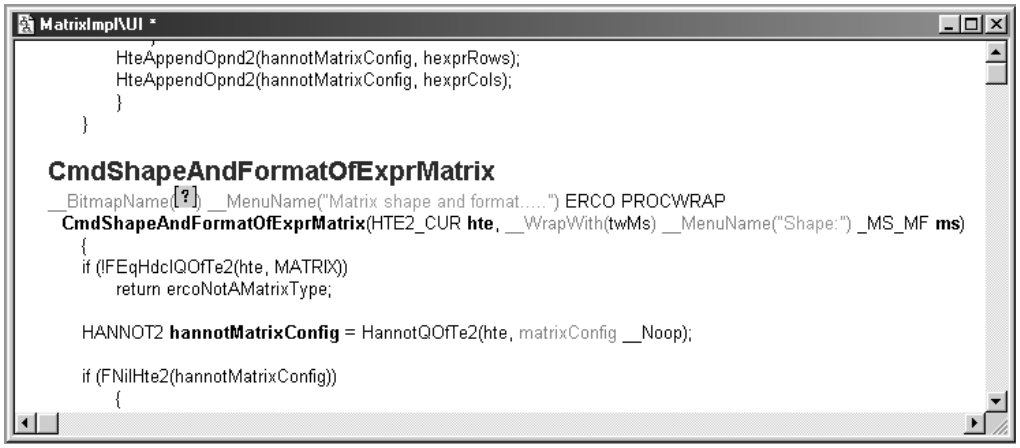


Figure 11-19 *Example of passing a bitmap constant as an argument*

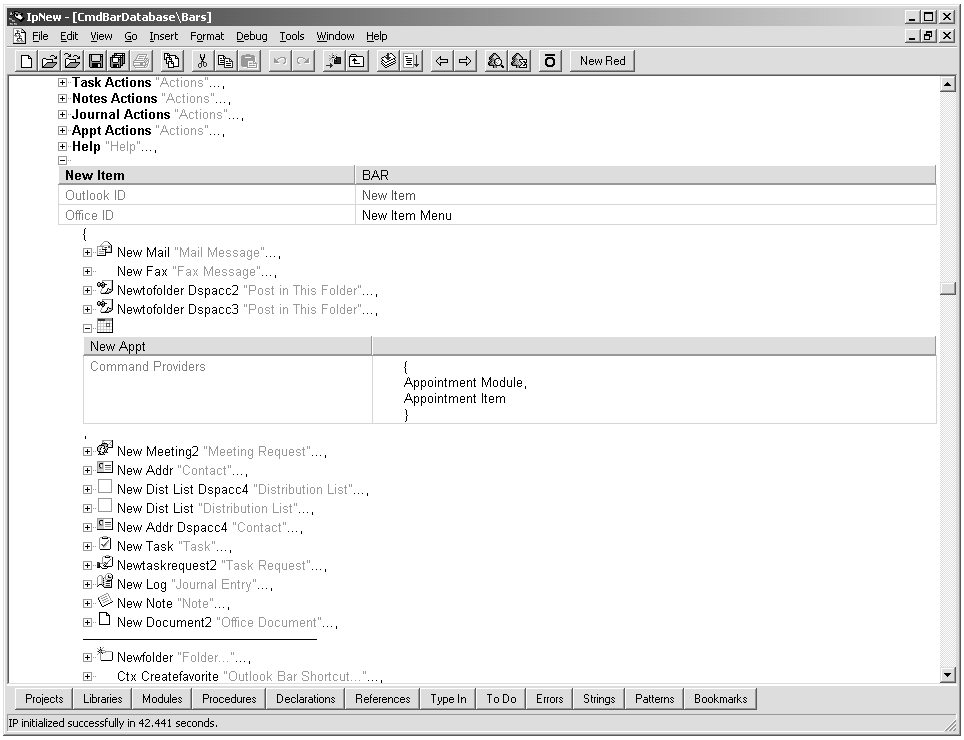


Figure 11-20 *Example of source containing a table control*

NOTE

We said earlier that important information that would otherwise make a great comment can often be captured in a machine processable way by introducing appropriate intentions. So you may ask why we still need comments. There are several reasons why comments are still useful. First, you often want to communicate something informal to other humans (e.g., "this line of code makes me feel so proud . . ."). Second, you may not have the time to state something formally (i.e., in a machine-processable way). Third, the intentions needed to state the information formally may be unavailable and defining them may not be an option. Why? *(1)* Maybe you don't have the time or skills. Remember: Intentions have to be well designed, and its not everyone that should introduce new intentions, but only intention designers, that is, language and library designers. *(2)* Or the intention could be so unique (i.e., of low reusability) that it would not be worth investing the effort. There is one important point about comments in IP, though: You can always come back and rewrite them in a machine-processable form when the necessary intentions become available.

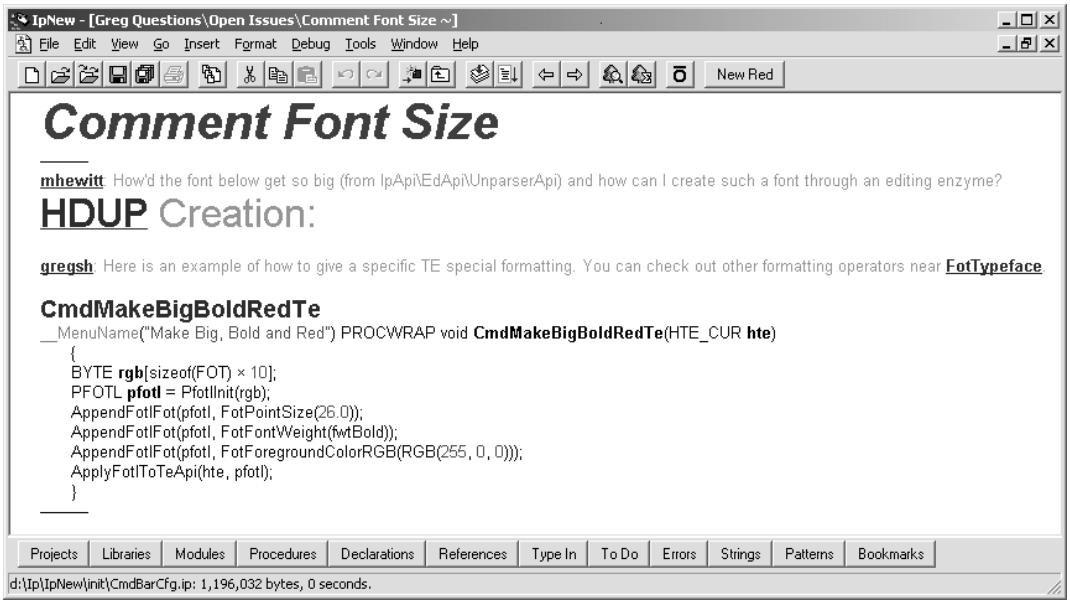


Figure 11-21 *Example of comments with references to declarations*

Core view

Most renderings display only a select part of the information contained in the source. For example, even the "Hello World" program shown in Figure 11-6 does not show all the detail contained in the underlying source graph. However, IP provides a special rendering called the *core view*, which shows more details of the underlying source graph. Intention programmers can use this rendering to debug the underlying program representations, but the intention users need not worry about it. The "Hello World" program in the core view is shown in Figure 11-24.

11.4.3 Extending the IP System with New Intentions

When you write an application, you simply use the general-purpose and domain-specific intentions provided by the extension libraries you've loaded into the IP system. Developing extension libraries is a different activity than application programming, however. This activity involves utilizing extension APIs and adhering to special IP protocols (e.g., the reduction protocol discussed in Section 11.5.5) and requires language design and implementation skills.

Library interfaces and extension DLLs

When implementing new intentions, you usually declare them in a separate file called the *library interface*. Next, you implement the reduction, rendering, type-in, debugging, and other methods in one or more other files. In most cases, you'll need at least the

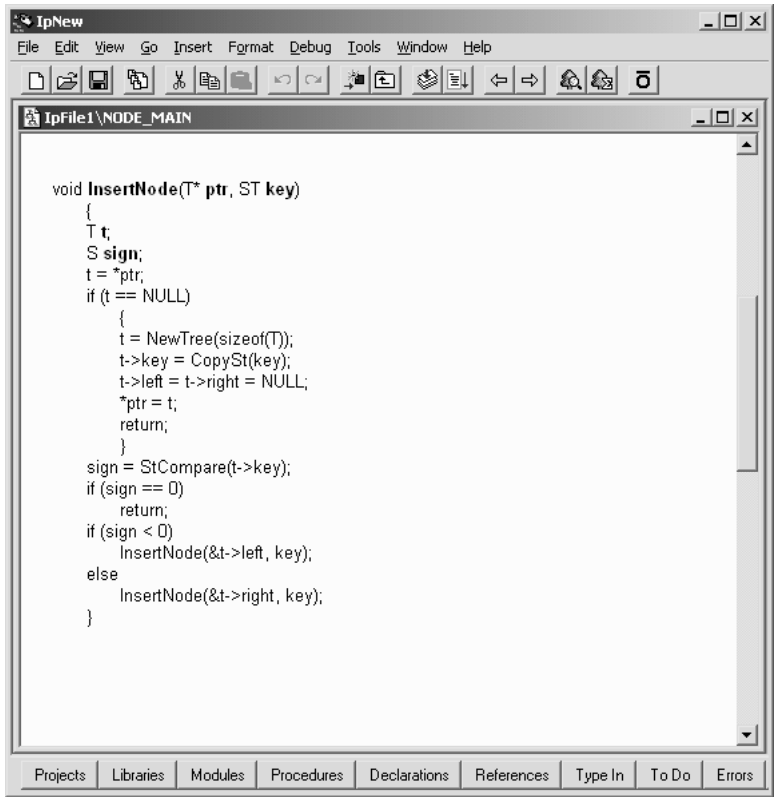


Figure 11-22 *A C function rendered in English*

reduction, rendering, and type-in methods. The methods use the extension APIs, that is, the reduction API, rendering API, type-in API, and so on. The files containing the methods are compiled into an *extension DLL*. You would usually package related intentions, for example, intentions implementing a domain-specific notation, in a single DLL. The interface file and the extension DLL get handed out to the application programmers. When the application programmers import the interface module of an extension library into their application project, the corresponding extension DLL gets loaded automatically into their IP system. The DLL contains the code needed to type in, render, and reduce the application program using the new intentions. The DLL can also contain special commands for working with the new notation (e.g., typing aids, analysis tools, and so on), which can automatically be made available on the IP menu bar after loading the DLL.

Let us take a look at the steps performed by a typical reduction method. A reduction method first analyzes the context of the tree

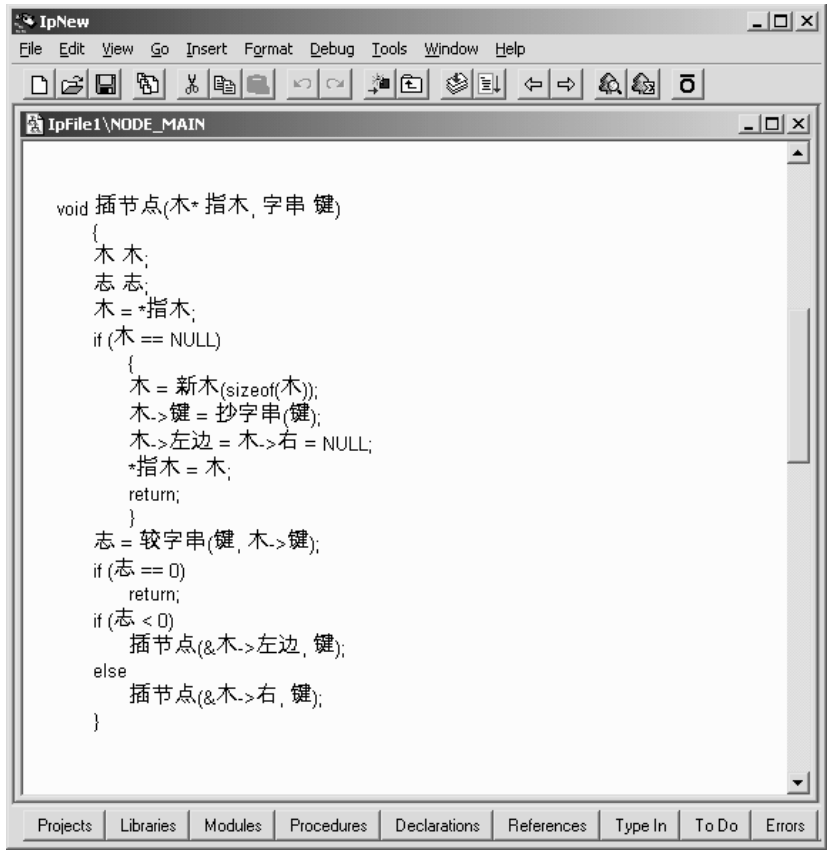


Figure 11-23 *The C function from Figure 11-22 rendered in Chinese*

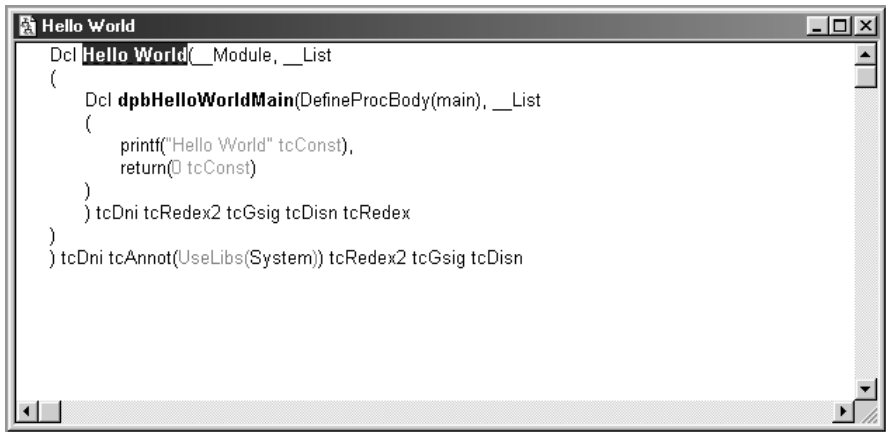


Figure 11-24 *The "Hello World" program from Figure 11-6 rendered in the core view*

element it was called on. It checks to see if the structure of the subtree is correct, so that it can then reduce it. It basically looks for syntax and semantic errors. If it discovers any errors, it attaches error annotations to the locations in the source where they are present. Even if there are no errors, it will still attach some other information gained in the analysis to various tree nodes in the source, so that other transformations can take advantage of it in later phases. In general, intentions can gather information from remote corners of the program in order to do various optimizations. The subtree is then reduced by adding lower-level representations to it (as discussed in Section 11.5.5). Finally, the reduction method returns the reduced representation of the source.

Accessing and modifying the source tree is done using the tree editing API, which consists of basic operations, such as create node, set the link to declaration, add an operand, and so on. In addition to this low-level tree editing API, there are also some higher-level facilities, such as pattern matching functions and quote constructs. The latter allow a compact definition of trees used for matching or reduction. As an example, assume that we are implementing an intention for representing matrices, that is, the type `MATRIX`. Furthermore, assume that we design the type so that the application programmer can declare a variable of type `MATRIX` and annotate the type with configuration parameters, such as the memory allocation strategy and the shape of the matrix. For example, the application programmer could declare a dynamically allocated, rectangular matrix as follows:

```
configuration(dynamic, rectangular) MATRIX m;
```

The implementation of `MATRIX` would, among other methods, require a reduction method for declarations with type `MATRIX`. Depending on the configuration parameters, this method would reduce a variable declaration with type `MATRIX` to a variable declaration whose type is a C array (for statically allocated matrices), or a C struct containing the number of rows and columns and a pointer to the matrix elements (for dynamically allocated matrices), or some other C data structure. The core of this reduction method could look like this:

```
HTYPE htype;
if (matrix_description.allocation == dynamic &&
    matrix_description.Shape == rectangular)
{
    htype = `struct
        {
```

```
        int rows;
        int cols;
        $htypeElement* elements;
        };
} else { ... }; //other cases
```

Quote and unquote operators

`htype` is a handle to a tree element that represents the type to which `MATRIX` gets reduced. `matrix_description` is a struct that was created during the analysis of the configuration parameters of the matrix declaration shown earlier. The statement inside the if-then branch assigns `htype` a tree representing a C struct. The struct contains the number of rows and columns and a pointer to the matrix elements. Instead of constructing this tree using the low-level tree editing API (i.e., calling the operations to create a node, setting the reference to declaration, adding the operands, and so on), we simply write the C code to be created preceded by the quote operator `` ` ``.[9] Once we have the tree representing the C data structure, we would attach it to the original source. Next, we would reduce the C intentions by calling their reduction methods.

Using the tree editing API and the simple metaprogramming constructs, such as quote and unquote, are still quite low level and tedious. However, you can implement more sophisticated, declarative notations for defining language extensions and provide them to intention programmers as extension libraries.

11.5 Advanced Topics

In the following five sections, we'll describe some more advanced concepts including questions, methods, and reduction. These topics are relevant to implementers of extension libraries, not to application programmers.

NOTE

Please note that `htypeElement` is a variable computed elsewhere and is used in the quoted code. In order to use its value, the variable is preceded by the unquote operator `$`.

9. The quote construct suppresses the standard reduction of the quoted code. Instead, the quoted code is reduced to code, which, when executed, actually creates the quoted source tree. The IP quote facility is analogous to the quote found in Lisp.

11.5.1 Questions, Methods, and a Frameworklike Organization

Questions

In IP, methods are invoked by asking *questions* of nodes. Questions are polymorphic operations on tree elements. Similar to polymorphic operations in object-oriented languages, you can have several different methods (i.e., implementation codes) associated with a single question (i.e., polymorphic operation). If you ask a node a question, there is a built-in lookup mechanism that will find the appropriate method to answer the question.

You can implement new intentions by declaring them and implementing the methods that define their semantics, appearance, and so on. As already discussed, the declarations and the methods go into separate modules. The module containing the declarations is called the library interface. The module with the methods gets compiled into a dynamic-link library (DLL), called an extension DLL, which can be dynamically linked to the IP system to extend it. If you want to use the new intentions in a program, you import the interface module, which will trigger the IP system to automatically link the corresponding extension library to itself. The system will now use this library to render, compile, and debug the instances located in your program that refer to the intentions in the declaration module.

When implementing extension libraries, you use a set of standard APIs defined by the IP system for rendering, type-in, reduction, versioning, and so on. These APIs consist of (1) declarations of procedures that you can call and (2) questions that the system calls and you provide methods (i.e., implementations) for. The questions are the entry points where user-defined code for rendering, type-in, reduction, and so on gets called. The system provides default implementations of these questions (i.e., default methods), which you can override. For example, the default implementation of the rendering question (i.e., the question returning the display representation of a source tree) is to display source trees in a functional notation. So, the tree in Figure 11-2 would be rendered by the default rendering method as follows: `+(x,x)`. If you want it to be rendered as `x+x`, you need to override the default rendering method for references to `+`.

Rcode question

An example of a question for which you usually need to provide your own methods when implementing new intentions is the reduction question, that is, the `Rcode` question. The system asks this question of the root node of a user program in order to initiate its reduction. The result of this question is the reduced version of the program, that is, a graph containing only instances of the R-code intentions for a given target platform.

As you can see, the IP system is structured like a framework making calls to default and user-defined methods, where the latter

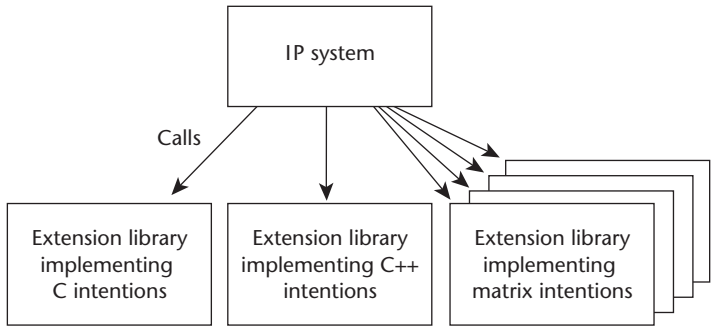


Figure 11-25 *The IP system calls extensions libraries*

are located in extensions libraries. This is illustrated in Figure 11–25.

In addition to standard, system-defined questions, extension libraries may also define their own new questions. For example, the implementation of the C intentions defines the question `Type`, which returns the type of an expression. Methods implementing the `Rcode` question ask the `Type` question during the static analysis of the C code being reduced.

11.5.2 Source-Pattern-Based Polymorphism

As stated earlier, if you ask a node a question, there is a built-in lookup mechanism that will find the appropriate method to answer this question. Which method gets selected depends on the structure of the source graph the node lives in. This is so because you can register different methods to answer a given question for different node patterns in the source graph. We say that questions are polymorphic on source patterns.

When a node is asked a question, the system first checks to see if there is an appropriate method registered for the node pattern the node lives in. If not, the question is resubmitted to the declaration of the node. This is a kind of a method inheritance mechanism specially designed for source graphs.

It is important to be able to register methods for different source patterns because we want to define different behaviors for different constellations of instances of intentions. For example, we need different code to display and compile:

- A reference to a declaration of a type
- A declaration with that type
- A reference to a declaration with that type

Suppose that you want to implement the type `int`. First, you need to declare `int`, and then implement and register the `Rcode` methods (i.e., the methods answering the `Rcode` question) for the following tree patterns: references to `int`, declarations with type `int` (i.e., declarations of variables of type `int`), and references to declarations with type `int`. You don't need to implement any rendering and type-in methods because the default ones are good enough for `int`.

The system defines the question `Rcode` as follows:[10]

```
Rcode(linkAsking, phteRcodeRoot)
```

This question returns the reduced version of the subtree it is called on. The first argument, that is, `linkAsking`, is the link that is asking this question. (When a node asks its neighbor the `Rcode` question, you can think of the question as traveling along the graphlike or treelike link connecting both nodes. This is the "link asking a question".) The second argument, that is, `phteRcodeRoot`,[11] is a pointer to where the method activated by the question will store the result, that is, a handle to the root of the reduced tree. A method implementing the question can refer to the node it operates on using `hteThis`, which is similar to the pseudovariable `this` in C++.

As stated previously, you need to implement and register methods handling this question for different source patterns. For example, you need a method handling the question `Rcode` for references to declarations with type int:

```
<ref_to.dcl_with_type_that_is.int>::Rcode(linkAsking,
                                          phteRcodeRoot)
{
  ... //implementation of the method
}
```

Let us take a look at the whole picture (Figure 11-26). The user program (on the left) contains a declaration with type `int`, a reference to the declaration of `int`, and a reference to the declara-

10. The code samples shown here are slightly simplified.

11. The naming conventions used in the IP system are called Hungarian notation [SM91]. For example, "phte" stands for "pointer to a handle to a tree element." (A handle is a machine-independent implementation of a pointer.) Hungarian notation was invented by Charles Simonyi and propagated through the Microsoft Windows API.

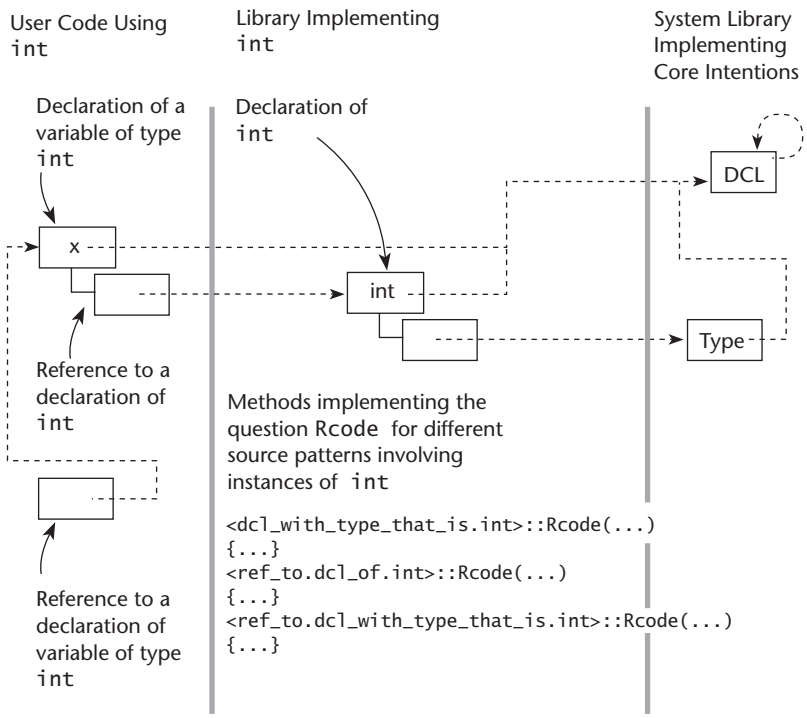


Figure 11-26 *An implementation of an intention consists of a declaration and methods registered for different source patterns involving instances of this intention*

tion with type `int`. The library in the middle implements `int`. First, there is the declaration of `int`, which would be located in the interface part of the library. This interface part would be imported by the user program (like a header file in C). Then you have the three methods implementing the `Rcode` question for the three different patterns involving instances of `int`. These methods would be compiled into an extension DLL and dynamically linked into the IP system. When you compile the user program, the system will call the corresponding reduction methods from the extension DLL.

11.5.3 Methods as Visitors

In IP, a method is not simply a piece of code called on one node, but rather it has the form of a visitor traversing the nodes of the pattern the method is registered for. The visitor may execute some user-defined code on each node being traversed. The accumulated

data is passed along the traversed path. The implementation of methods as visitors is not surprising because a large portion of code of any compiler deals with traversing ASTs.

There are cases where you only need to collect data from one node. For example, consider the question `Type`, which returns the type of an expression. When implementing the addition operator +, you need to implement a method handling this question for references to +. The method will be a visitor visiting only the reference node. During this visit, it will ask the question `Type` of the children of this node and then compute the return type. However, there are also cases where a visitor needs to traverse several nodes and execute different code on each of them.

11.5.4 Asking Questions Synchronously and Asynchronously

Questions can be asked synchronously or asynchronously. When you ask a question synchronously, you give the control to the system, and when the call returns, you have the answer. For example, a node can ask its neighbor the `Rcode` question as follows:

```
AskQuestion("Rcode", linkAsked, phteRcodeRoot);
```

In this call, `linkAsked` is the link connecting the node asking the question to the neighbor being asked, and `phteRcodeRoot` is the return parameter.

Alternatively, you can ask questions asynchronously. In this case, you first submit a question, continue with your work, and later ask for the result, which may cause blocking until the system has answered the question. The idea of asynchronously asking questions is that you can first submit several questions and then, when you block waiting for the first answer, the system may compute the answers to all the submitted questions in any order it chooses. Thus, you should call questions asynchronously whenever the order of answering the questions is not relevant to your code. For example, the method answering the question `Type` for a reference to + will ask its children the question `Type`, and then compute the resulting type. Because it does not matter whether we first ask the left child or the right child, both questions are asked asynchronously:

```
//submit the first question and return a handle to it
hsq1 = SubmitQuestion("Type", hteLeftOperand);
```

NOTE

As of this writing, the IP visitor mechanism is implemented using closures.

```
//submit the second question and return a handle to it
hsq2 = SubmitQuestion("Type", hteRightOperand);
//get the results
WaitForQuestion(hsq1, phteLeftType); //at this point the
                                     //system may compute
                                     //the answer to hsq1 and
                                     //then hsq2 or
                                     //the other way round
WaitForQuestion(hsq2, phteRightType);
```

The ability to delegate the order of answering questions to the system is very important because it promotes the composability of extension libraries. We'll explain this in the following section. It also creates the opportunities for parallel execution on multiprocessor machines.

11.5.5 Reduction[12]

As stated earlier, reduction is the process of transforming source trees into lower-level trees. It corresponds to compilation in conventional programming environments. Reduction is initiated by the system when it asks the question `Rcode` of the root node of a user program. This question computes and returns the R-code representation of the program for a given platform. This computation involves asking the `Rcode` and many other questions of other nodes in the source. In particular, these questions check the structure of the program source for correctness, perform optimizations, and generate the R-code representation. It is important to note that optimizations in IP can be and often are domain-specific because domain-specific abstractions provide their own reduction and optimization methods. Furthermore, optimizations in IP may inspect remote parts of the source tree, that is, we can have optimizations of wide scope. Of course, the process of reducing a higher-level abstraction can take advantage of the existence of lower-level abstractions. For example, the reduction method of a new high-level abstraction may first generate its implementation in C-abstractions, and then ask this implementation for its R-code.

The reduction process is governed by a few general principles. First of all, reduction methods are not allowed to delete links and nodes in the source, but only to add new links and nodes. The idea is that any new information computed by the reduction methods

12. This section describes the IP reduction protocol as of this writing. In particular, the version described here supersedes the one described in [ADK+98].

(e.g., typing information, intermediate representations, R-code representations, and so on) has to be attached to the source. If you ask a node a question for some representation, the question will first check to see if the representation has already been computed and is present in the source, and then it would return the existing answer or compute it if no answer was available. In other words, you can think of reduction as a process of growing the source graph until it incorporates its low-level R-code implementation. At that point, the source graph contains the original source, all the intermediate representations, and the R-code implementation. By not deleting any information during reduction, the process is monotonic, and you can control its progress. If the reduction methods were allowed to delete information, you basically could not always guarantee that reduction would terminate. An example of attaching lower-level representations to the original source is shown in Figure 11-27. This figure shows the reduction of `while` to an implementation using `if` and `goto`. The new implementation (shown in gray) is simply attached to the original tree. Because the condition of `if` is the original condition of `while` and the original body statement of `while` is also part of the body statement list of `if`, both trees share these statements. This is achieved through special docking points provided by treelike links.

The second principle is that nodes are only allowed to directly access their local neighborhood. More precisely, a method executing on a node is allowed to add links only to this node and access only the nodes to which the original node has direct links to or the nodes that were passed as parameters to the method. If a method needs any information from any remote nodes or needs to add links to remote nodes, it has to ask questions of their own direct neighbors, which may need to ask questions of their neighbors, and so on until the target nodes are reached. The idea behind this principle is that, because the question mechanism crosses the system, the system can automatically monitor which nodes acquire information from or add links to which other nodes. This way, as the reduction proceeds, the system builds a map of dependencies in the source (which is basically the record of who asked which question of whom). You'll see in the following paragraphs what this is good for.

The third principle is that the answer to a question may never change during the reduction. Even if a node asks a particular question of some other node only once, it has to be guaranteed that if the node asked that question again, it would get the same answer. This is not simple to guarantee at all because methods can add new links, that is, they can have side effects. For example, if a node asked some other node about its number of children and got the

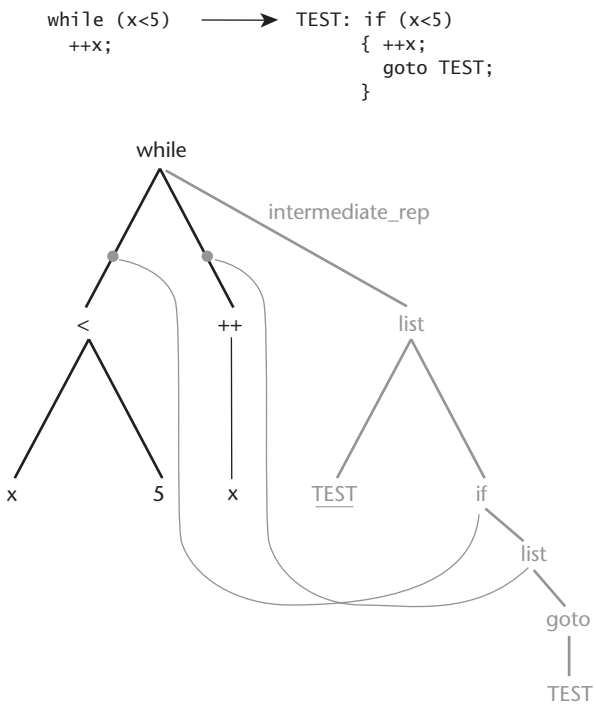


Figure 11-27 *Reduction of* `while` *by attaching a lower-level representation using* `if` *and* `goto`. *Please note that only treelike links are shown and each node except for* TEST *(which is a declaration) and* `5` *is represented by the name of its declaration.*

answer 3, this answer is not allowed to change and effectively no other method is allowed to add any children to that node once this question has been asked. The system checks this principle automatically. Whenever new links are added to a node, all questions that were run on this node are automatically reevaluated by the system to check to see if the answer is still the same. If the answers are different, the previous answers to these and all other questions calling these questions are invalidated. As a result of this invalidation, the system rolls back the reduction to the point where no answers are invalid and then tries asking the questions in a different order. Please remember that you can ask several questions asynchronously (see Section 11.5.4)—that's why the system has the opportunity to answer some questions in an order selected by the system. In other words, reduction involves a search for an order of answering questions that does not lead to the invalidation of any previous answer. In our example, if the question trying to add an extra child and the

question about the number of children were asked asynchronously, the system would make sure that the question adding a child would run before the question about the number of children.

The basic idea behind this whole reduction protocol is that a method can acquire anything it needs to perform its job through asking questions, and it is guaranteed that the answers to the questions it asks remain valid and won't be silently invalidated by some other method. Because nodes can acquire information from or modify remote nodes *only* through asking questions, and the answer to a question cannot change, the nodes actually acquire a view on the source that is static throughout the whole reduction process. In other words, the local changes performed by a method occur based on a view of the source that is still valid at the end of the reduction. Thus, when a method starts its execution, it can basically assume that the source already has its final form except for the additions to be made by this method.

The reduction protocol is designed to support combining extension libraries from different vendors. By asking questions asynchronously, you avoid overspecifying the order of invoking transformations. This is very useful whenever you want to extend or modify the compilation of certain instances of intentions, for example, by plugging in an extension library that optimizes certain patterns in the intermediate representations generated by other libraries. The new extension library could require a specific order of transformations, so that it can view all these patterns at one point and transform them. In other words, some extension libraries may constrain the set of possible transformation orders of other libraries. Because libraries support sets of alternative transformation orders rather than enforcing just one particular order, there is a higher probability that there is a transformation order that works for a composition of several libraries.

Of course, it is possible that the sets of transformation orders supported by several libraries are disjoint, meaning that no order can be found and the reduction process fails. However, this can only happen whenever the libraries want to transform instances of the same intentions and do it in incompatible ways. Such incompatibilities have to be resolved by the vendors of these libraries. The system at least guarantees that no garbage code will be generated due to such library interaction.

In order to minimize the search for valid transformation orders during reduction, future releases of the IP system will allow libraries to specify preferred orderings of transformations that are more likely to yield success.

11.6 The Philosophy behind IP

11.6.1 Why Do We Need Extendible Programming Environments? or What Is the Problem with Fixed Programming Languages?

Most real-world applications require many domain-specific abstractions. There are at least two obvious alternative ways to deal with this requirement. One solution is to use a general-purpose programming language with abstraction mechanisms, such as procedures or objects, which you can use to implement your own libraries of domain-specific abstractions. This is the conventional and widely-practiced solution. The second solution is to provide one comprehensive application-specific language for each kind of application you need to build. By saying a *comprehensive* application-specific language, we mean that the language contains dedicated language features for representing all the domain-specific abstractions needed for the given kind of application. It turns out that both solutions have severe problems, which we describe in the following two sections.

11.6.1.1 Problems with General-Purpose Languages and Conventional Libraries

There are four main problems with the "general-purpose programming language and conventional library" approach: loss of design information, code tangling, performance penalties, and no domain-specific programming support. Let us take a look at each of them.

- *Loss of design information:* When you use a general-purpose programming language, you have to map domain-specific abstractions onto the features and idioms of the programming language. The resulting code usually includes extra clutter and fails to represent the abstractions intentionally because some of the domain information is lost during this transformation. For example, there are many ways to implement the *singleton* pattern [GHJV95].[13] And given a particular implementation code only, it is not 100 percent certain that the intention of the code is to implement the singleton pattern. This information could be included in a comment, but such information is lost to the compiler. On the other hand, with the domain-specific (or application-specific) language approach, we would provide the special class annotation *singleton*, which would allow us to unambiguously express the singleton intention. The loss of design information makes software evolution

13. Singleton is an idiom of OO languages for implementing classes that can have only one instance.

extremely difficult because change requests from the customers are usually expressed at a higher level of abstraction. Using the general-purpose programming language and conventional library approach, program evolution requires code analysis to recover the intended abstractions, which, as we illustrated with the singleton concept, is impossible to automatically perform fully.

- *Code tangling:* Programming problems are usually analyzed from different perspectives and an adequate, intentional encoding should preserve the separation of perspectives (e.g., separating synchronization code from functional code). As we saw in Section 8.7, in most cases, achieving this separation requires domain-specific transformations, but such transformations cannot be encapsulated in conventional libraries unless the language supports static metaprogramming.[14] In other words, we need to put some code extending the compiler into the library, but this is usually not supported by current library technologies. Thus, when using conventional procedural or class libraries, we are forced to apply these transformations manually. In effect, we produce tangled code, which is difficult to understand and maintain. We investigated these issues in Chapter 8 in great detail.
- *Performance penalties:* The structure of the domain-level specification does not necessarily correspond to the structure of its efficient implementation. Unfortunately, the main property of procedures and objects is that they preserve the static structure of a program into runtime. A compiler for a general-purpose programming language can only apply simple optimizations because it only knows the level of that language, but not the domain level. For example, as we discussed in Section 9.8.1, no compiler could possibly optimize the call `EXP(x,2)` to the exponentiation function `EXP()` (which is implemented using the Taylor expansion formula) into `x*x` or optimize `EXP(0.5,x*x)` into `x`. In general, a considerable amount of domain-specific computation at compile time might be required in order to map a domain-level representation into an efficient implementation. With the general-purpose programming language and library approach, no such computation takes place (again, this would require static metaprogramming).
- *No domain-specific programming support:* Domain-specific abstractions usually require some special debugging support (e.g., debugging synchronization constraints), special display and

14. We saw in Chapter 10 that compile-time metaprogramming is possible in C++ in the form of template metaprogramming. However, template metaprogramming, although Turing complete, has only very limited program structuring constructs.

editing support (e.g., displaying and editing pretty mathematical formulas), and so on. Such support requires that libraries, in addition to the procedures and classes to be used in client programs, also contain extensions of the various components of the programming environment. However, current programming technologies do not support such extensions.

11.6.1.2 Problems with Comprehensive Application-Specific Languages

Given a comprehensive application-specific language containing all language features we need for a given application, we could implement a programming environment with all the necessary optimizations and debugging, displaying, and editing facilities. The language itself would allow us to write intentional, well-separated application code. In other words, we would solve all the problems mentioned in the previous section. Unfortunately, there are four major problems with this approach: the parsing problem, high cost of specialized compilers and programming environments, problems of distributing new language extensions, and problems of evolving domain- and application-specific languages.

- *Parsing problem:* Conventional compiler technology uses parsing in order to transform program text into the internal program representation used in a compiler. Parsing poses two problems to feature-rich and domain-specific text-based languages. First, it is difficult or impossible to add more and more new language features to a language without eventually making it unparsable. As already discussed, C++ is a good example of a language reaching this limit (see Section 11.2). Second, the requirement of parsability represents quite a restriction on domain-specific notations. This is so because natural domain-specific notations often do not reveal all the detail of the underlying model and allow for views that contain too many ambiguities to be parsable. Mathematical books are full of notations that are impossible to be directly represented as conventional, text-based computer languages. As already discussed in Section 11.4.2, this is caused by the fact that they do not have to show all the details of the underlying representation (which is unambiguous). For example, when editing text in a WYSIWYG text editor, such as Microsoft Word, one does not see whether two paragraphs were assigned the same style (e.g., *body text*) because they could have the same text properties (e.g., font size, font type, and so on). An example of an unambiguous textual representation is the TEX file format [Knu86], where all the formatting information is included inline as special commands. However, viewing a TEX file in an ASCII

editor is not WYSIWYG. Other limitations of textual representation include being confined to one-dimensional representations, no pictures, no graphics, no hyperlinks, and so on.

- *High cost of specialized compilers and programming environments:* The cost of developing compilers and programming environments is extremely high. For example, in [Vel98a], Veldhuizen cites Arch Robinson, lead developer of Kuck and Associates, Inc. (which is the maker of the high-performance KAI C++ compiler), estimating the cost of compiler development at $80 and more per line of code (as of 1998). Given such a high cost, vendors of compilers for general-purpose languages usually do not have the resources to extend their products with domain- or application-specific features (e.g., domain-specific optimizations for scientific computing), which are only useful to a relatively small group of users. Developing dedicated programming environments for whole domain-specific languages is even more costly and definitely should not be undertaken by application developers. The only solution to this economic problem is to provide a common (meta-) programming infrastructure that can be reused across different languages and build a third-party market for specialized language extensions on top of it.
- *Problems of distributing new language extensions:* Even if we extend a language with new features, dissemination of the new features is extremely difficult because languages are traditionally defined in terms of fixed grammars and compilers, and programming environments do not support an easy and incremental language extensibility. That is why, as Simonyi notes in [Sim97], new useful features have the chance to reach a large audience only if they are lucky enough to be part of a new, widely disseminated language. A good example of such feature are interfaces in Java. The only problem is that such opportunities are quite rare. Furthermore, Java is also a good example of a language that is extremely difficult to extend because of its wide use and the legacy problem that comes with it. This is also the reason why, as of this writing, genericity is still not in the language. Another problem is that just as new features are difficult to add to a widely used language, bad features that are already in the language are difficult or impossible to get rid of.
- *Problem of evolving domain- and application-specific languages:* As Eric Van Wyk pointed out, domains evolve and "may evolve out of the coverage of any domain-specific language." As discussed in the previous point, fixed grammars, compilers, and language infrastructures make it difficult to evolve domain-specific languages to adequately support evolving domains.

11.6.1.3 The IP Solution: An Extendible Programming Environment

Fortunately, there is a third way to provide adequate support for domain-specific abstractions without running into the problems discussed in the previous two sections. The solution is to use an extendible programming environment, such as IP, which replaces the fixed-programming-language view with the idea of configuring your programming notation by composing active libraries implementing single or sets of language features (i.e., intentions). IP addresses the aforementioned problems as follows.

- Loss of design information and code tangling are avoided by providing domain-specific language extensions, that is, specialized language constructs that capture your domain-specific abstractions intentionally. In IP, language extensions are packaged into active libraries called extension libraries, which you can load into the programming environment in order to extend it.
- Performance penalties are avoided by applying domain-specific optimizations, which are distributed as a part of the extension libraries.
- Domain-specific programming support (e.g., domain-specific debugging, editing, displaying, and so on) can also be provided as a part of the extension libraries.
- The parsing problem is solved in IP by abandoning the textual representation altogether and allowing direct and unambiguous entry of AST nodes using commands. Furthermore, thanks to rendering, any kind of notation can be supported.
- Some of the high development cost of specialized compilers and programming environments is reduced by providing a common reusable (meta-) programming platform (i.e., the IP system), so that only the language extensions themselves need to be programmed. Furthermore, you can usually reuse a given language feature with different configurations of other language features. The interoperability between the language features has to be provided by the vendors of the language features or frameworks of language features, which may require special extension libraries with glue code.
- Extension libraries represent a convenient and economical means for distributing language extensions. New features can be added to the programming environment as the application scales. For new development, you can easily exclude deprecated features by simply not loading them (although you can still load them for legacy code).
- IP provides a platform for language evolution driven by the evolution of domains and by market needs. Charles Simonyi

compares this evolution to the processes found in biological systems by referring to the future intention market as an "ecology of intentions" [Sim95].

11.6.2 Moving Focus from Fixed Languages to Language Features and the Emergence of an Intention Market

As Simonyi notes in [Sim97], with IP we have a major shift of focus from languages to language features, that is, intentions. Currently, new language constructs have to look for a host language and this is quite difficult because the most popular languages are difficult to extend (this requires updating all the compilers, standards, manuals, and so on, which are not designed to be extendible). As we mentioned previously, new features can only spread through new and successful languages, such as Java. Unfortunately, not only good features reach large audiences this way. If a bad feature makes it into one of the widely used languages, it is difficult, or impossible, to get rid of it. The situation is very different in IP: Programming abstractions become true entities with their own "life." They have to survive based on their own merits. They encapsulate the knowledge they need to be displayed, compiled, and debugged in different contexts and can be easily distributed as extension libraries.

The research on design patterns and idioms gives a further motivation for the need of change of focus from languages to programming abstractions (see e.g., [GL98]). The pattern work attempts to classify new useful domain-specific and general programming abstractions and mechanisms. There is the conviction that programs are essentially assembled from these fundamental building blocks. On the other hand, as more of such patterns are identified and documented, it becomes increasingly difficult for any language to express them adequately. Few of these abstractions make it into languages. For example, dynamic polymorphism or inheritance require implementation idioms in C, but they are part of OO languages. However, hardly any existing language could keep up with the explosion of new abstractions.

The vision of IP is the emergence of an intention market. In such a market, there will be intention vendors, who will develop new intentions and will have to make sure that these intentions cooperate whenever there is a need for it. Obviously, there will be different categories of vendors, for example:

- Vendors providing frameworks of intentions implementing general purpose programming and modeling notations.

- Vendors providing frameworks of intentions implementing domain-specific programming and modeling notations.
- Smaller vendors providing nifty, innovative extensions to the existing notations.

Given such a market, language abstractions are no longer "looking for" host languages, but rather for customers [Sim97]. With all the critique of current programming languages, they are still very important from the IP viewpoint: They are sources of useful language features and notations. As Simonyi predicts in [WTH+99], the development of the intention market will probably start with vendors providing extension libraries implementing existing programming languages (such as Java, C++, COBOL, and so on), and then supplying useful additions to them (e.g., genericity, preconditions and postconditions, procedure specialization, and so on). The next step will be the development of domain-specific intentions.

It is important to note that IP creates an enormous development potential in several areas including novel debugging mechanisms, specialized rendering and editing support, domain-specific optimizations, language-specific refactoring support and so on, which goes much beyond the mere evolution of programming languages we have today. This is so because it gives third-party vendors a common infrastructure and a common internal source representation for implementing and distributing language extensions and language-based tools, which they didn't have before.

11.6.3 Intentional Programming and Component-Based Development

Intentional Programming provides advanced support for Component-Based Development (CBD). As a particular focus, in addition to promoting building applications from reusable, replaceable parts, IP enables automating their assembly in unique ways.

Now, you may wonder: What is the relationship between IP and component standards, such as CORBA, COM, or JavaBeans? First, let us take a look at the similarities (for brevity, in the rest of this section, we refer to CORBA, COM, or JavaBeans components as just "components").

- Both intentions and components are building blocks used by application programmers to build applications.
- Just like intentions, components may also have design-mode methods used to enter, visualize, and manipulate them at programming time.
- Both intentions and components support visual programming.

- They both have globally unique identifiers allowing their global distribution.

However, there are also several important differences. In contrast to IP, most CBD environments based on the CORBA, COM, or JavaBeans standards have the following deficiencies.

- They only allow you to create assemblies of components whose structure will be preserved into runtime. In other words, transformations implementing domain-specific optimizations and weaving and distributed as part of the components are not supported. This usually leads to poor separation of concerns and/or poor runtime performance.
- They do not support domain-specific debugging capabilities distributed as part of the components.
- They are not capable of mimicking the traditional, textual way of programming using tree editing like IP does, that is, they only provide visual programming as a direct way to manipulate components. (Alternatively, you can use traditional textual, scripting languages to glue components, but such code has all the problems of traditional programming, for example, it has no adequate support for refactoring, extendibility, and so on).
- They still need conventional, textual languages to code the components. In contrast, IP does not have this conceptual discontinuity: Everything (including intentions) is coded using intentions and all the source can enjoy all the advantages of intentional encoding.

Put another way: If you implement a component-based programming environment that:

- Contains a code-transformation framework for generating efficient code from design-time assemblies of components and that supports the integration of independently developed transformations
- Provides a special support for metaprogramming including debugging of transformations
- Provides a set of standard APIs for extending any part of the environment (including the debugger)
- Supports not only visual programming, but it also views traditional, textual language features as components and perfectly mimics the traditional type-in of such features and
- Is completely implemented in itself, that is, it does not need any traditional programming technology to implement any of its components

then you've got another IP.

11.6.4 Frequently Asked Questions

IP represents quite a radical paradigm change departing from many current programming traditions. Therefore, it isn't surprising that there are a number of questions frequently brought up in discussions about IP.

Q1: General-purpose programming languages are commonly understood. On the other hand, each new domain-specific notation needs to be learned first. Isn't the cost of learning new domain-specific notations prohibitive?

If you code a library of domain-specific abstractions in a general-purpose programming language, the library user will need to learn the domain concepts behind the library in order to be able to use it. The understanding of the general-purpose programming language will be of little help in understanding the domain concepts. Thus, learning a conventional library of domain-specific abstractions is much the same as learning a new domain-specific language. By using a domain-specific language instead of the conventional library approach, you get many advantages, however. Problems expressed in a domain-specific language are often more concise and contain less clutter. An encoding in a domain-specific language is more intentional because you don't have to take the detour of coding idioms and patterns to express key domain concepts, that is, domain knowledge doesn't get lost. Finally, you get all the advantages of domain-specific support including domain-specific optimizations, natural notations, domain-specific error reporting and debugging (see Table 8-2 in Section 8.7.1). For example, the STL [MS96] is famous for causing long and cryptic error reports (due to the long and complicated identifiers generated during template instantiation) when you have an error in your application code using STL. With domain-specific error support, the library would actually be able to issue clear-text, domain-specific compile-time error reports telling you what is wrong with the way you used a given container in your program.

It is often said that a common general-purpose language promotes communication between developers. Yes, this is true for the general-purpose programming mechanisms it provides. But, this advantage can be easily carried over to domain-specific, extensible languages: If necessary, a domain specific notation can be based on widely known general-purpose programming mechanisms. Furthermore, just as standard libraries help to foster a common communication basis beyond the syntax of their implementation language, standard domain-specific notations will emerge and serve the same purpose.

Q2: Using simpler languages is easier and makes clearer programs. IP propagates feature-rich languages. Isn't programming in feature-rich languages more complicated and doesn't it result in more complicated programs?

First, we need to be clear about our goal: Do we want to have a simpler language or one that simplifies the task of writing a given program? Probably the latter. Assembler is a simple language with few language features, but writing and maintaining complex assembler programs can be a nightmare. We definitely want to use an optimal set of language features for a given problem, that is, we don't want to use more features or more complicated features than necessary. (As Albert Einstein once said: "Make it as simple as possible, but no simpler than that.") Unfortunately, simplicity is often confused with primitiveness.

An intense discussion about simplifying languages has been sparked by Java, whose simplicity compared to C++ is considered its main advantage. Java definitely makes programming certain kinds of applications simpler (e.g., due to its automatic memory management). However, because certain useful general-purpose features are missing, some classes of problems are not as easy to code in Java. For example, due to missing genericity, programming with containers is quite a tedious task. Furthermore, the intentionality of the code suffers. For example, you cannot express enumerations in Java intentionally—you have to simulate them using constants, and therefore, you don't get specialized type-checking as in C++ (this and other painful omissions from Java are discussed in [Gil99]). This is not a critique of Java as a language. First, Java can be seen as a useful language redesign based on other object-oriented languages. Language redesign and evolution is very important and extendible programming environments like IP just make it simpler. Second, Java is a fine, well-selected set of features including some new useful features (e.g., interfaces). And it should be seen as that. That is, you can use it for a given problem if it fits it well, but you should be able to extend it if you need to. Extendible programming environments do not prevent you from using a well-selected set of features for your problem. In fact, they make it simpler because you can also add problem-specific features, which are generally missing in general-purpose languages, but can greatly simplify the programming task. Furthermore, as your application scales and evolves, the "set of optimal languages features" also changes. If you are using an extendible programming environment, you can evolve this set more easily.

Another argument on language simplicity brought up by Guy Steele in his excellent and entertaining OOPSLA'98 keynote

[Ste98] is that simple[15] languages enforce clarity and promote understandability by using fewer special terms and being more verbose (i.e., by showing more of the definition of terms). In other words, he supports the standpoint that domain-specific languages are cryptic. (As Mason Cooley once said: "Jargon is any technical language we do not understand.") Yes, domain-specific languages are cryptic to those that do not know a given domain. But then, they should either learn it or should not be programming in that domain. We feel that only beginners can be helped with somewhat more verbose encoding. However, if this is the goal, then you can provide a verbose rendering of the code in IP (i.e., the intentions can be rendered in a more verbose fashion, showing more of their definitions). By using verbose encoding (rather than just verbose rendering), however, you loose design information because you don't get unique handles on the important domain concepts in the code, but rather code patterns representing them. Finally, to experts, verbose renderings are awkward and annoying.[16]

At last, what can be more simple to use than a domain-specific representation, which was specially designed to make the task at hand simpler?

Q3: Won't letting any programmer extend a language create a notational havoc?

The problem of opportunistic language extensions is pertinent to languages with an extensive, built-in support for metaprogramming, such as CLOS or Smalltalk. Such languages make it easier for an application programmer to include metacode implementing some ad hoc language extensions directly in the application code. The problem is that, although metacode makes the base code simpler, it is usually itself inherently more complex, harder to debug, and errors in the metacode often have a more severe impact on a system than errors in the base code. On the other hand, well-designed and reusable metacode can greatly simplify application development.

The situation in IP is quite different than in languages with built-in metaprogramming capabilities. This is because IP makes a

15. In the sense of "primitive."

16. In fact, Steele's OOPSLA99 keynote transcript [Ste98] demonstrates just that. The keynote itself is written using a primitive subset of English, which is also defined in the paper. The intention was to show how far one can go with a primitive language. Despite an impressive demonstration of this point, to a fluent English speaker, the keynote text appears awkward. Thank God we don't have to communicate using a primitive English in everyday life!

clear separation between programming and metaprogramming. Application programmers use IP together with general-purpose and domain-specific extension libraries that optimally support their job. Writing language extensions (i.e., extension libraries) is a separate and completely different activity than application programming. It involves utilizing extension APIs and adhering to special IP protocols and requires language design and implementation skills. Developing extension libraries will be the business of library vendors, not application developers.

Just as standard, conventional libraries emerge in different domains today, standard domain-specific notations will emerge. (Many fields already have their domain-specific notations, IP will just help to implement them as sharable and embeddable sets of programming abstractions.) In the presence of an intention market, developers will have access to high-quality and sophisticated domain-specific intentions, so that the temptation to invent their own ad hoc solutions will diminish. An intention market will promote strong specialization, which will allow us to build systems of greater quality and complexity.

Finally, having a common platform for language extensions will help us to eliminate "islands of insulated domain- and application-specific languages," which are common today.

Q4: How about the interoperability of extension libraries? Doesn't the problem of feature interactions (i.e., adding a new feature may easily break the language) render extending programming languages impractical?

Kinds of language extensions

Before answering how IP addresses the problem of feature interactions and what the challenges are, let us first take a look at the two possible kinds of language extensions.

- *Encapsulated language extensions:* Encapsulated language extensions do not affect the semantics of the language features of the language being extended. Technically, this means that an encapsulated language extension does not contribute transformations that would participate in the compilation of the language features in the language being extended. An example of such an extension is embedded SQL, where the compilation of the host code (e.g., written in Java) and the embedded SQL code are two different activities. The only requirement is that the embedded SQL code returns values of types understood by the host language. Composing encapsulated language extensions is analogous to composing well-encapsulated components (such as JavaBeans, COM, or CORBA components).

- *Language extensions with external influence:* Language extensions with external influence affect the semantics of the language features of the language being extended. For example, a language extension with external influence could contribute wide-scope, domain-specific optimizations, which also transform instances of the language features of the language being extended. Other examples of features with wide influence are evaluation mode (eager versus lazy evaluation), exception handling, and memory management. In other words, language extensions with external influence have an aspect-oriented character.

Kinds of language feature interactions

In the context of language extensions, we can have three kinds of feature interactions.

- *Syntactic feature interactions:* A language extension could introduce parsing ambiguities and make the resulting language unparsable.
- *Semantic feature interactions between encapsulated language extensions:* This kind of feature interactions is the same as feature interactions between conventional, encapsulated components (e.g., JavaBeans, COM, or CORBA components).
- *Semantic feature interactions involving language extensions with external influence:* These are the hardest kind of interactions to deal with. It involves coordinating transformations across several components (possibly from different vendors).

Now, we can take a look at how IP helps to resolve each of these three kinds of feature interactions.

NOTE

It is important to realize that "components with metaprogramming capabilities = language features." Indeed, you could build a source tree out of a set of JavaBeans or COM instances and association relationships. The links to declarations would be the instantiation relationships between the components and the instances. You could construct this tree in a COM or JavaBeans component builder. If the component hierarchy is deployed as is (i.e., the structure of the components is preserved into runtime), you've got the typical functionality of a component-based development environment. However, if the components are asked to generate code and perform transformations, you can think of each of the components as a language feature! Put another way, the requirement to support traditional components and metaprogramming (i.e., aspectual components, automatic configuration, and so on) blurs the borderline between components and language features!

- *Syntactic feature interactions:* This problem is completely solved in IP by abandoning parsing altogether.
- *Semantic feature interactions between encapsulated language extensions:* As stated previously, these kind of feature interactions are the same as feature interactions between conventional, encapsulated components. The usual way of resolving such interactions is to provide glue code taking care of the incompatibilities. The same strategy can be applied in IP. But, in IP, we can do even better than that. Conventional glue code introduces additional levels of indirections incurring runtime penalties. With IP, we can apply domain-specific transformations to eliminate this overhead, that is, we get zero-overhead glue code. However, it is important to note that this approach often involves turning encapsulated extensions into ones with external influence. On the other hand, the latter are the specialty of IP. Finally, IP gives us another great advantage when it comes to embedding domain-specific language extensions. For example, when using a conventional database technology, such as JDBC (Java Database Connectivity), syntax errors in embedded SQL queries are not detected at compile time, but at runtime (when the query is sent to and evaluated by the server). However, if you implement embedded SQL as a true language extension of the host language (which is how you do it in IP), syntax errors in a query will be detected during the compilation of the client.
- *Semantic feature interactions involving language extensions with external influence:* As, stated earlier, these are the hardest kind of interactions to deal with. This problem involves coordinating transformations across several extension libraries (possibly from different vendors). IP addresses this problem with its sophisticated reduction protocol. As we discussed in Section 11.5.5, extension libraries may provide partial orderings of transformations and let the IP system find a valid total ordering for a set of extension libraries. Expressing language extension in terms of partial orderings of transformations rather than total ones increases the chance that a given set of extension libraries will cooperate. Practically, this means that an extension library may request a particular transformation order and will look at some transformation results of other libraries. Of course, this kind of library interaction (i.e., where transformations from different vendors try to reduce the same instances of intentions) will require some kind of cooperation between the library vendors, for example, they will need to agree on some common interfaces and protocols.

As you can see, IP improves the situation in all three cases. However, there is still a great need for new research on language

extensibility. The idea of modularly extendible languages is only beginning to gain popularity in the research community (e.g., see [Hud98, DNW+99, KFD99]). In particular, we need to work on domain models of general-purpose language abstractions and mechanisms (i.e., studying which combinations of features are possible, which are useful, which are preferred, and so on). Furthermore, we need more work on language features for encapsulating aspects in the context of other features (such as the work on aspect-oriented features in object-orientation as discussed in Chapter 8). We need to study interfaces between features and work towards standard protocols for metaprogramming. Work on typing components that implement language features is an active research area (see e.g., [MPW99]). Such type systems will allow an extendible programming environment to determine whether a given configuration of language features is compatible or not.

Q5: With program source expressed as text, I see all the information contained in the source at once. By introducing internal representations and rendering, I usually don't. Isn't it confusing?

In our mind, seeing all the detail of a complex system at once is confusing. In IP, you can, but usually don't want to view the core rendering of the source (see Figure 11-24). Just as in the case of a CAD system, the separation of internal representation and rendering gives us the opportunity to build even more complex systems because we are not forced to work in the most-detailed view all the time.

Q6: Some people still haven't made the transition to OO. IP seems to be a completely new paradigm. Isn't IP too big a paradigm change to digest for the average programmer?

A very exciting property of IP is that you can introduce it in an evolutionary way. The only necessary initial investment in terms of training is teaching your developers to work with the IP editor, which is not much. At this basic level, your developers can continue working with the same language(s) and paradigm(s) as they did previously. You basically import all the legacy code into IP and use IP as a development environment. Even at this basic level, you already get several of the following benefits.

- You get the benefits of tree editing including more efficient typing and the ease of changing names.
- You get browsing information that is present and up-to-date all the time.
- You get all the refactoring functionality that comes with IP.

In other words, the basic level is to use IP as an efficient forward and reverse engineering environment. At the next level, as new and

useful general-purpose and domain-specific language extensions, refactoring, and other tool extensions for your language appear on the market, you can use them to improve your code. Finally, as you collect more and more expertise in a given domain, you may consider developing and marketing your own extension libraries.

It is important to note that IP is not displacing any existing language paradigms (such as the object-oriented paradigm), but only helps to better harvest their benefits by promoting multiparadigm programming and the evolution of language paradigms.

11.7 Summary

The main advantages of IP stem from the way IP represents domain concepts, which is summarized in Figure 11-28. The structure of a domain concept is given by the structure of its source graph. Its external representation is defined by the rendering and type-in methods and its semantics is given by the reduction methods.

This separation has profound consequences. If you represent a specific concept using a source graph, the structure of the graph represents the most invariant part of the concept. But, the concept's external view and semantics may depend on the context: The concept may be displayed differently at different times (e.g., editing or debugging time) or it may be displayed differently depending on its source context (i.e., on the way it is used in a program). The generated code may depend on the tree context, on the platform, and so on. Given this separation, it is usually enough to modify or extend the reduction methods to implement new optimizations and adapt the client code to a new context, new platform, and so on without having to modify the client code at all.

If we need to change the source representation itself, it is easier to do this than to change a textual representation. Tree editing has many advantages over text editing, such as fewer syntax errors,

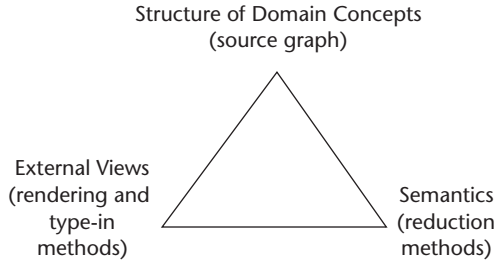


Figure 11-28 *Separation of structure, external representation, and semantics in IP [Sim98]*

active interaction with the intentions during editing, and often less typing (e.g., using additional, short intention names for efficient typing; using "display-only artifacts" to avoid retyping signatures of remotely declared procedures and methods; using automatic type-in templates; and so on). Also, automatic refactoring and reengineering of code is easier if the source is already in the form of a resolved AST.

Rendering allows different views, formatting conventions, special notations, graphical representations, the translation of names into different national languages (e.g., English or Chinese), and so on, and together with the binary source graph representation, it gives an opportunity for realizing a true document-based programming (as in Knuth's literate programming [Knu92]). You can embed animations and hyperlinks in the comments, use pretty notations for the code, embed bitmaps and controls, and so on. At the same time, the reduction methods may perform complex computations in order to generate highly-optimized code for these programs.

At last, after learning so much about the IP technology, let's go over the benefits of IP again.

- It enables the achievement of natural notations, great flexibility, and excellent performance simultaneously.
- It provides optimal, domain-specific support in all programming tasks (supports effective typing, rich notations, debugging, error reporting, and so on).
- It addresses the code tangling problem by allowing you to implement and easily distribute aspect-oriented language features.
- It allows you to include more design information in the code and to raise its intentionality.
- It helps with software evolution by supporting automated editing and refactoring of code.
- It makes your domain-specific libraries less vulnerable to changes on the market of general-purpose programming languages.
- It can be introduced into an organization in an evolutionary way with exciting benefits for the minimal cost of initial training.
- It supports all of your legacy code and allows you to improve it.
- It provides third-party vendors with a common infrastructure and a common internal source representation for implementing and distributing language extensions and language-based tools.
- It promotes domain specialization and facilitates more effective sharing of domain-specific knowledge.